AD-A204 907

EXPERT SYSTEMS ON MULTIPROCESSOR ARCHITECTURES: Phase One Final Report

Stanford University

DTIC
ELECTE
2 3 FEB 1989
S
E
D

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

89 2 23 029

This report has been reviewed by the RADC Public Affairs Division (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.
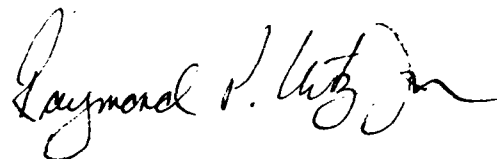
RADC-TR-88-187 has been reviewed and is approved for publication.

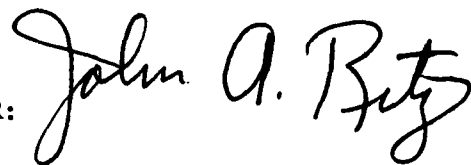APPROVED: *Northrup Fowler III*

NORTHRUP FOWLER III
Project Engineer

APPROVED: *Raymond P. Urtz, Jr.*

RAYMOND P. URTZ, JR.
Technical Director
Directorate of Command & Control

FOR THE COMMANDER: *John A. Ritz*

JOHN A. RITZ
Directorate of Plans & Programs

EXPERT SYSTEMS ON MULTIPROCESSOR ARCHITECTURES:
Phase One Final Report

Edward A. Feigenbaum
Robert S. Engelmore

Contractor: Stanford University
Contract Number: F30602-85-C-0012
Effective Date of Contract: 14 Mar 85
Contract Expiration Date: 13 July 89
Program Code Number: 8E20

Short Title of Work: Expert Systems on Multiprocessor
                       Architectures
Period of Work Covered: 14 Mar 85 - 13 July 87

Prinicpal Investigator: Edward A. Feigenbaum
                 Phone: (415) 723-4878

Project Engineer: Northrup Fowler III
             Phone: ((315) 330-7794

Approved for public release; distribution unlimited.

ADA204907

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b RESTRICTIVE MARKINGS N/A

2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	Approved for public release; distribution unlimited.

4. PERFORMING ORGANIZATION REPORT NUMBER(S) N/A	5. MONITORING ORGANIZATION REPORT NUMBER(S) RADC-TR-88-187

6a. NAME OF PERFORMING ORGANIZATION Stanford University Knowledge Systems Laboratory	6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Rome Air Development Center (COES)

6c. ADDRESS (City, State, and ZIP Code) Computer Science Department 701 Welch Rd, Bldg C Palo Alto CA 94304	7b ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700

8a. NAME OF FUNDING/SPONSORING ORGANIZATION Defense Advanced Research Projects Agency	8b. OFFICE SYMBOL (If applicable) ISTO	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F30602-85-C-0012

8c. ADDRESS (City, State, and ZIP Code) 1400 Wilson Blvd Arlington VA 22209-2308	10 SOURCE OF FUNDING NUMBERS			
	PROGRAM ELEMENT NO. 62301E	PROJECT NO E291	TASK NO 00	WORK UNIT ACCESSION NO 12

11. TITLE (Include Security Classification)
EXPERT SYSTEMS ON MULTIPROCESSOR ARCHITECTURES: Phase One Final Report

12. PERSONAL AUTHOR(S)
Edward A. Feigenbaum, Robert S. Engelmore

13a. TYPE OF REPORT Interim	13b. TIME COVERED FROM Mar 85 TO Jul 87	14. DATE OF REPORT (Year, Month, Day) August 1988	15 PAGE COUNT 346

16 SUPPLEMENTARY NOTATION
N/A

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Expert Systems, Architectures,
12	05		Parallelism, Blackboard Systems
			Multiprocessor, parallel processing (KT)

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

The military has a demonstrated need for knowledge-based systems with significantly higher quantitative performance. The current hardware and software architectures for knowledge-based systems cannot support such requirements. Based on near-term projections for integrated circuit technologies, it is clear that highly parallel multiprocessor computers consisting of 100s to 1000s of processors and realizing a variety of concurrent architectures can be built. The most promising approach for achieving orders of magnitude improvement in the quantitative performance of knowledge-based systems is by exploiting concurrency on multiprocessor systems. The major issue is whether such computer systems can be effectively used to enhance the performance of knowledge-based systems. Expert systems on Multiprocessor Architectures project is addressing the following questions: (1) Can multiprocessor computers be used to achieve significant execution speedup (two to three orders of magnitude) over serial machines for knowledge-based system applications?

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT ☐ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a NAME OF RESPONSIBLE INDIVIDUAL Northrup Fowler III	22b TELEPHONE (Include Area Code) (315) 330-7794	22c OFFICE SYMBOL RADC (COES)

DD Form 1473, JUN 86 — Previous editions are obsolete

(2) What are the limiting factors in achieving speedup for such systems? (3) What are appropriate software models and methodologies for programming such systems? (4) What are appropriate hardware architectures for supporting such systems? Given the lack of any formal foundations for studying concurrent knowledge-based systems, the approach taken to answering these questions is empirical rather than theoretical. The research methodology is: (1) Select specific knowledge-based system applications, primarily signal understanding applications. (2) Encode these applications following various proposed concurrent software models. (3) Evaluate the qualitative and quantitative performance of the applications running on simulated multiprocessor machines with respect to varying hardware parameters, for example, number of processors and communication protocols, and varying software organizations, for example, degree of control centralization. This report summarizes the activities and results of the major components of this project during the 27-month Phase One effort that commenced in March of 1985. Included as appendices are copies of the major research articles stemming from this project during that period.

Table of Contents

List of Figures

ii

1 Introduction

The military has a demonstrated need for knowledge-based systems with significantly higher quantitative performance. The Pilot's Associate, for example, will require knowledge-based systems that can cope with large amounts of data and that produce responses in real-time. The current hardware and software architectures for knowledge-based systems cannot support such requirements. The most promising approach for achieving orders of magnitude improvement in the quantitative performance of knowledge-based systems is by exploiting concurrency on multiprocessor systems.

Based on near-term projections for integrated circuit technologies, it is clear that highly parallel multiprocessor computers consisting of 100's to 1000's of processors and realizing a variety of concurrent architectures can be built. The major issue is whether such computers can be effectively used to enhance the performance of knowledge-based systems. Since 1985, the Knowledge Systems Laboratory at Stanford University has been investigating this issue. More specifically, our Expert Systems on Multiprocessor Architectures project is addressing the following questions:

1. Can multiprocessor computers be used to achieve significant execution speedup (two to three orders of magnitude) over serial machines for knowledge-based system applications?

2. What are the limiting factors in achieving speedup for such systems?

3. What are appropriate software models and methodologies for programming such systems?

4. What are appropriate hardware architectures for supporting such systems?

Given the lack of any formal foundations for studying concurrent knowledge-based systems, the approach that we have taken to answering these questions is empirical rather than theoretical. Our research methodology is:

1. Select specific knowledge-based system applications, primarily signal understanding applications.

2. Encode these applications following various proposed concurrent software models.

1

3. Evaluate the qualitative and quantitative performance of the applications running on simulated multiprocessor machines with respect to varying hardware parameters, for example, number of processors and communication protocols, and varying software organizations, for example, degree of control centralization.

This report summarizes the activities and results of the major components of our project during the 27-month Phase One effort that commenced in March of 1985.

2 SIMPLE/CARE Multiprocessor Simulation System

Simulation of systems at an architectural level can offer an effective way to study critical design choices if (1) the performance of the simulator is adequate to examine designs executing significant code bodies -- not just toy problems or small application fragments, (2) the details of the simulation include the critical details of the design, (3) the view of the design presented by the simulator instrumentation leads to useful insights on potential problems with the design, and (4) there is enough flexibility in the simulation system so that the asking of unplanned questions is not suppressed by the weight of the mechanics involved in making changes either in the design or its measurement.

SIMPLE/CARE [4] is a simulation system which satisfies these requirements. It forms the foundation for our empirical investigations of software architectures and hardware system architectures for concurrent knowledge-based systems. SIMPLE is a CAD (Computer Aided Design) system for hierarchical, multiple level specification of computer architectures and includes an associated mixed-mode, event-based simulator. CARE is a parameterized, multiprocessor array emulation specified in SIMPLE's specification languages and running on SIMPLE's simulator. Our simulation system is in use by several research groups at Stanford, and it has been ported to several external sites including NASA Ames Research Center.

2.1 The Design of SIMPLE/CARE

The overall research problem motivating the development of both SIMPLE and CARE is the performance study of 100 to 1000-processor multiprocessor systems executing knowledge-based signal interpretation applications.

A set of constraints pertinent to this problem governed the design of SIMPLE/CARE. The applications represent significant bodies of code and so simulation run times are an important consideration. Moreover, the issues involved with the interactions of multiprocessor system elements are sufficiently unexplored

prior to simulation that simplifications in the architectural model, specifically with respect to processor interactions, are suspect. This need for detail is, of course, in tension with the need for simulation performance. The ways that simulated system components are composed into complete systems is difficult to bound. Further, it is clear that the models of these components are elaborated over time and undergo substantial change as design concepts evolved. It is also clear that the ways of examining the operation of these components would change independently (and at a great rate) as early experience indicates what alternative aspect of system operation should have been monitored in any given completed run.

The design goals that emerged are (1) that the simulation system should support the management of substantial flexibility with regard to simulated system structure, function, and instrumentation and (2) that, in order to accomplish runs in acceptable elapsed times, the detail of simulation should be particularly focused on the communications, process scheduling, and context switching support facilities of the simulated system -- that is, on just those aspects of system execution critical to multiprocessor (as opposed to uniprocessor) operation.

2.2 Architecture Design-time Interaction and Simulator Run-time Operation

Encapsulation of the state of design components with the procedures that manipulate that state is one clear way to manage architectural design evolution. Such encapsulation partitions the design along well defined boundaries. Components (by and large) interact with other components only through defined ports. Connections between components terminate at such ports. When a system simulation is initialized, connections are traced so that for every port, the simulator knows the connected (terminating) ports together with their containing components. Once such initialization is complete, that is, throughout the simulation run, assertions about the state of a port of one component can be directly translated to assertions about the state of connected ports of other components.

Partitioning issues of system structure, component behavior, and instrumentation into separate domains of consideration helps in managing a design that is both fluid and complex. System structure, that is, the relationship between components, can be specified through use of an interactive, graphics structure editor and is largely independent of component function per se. Figure 1 shows an example of SIMPLE's structural editor.

Component behavior is encapsulated in a set of definitions pertinent to the given class of component. Each component in a SIMPLE specified simulated system is a
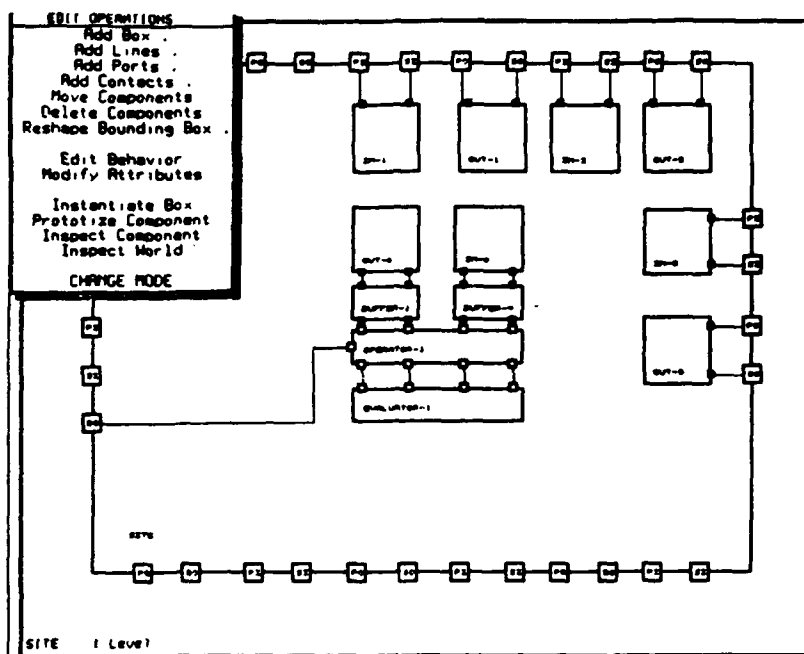
Figure 1: Graphic Structure Specification

member of a class defined for that component type. Instrumentation is automatically and invisibly made part of the definition of each simulated component that is to be monitored during a run. This is done by arranging that the class of every component to be monitored is a specialization of the general instrumented-box class. The basic data structures and procedures for monitoring simulated components and maintaining the organizational relationships between each component and its related instrumentation are inherited through this general, ancestral class and are thus made a separate, substantially independent consideration in the design.

A further partitioning of concerns is employed to separate out the definition of the application programming language interface and its support (as provided by CARE) from the underlying information flow control governing component behavior. The behavioral descriptions of components (which are expressed as sets of condition/action rules) deal generically with gating information, independently of the structure of the information, between ports of the component and its internal state variables. This is separated in the component model definitions from the

4

functions performed to create and manipulate the information so gated. The simulated implementation of the application programming language support facilities, on the other hand, relies only on the specifics of the information and its structure and plays no part in gating it between the components of the system. Changing the definition of the application language is thus done independently of changing component flow control behavior. The application programmer and the implementer of the application language interface may use whatever data structures seem suitable to them, be they numbers and keywords or procedure bodies and execution environments. The simulation system doesn't care.

The component probe definitions, that is, the specifications of what information should be captured for each component type, are separated from the descriptions of the behavior of such components. In designing for flexibility in the instrumentation system, it turns out to be important to further divide the information presentation from the information collection issues. The mapping from particular component probes to particular instrument panels and the transformations to be applied to the information as it passed from a given kind of probe to a given panel (and between panels) is captured in the instrument specification. This is a definition of what kinds of panels are included in an instrument, how they fit on an instrument screen, how they are labeled and scaled, and what information from which kinds of probes are displayed on each panel. The instrument specification also indicates what kinds of probes are to be connected to which kinds (that is, which classes) of components in the system.
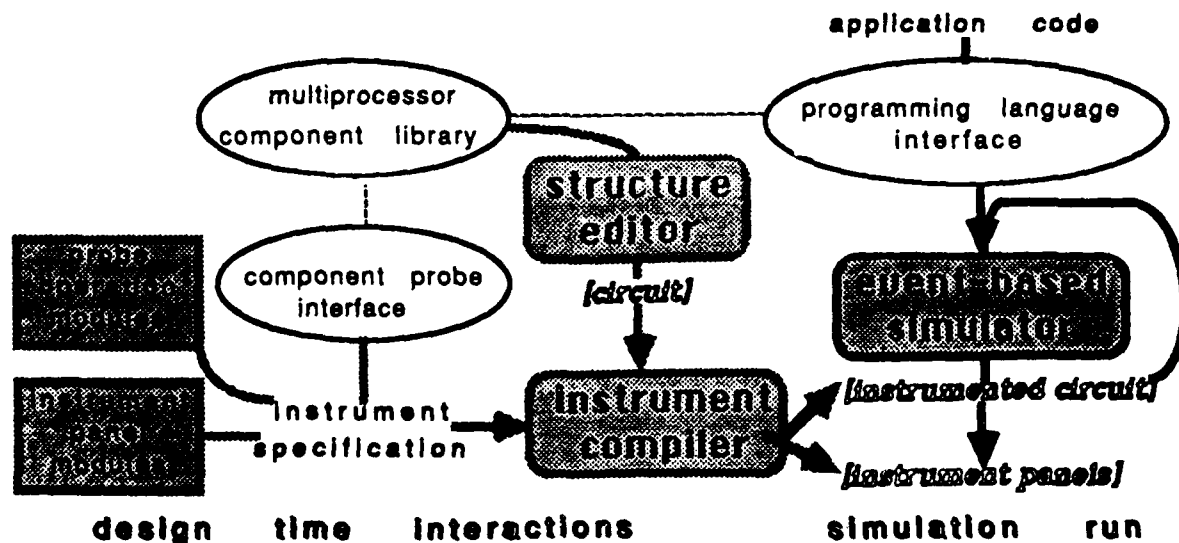

Figure 2: Design Time Interactions and Run Time Representations

5

Putting together all the definitions of components, component probes, panels, instruments, applications interfaces, and inter-component relationships is done in a set of design time interactions by a system architect. These interactions are used by the simulation system to generate efficient run time representations so that simulation performance goals can be met. Figure 2 illustrates the partition between design time interactions and simulation run time operation. Structure editing pulls together components from the component library to produce a circuit. Associated with some components in the library, there are definitions for the syntax and underlying mechanisms of a multiprocessor applications language. These specify the interface used to provide the program input to the multiprocessor system being simulated. The definitions used to generate component probes are associated with each library component to be monitored. There may be several such definitions, each appropriate to measuring a different aspect of the associated component's operation. An instrument specification selects from these definitions, elaborates them with selections from a set of probe operation modules to include any pre-processing (for example, a moving average) to be calculated by the probe, and indicates under what conditions what information from the probe is to be sent to which panels of the instrument and how it is to be transformed and displayed there. Instrument specifications also partition the screen among the panels of the instrument. The end product of these design time interactions is an instrumented circuit and an instrument. The instrument comprises a set of instrument panels and a set of constraints relating them to the instrument screen. The instrumented circuit ties together instances of components, probes, and panels for a simulation run. Figure 3 gives an example set of instrument panels for a run.

For each defined class of component and its associated probes, the design time interactions produce code bodies that accomplish simulation operations during a run. It is an attribute of the underlying Lisp base of the simulation system that changes in these definitions have immediate effect even during a simulation run -- an important capability during debugging.

3 LAMINA Programming Interface

LAMINA [3] provides extensions to Lisp for studying expressed concurrency in functional programming, object oriented, and shared variable models of concurrent computation. The implementation of the support for all three computational models is based on the common notion of a stream, a datatype which can be used to express pipelined operations by representing the promise of a (potentially infinite) sequence of values. LAMINA also provides system support for the
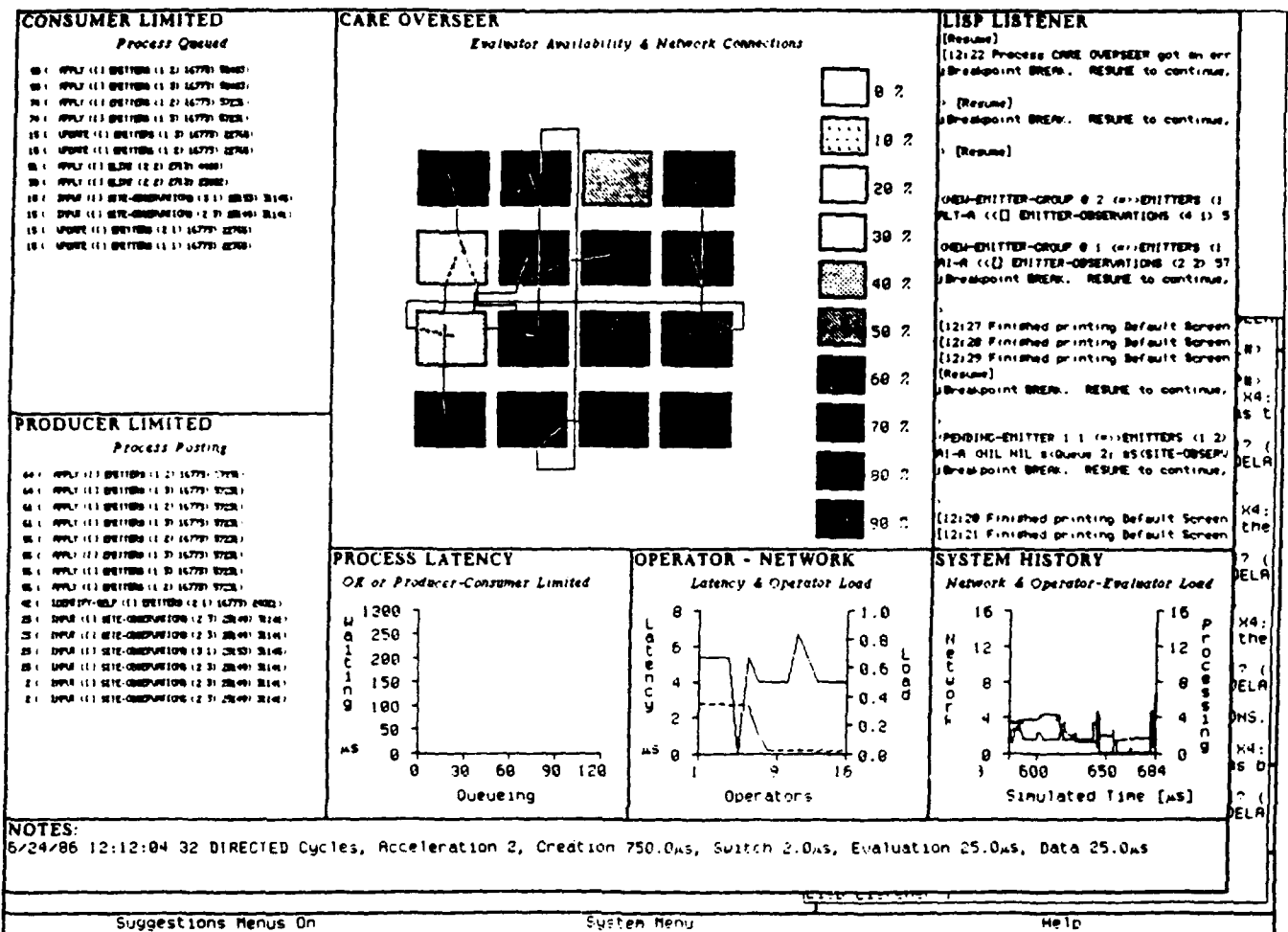
Figure 3: Overseer Instrument

management of software pipelines and dynamic structure creation, relocation, and reclamation in a multiprocessor, multi-address-space system.

Algorithms and applications written in LAMINA may be run on the SIMPLE/CARE simulation system in order to study their execution on alternative multiprocessor architectures.

7

3.1 Futures and Streams

Futures and streams provide the common ground between functional, object oriented and shared variable programming in LAMINA. They are fundamental to the LAMINA functional and object oriented programming regimes for parallel programming and, since they are the only mutable items passed as references (rather than structure values) between potentially concurrent computations in LAMINA, they are also used to build the mechanisms for shared variable computation.

Futures and streams represent promises for values. In LAMINA, futures can be used as placeholders in a computation while the values themselves are being eagerly produced by concurrent evaluations for consumption as available. Extending this idea, LAMINA defines a stream as an abstract data type which is a placeholder representing a sequence of eagerly produced but potentially unavailable values.

Some operators do not require the actual values promised by a stream or future in order to perform their work. For example, a constructor may create data structures that include streams as structure elements. The creation can be accomplished without accessing any of the promised values that the streams represent; referencing streams as placeholders is sufficient. Further, streams, as sequences of potentially unavailable but eagerly produced values, can be used in LAMINA to build pipelines of computation connecting the producers and consumers of such values.

Streams may be arguments to or the results of function application. In LAMINA, streams are a primitive data type developed for use in an object oriented programming style and futures are a specialization of streams that represent only a single (potentially unavailable) value as required for the functional programming style. Streams and futures are always passed as references.

3.2 LAMINA'S Models of Concurrent Computation

Perhaps the style of computation most readily treated as concurrent is that of functional programming. LAMINA supports concurrent programming using this style by providing means (1) to spawn computations that will provide values to futures and (2) to accept such values in a computation -- scheduling the computation when they are available. The constructs defining the LAMINA interface for functional programming are:

- (future form) spawns execution of a lexical closure, that is, a procedure body to execute a given form together with an environment (determined by the rules of lexical scoping) in which to do the execution. This closure is executed (eagerly) on a randomly selected site. A future which

8

will contain the value of the computation when it is available is immediately returned.

- (with-values future-bindings forms) spawns an evaluation on the local site to execute the closure corresponding to the forms. The evaluation is done within an environment that includes bindings for given variables to the values available for the indicated futures. The evaluation is deferred until all of the indicated futures have values that are not themselves futures. The immediate result of executing a with-values form is a future whose value will be supplied by the deferred evaluation.

In LAMINA's object oriented programming interface, an object encapsulates related state variables and is referenced throughout an application by that object's Self-Stream, a stream which is one of the object's state variables. Objects are allocated in a processor's local address space. To perform operations on an object, potentially involving and modifying its state variables, a task request posting consisting of a task selector and associated parametric values for the operation is sent to the object, that is, provided as one of the values of the self-stream for that object. Each of the task request postings that provide the values for the self-stream of a object is taken in turn from that stream and serviced by that object.

Task request postings are serviced atomically in the context of an object. Executions specified by such request postings are done without visible partition with respect to other operations on that object, that is, operations on any given object will not be interleaved. Each operation is thus defined to be independently atomic.

All the operations on an object done as specified by the requests are taken in turn from the object's self-stream. Each operation runs to completion. If an operation on an object is preempted (due, for example, to page faulting, schedule quanta lapse, or error condition), no other operation on that object will be started before the preempted operation is completed. However, operations on other objects may proceed normally. A stack is maintained for each preempted operation.

Shared variables are dealt with in LAMINA by treating them as references whose associated value may be mutated. A shared variable reference is constructed, accessed, and mutated by provided interface operations. Support for shared data pairs and arrays is also provided. For all these operations, execution is deferred and no other executions are performed by the initiating processor until the indicated operation is accomplished.

Shared queues (which are streams) are also provided. These queues are maintained

9

in a processor's local memory. When a process reads from a shared queue, it is halted and descheduled; execution is resumed when the requested data arrives. A simple spin lock is provided for busy-wait synchronization in the LAMINA shared variable interface.

Several utility operations are provided by LAMINA to specify computation (and storage) sites, dismiss computations, and provide a timeout facility for applications desiring one. LAMINA also provides simulation control facilities to initiate a CARE simulation, read the current simulation time, and do a computation without increasing the simulation time.

4 Poligon Problem Solving Framework

Poligon [9, 10] is a framework for the development of Blackboard-like applications on a (simulated) multiprocessor. It consists of:

1. A compiler, which compiles a high-level description of the Blackboard's structure and the Knowledge to be applied by the system, to run on a distributed memory multiprocessor.

2. A run-time system which provides a debugging and testing environment for Poligon programs as well as run-time support.

Both the compiler and the run-time system are thoroughly integrated with the program development environment of TI Lisp machines, the machine on which the execution of Poligon programs are simulated.

Serial Blackboard Systems are implemented with the Nodes being represented as records on the Blackboard. The Knowledge is encoded in Knowledge Sources. These are typically compiled into procedures which are invoked by the Blackboard System's kernel. There is some form of scheduler for the Knowledge, which invokes one Knowledge Source after another. The Blackboard and the Knowledge Base both share the same address space, though they are functionally distinct. Knowledge Sources are "invoked" (executed) as a result of changes in the Blackboard, placing that change event in a queue used by the scheduler. The scheduler repeatedly picks a Knowledge Source which is interested in the type of event at the end of the queue.

The design of Poligon has been motivated by the idea of trying to eliminate the bottlenecks that would be experienced : an existing, serial Blackboard System were to be parallelized only by the inclusior of "do this bit in parallel" constructs. The major changes from the serial blackboard model are listed below.

- The scheduling queue of a serial system is eliminated altogether in Poligon. This means that concurrent attempts to invoke Rules are not held up waiting for access to this shared data structure.

- Having a Knowledge Base, which is logically distinct from the Blackboard, is no longer necessary since there is now nothing to get between them to control the application of the knowledge. This allows all Knowledge to be attached to those Nodes that are interested in the Knowledge by the compiler.

These changes eliminate at one stroke the bottlenecks of the shared scheduler and the Knowledge Base to Blackboard interface. These changes allowed the development of the idea of the "Node as a processor" metaphor for parallel Blackboard systems.

Having eliminated the scheduling mechanism, however, one needs some means of determining when a certain piece of Knowledge should be invoked. It would be hopelessly inefficient to have all of the Knowledge executed all of the time, since most of the time it would find itself inapplicable. It was decided that a simple daemon-driven approach would be used to avoid this problem. This results in the Knowledge being directly sensitive to changes in the Blackboard and able to act immediately upon any such changes.

Existing Blackboard Systems often express the Knowledge in their Knowledge Sources as collections of Pattern/Action Rules. These are normally executed serially, in the lexical order in which they are defined. Poligon on the other hand compiles Knowledge Sources away all together, allowing their constituent Rules to be executed in parallel.

The "Node as a processor" metaphor is itself a major step away from the normal means of implementing Blackboard Systems. This, however, is not enough. This would give us data parallelism, resulting from the large number of Nodes in the system being able simultaneously to execute Rules, whilst still failing to exploit the potential Knowledge parallelism. This is because each processing element is a uniprocesor capable of executing at most one Rule at a time. Poligon, therefore, goes beyond this simple model to one which would more accurately be called the "Rule invocation as a process" model. This allows the Poligon system to distribute concurrent Rule invocations to different processing elements.

The elimination of serializing components in a Blackboard system also eliminates those mechanisms which are normally used to preserve coherency in the solution.

Clearly there is a trade-off which can be made between the amount of control and coherency preserving mechanisms and the amount of exploitable parallelism. Poligon is an experiment to explore one extreme of this spectrum. It remains to be seen whether the trade-off made in Poligon results in an overall improvement in system performance.

4.1 How Poligon matches the problem domain

Poligon is not a general purpose programming language, other than in the Turing Complete sense. It is specialized to support one computational model and that computational model, itself, has limitations on its sphere of reasonable applicability. It has been designed with applications such as real-time signal understanding and data fusion in mind, though applications outside this domain are being investigated.

The structure of the problem domain is one that requires the representation of a large number of distinct entities in the solution space. For example the vocabulary of the Elint problem domain [2] is full of such things as aircraft, radar emitting platforms and radar track segments. Poligon provides a rich representation language in which these objects and specializations of them can be expressed. This allows the system to take full advantage of the mutual independence of any of the objects in the solution space to exploit parallelism.

4.2 How Poligon matches its target hardware

Poligon could, of course, run on any machine in principle. In practice, however, it has been designed with a CARE type of machine model in mind and has been optimized to take advantage of it. The grain size of the executable chunks in Poligon programs is designed to suit this model, i.e. each chunk represents, ideally, a few function calls. This makes it coarser grained than those systems that want to execute everything that can be in parallel, for instance data flow machines, but it is a lot finer grained than most other concurrent Blackboard Systems in which each processing element contains a complete Blackboard System.

The target machine model, being of the distributed-memory, message-passing variety including essentially no capability to pass references, strongly discourages shared variables or mutable global data of any sort and encourages a message-passing style of programming. The Poligon language is one in which the programmer is given an abstract view of programming using the Blackboard Problem-Solving model. The Poligon language has no construct for message sending at all, nor has it any primitives by which the user has access to the underlying

12

architecture or topology. It is assumed to be the duty of the Poligon system or the target machine's operating system to look after such concerns. The Poligon compiler compiles its programs into the message passing primitives of the underlying system. This allows the efficient use of the underlying architecture, whilst still leaving the source program uncluttered by concrete details of the target architecture.

Poligon allows only global constants (but not variables) since these can be distributed at program load-time.

4.3 What we have learned to date

Experiments with Poligon are by no means complete, but we have learned quite a bit so far. Some of these lessons are enumerated below.

- It is very hard to write any program which implements either a framework, such as Poligon or an application such as those which have been mounted on Poligon. This is due largely to asynchronous side effects. A system with better formal properties would be less error prone in this respect but might well make much less efficient use of the hardware. These difficulties could also be caused by an insufficiency of mechanisms to control coherency in Poligon.

- In order to produce a *reliable program* it is necessary to write code which makes no assumptions about anything that any other part of the system might be doing. Failure to do so results in brittle systems.

- In order to achieve a coherent solution it was found to be necessary to develop a number of programming methodologies.

Node Level   The creation of Nodes is tricky. Because each element is likely to represent some real-world object, such as an aircraft, it is important either to provide a mechanism for resolving the conflict caused by multiple asynchronous requests to create an element that represents the same thing or to provide a mechanism for managing the creation of Nodes. Poligon opts for the latter approach.

Slot Level   The programmer should cause each Node to have an idea of how to improve its own idea of the solution

- to have *Goals.* In Poligon this is done at a fine grain, with each field of each element in the solution being able to have associated with it functions which enable it to evaluate itself.

It was found that a good axiom for programming these systems is "Never throw away any data unless you are convinced that you have better data." This is the sort of behavior that is used in the evaluation functions mentioned above.

Rule Execution   Poligon attempts to maintain the smallest critical sections possible. The original implementation of Poligon in fact had as its only atomic actions reading a field and writing a field. It was soon found that, in order to maintain consistency during rule execution, it had to be possible to read the values from a number of fields simultaneously - taking a snapshot without the subject moving. This, coupled with critical sections for the writing of collections of values, allows confidence that the picture that one sees when taking such a snapshot of a Node is consistent, even if not necessarily the most up to date. It is important for a Poligon programmer to be aware that the Node of which a snapshot has been taken may well be read from and written to by other Rules asynchronously during the invocation of the Rule taking the snapshot.

5 CAGE Problem Solving Framework

CAGE [1, 8] is a framework for building and executing applications as a concurrent blackboard system. CAGE is based on the AGE [7] serial blackboard framework. It includes mechanisms for the concurrent execution of knowledge sources, rules and parts of rules. The CAGE user has complete control over which of these mechanisms are used. CAGE is designed to execute on a shared-memory, multiprocessor system with tens to hundreds of processors. It is implemented using Qlisp, a concurrent dialect of Lisp designed for multiprocessors with a single, shared address space. CAGE currently executes on a shared-memory variant of CARE [4] simulated using the SIMPLE simulation system.

5.1 CAGE Design

CAGE is a blackboard framework system. In addition to the basic functionality found in AGE, CAGE allows user-directed control over the concurrent execution of many of its constructs. Otherwise, the two systems are functionally identical. The basic components of a system built with CAGE are:

- A global data store (the blackboard) on which emerging solutions are posted. The elements on the blackboard are organized into levels and represented as a set of attribute-value pairs.

- Globally accessible lists on which control information is posted (e.g. lists of events, expectations, etc.).

- An indefinite number of knowledge sources, each consisting of an indefinite number of condition-action rules.

- Various kinds of control information that determine (a) which blackboard element is to be the focus of attention and (b) which knowledge source is to be used at any given point in the problem solving process.

- Declarations that specify the components (knowledge sources, rules, condition and action parts of rules) to be executed in parallel, and when to force synchronization.

Using the concurrency control specifications, the user can alter the simple, serial control loop of CAGE by introducing concurrent actions. CAGE allows parallelism ranging from concurrently executing knowledge sources all the way down to concurrent actions on the condition and action sides of the rules.

5.2 Building applications in CAGE

The CAGE System provides a CAGE language with which the user can write an application. The type of user-supplied information is similar to that required for applications constructed in the AGE system, however, the structure of the information is somewhat different.

5.2.1 Blackboard Data Structure

There are two major components in the CAGE blackboard structure, the hypothesis *classes* (frequently called levels in hierarchical blackboard structures) and the hypothesis *nodes*. The user must specify the classes that make up his application's blackboard structure. For each class, the user must define the fields to be associated with the nodes created in that class. Nodes are created in those classes, either a priori by the user or dynamically while executing the user's rules. Each of the classes is defined as an object with the attributes as instance variables and with the nodes as instances of the class objects.

5.2.2 Control Structure

All CAGE control information is referenced through the Control-Structure object which is basically the same as in AGE.

5.2.3 Knowledge Sources

CAGE knowledge sources are partitions of the application knowleage. Each knowledge source consists of some declarative information and a set of rules.

Knowledge Source Declarations A knowledge source consists of more than just groups of rules. In order to interpret the rules properly, CAGE needs answers to some questions about knowledge source control, for example,

- Under what circumstances should this knowledge source be invoked?

- Which one, of all of the rules whose condition part is satisfied, should be executed?

- Are there any local variables to be defined for this knowledge source?

The following are the primary knowledge source control options available for the user to use in order to tailor a knowledge source:

> **Preconditions:** A list of tokens, representing the *event names* used in rules. If the currently focused event has an event name that matches one of the knowledge source's preconditions, then that knowledge source is activated.

> **Hit Strategy:** There are two main hit strategies available in CAGE, Single and Multiple. When a knowledge source with a single-hit strategy is invoked, the rules of that knowledge source are evaluated, in order, until one rule's condition is satisfied. Then, the actions of the action part of the rule are executed, and no further rule is evaluated. With a multiple-hit strategy, the condition parts of all the rules are evaluated, and all the action parts of the rules whose conditions were true are executed.

16

Definitions: A list of local variables. The definitions are an efficiency feature to avoid the repeated calculation of the same variable. The structure is similar to that of LET, pairs of a variable names and expressions.

Rule Order: A list of rule names, representing the rules of the knowledge source. This is the order in which the rules are to be evaluated when in serial mode.

5.2.4 Rules

CAGE rules consist of three major parts: definitions, conditions, and actions.

Definitions: The definition part of a rule is similar to a LET in structure. The scope of the variables defined here is the rule, both in the condition and action parts, as well as other definitions in the rule.

Condition part: The condition part consists of one or more conditional clauses. The clauses can be an arbitrary expression. The condition part can reference both the variables local to the rule or to the knowledge source. The CAGE system provides several access functions for retrieving values from the blackboard nodes which can be used in the condition part.

Action part: The action clauses make up the final part of a CAGE rule. The actions specify the changes to be made to the blackboard and how those changes are to be made. The user must specify what node and attributes on the blackboard are to be changed, what the new links or values are, and how those changes are to be made (possibly deleting some old values). The user must also specify an event name representing the type of change this action makes to the blackboard. If and when the event created by this action is selected as a focus event, this token will be matched against the preconditions of the knowledge sources to determine which knowledge source to invoke next.

5.3 Specifying Concurrency

CAGE supports the concurrent evaluation of various pieces of knowledge. The use of knowledge sources to partition the knowledge in blackboard systems and, in particular, the structure of the knowledge sources in CAGE provide several obvious places for concurrency. The knowledge sources group the domain knowledge into independent modules, which, theoretically, could be invoked independently and concurrently. Within each knowledge source the rules provide another source of parallelism, and within each rule, the clauses of the condition part and the different actions within the action part provide others. Of course, not all the clauses, rules or even knowledge sources are actually implemented totally independently of each other and some serialization may be necessary to solve the application problem correctly.

The following are the options for parallelism available in CAGE, grouped according to their allowed use in combination.

17

Clause level: can be used in combination with each other or any other parallel option.

>**actions**: Execute the action clauses of a rule in parallel. Note: When running the actions concurrently non-determinism may result if both destructive (Supersede in CAGE) and constructive (Modify) actions occur to the same object-attribute.

>**conditions**: Evaluate the condition clauses of a rule in parallel. Note: Use the rule definitions to set any local variables tested here, insuring that the lhs clauses will not be contending for the same data element.

>**rule-definitions**: Evaluate the definitions of a rule in parallel. Again, these definitions should be independent of each other AND should avoid accessing the same data, if their concurrent evaluation is to result in an actual speed-up.

Rule level: Definitions can be used in combination with any of the other options, but only one of the rule options, single, multiple, sync or nosync can be used at a time.

>**definitions**: Evaluate the definitions concurrently at the beginning of a knowledge source.

>**rules-single**: Evaluate all the condition parts of the rules of a knowledge source concurrently, but only execute the actions of one successfully evaluated rule.

>**rules-multiple**: Evaluate all of the conditions of the rules of a knowledge source concurrently, wait until all the evaluation is completed, then execute the actions of all the successfully evaluated rules serially.

>**rules-sync**: Evaluate all the condition parts of the rules of a knowledge source concurrently, wait until all the evaluation is completed, then execute the actions of all applicable rules concurrently.

>**rules-nosync**: Evaluate the condition parts of the rules of a knowledge source in parallel and execute the action part of each rules as soon as the conditions evaluate to true. Executed the actions within the action part in parallel. With this option there is no synchronization between the rules in the knowledge source.

Knowledge source level: Only one of the concurrency options for the knowledge source can be set at any one time.

>**kss**: Activate all the applicable knowledge sources at once. Synchronization is accomplished by waiting for all knowledge sources to complete execution (and the event list is updated) before invoking a new set of knowledge sources concurrently.

>**kss-nosync**: Invoke all applicable knowledge sources as soon as a new event is created. This option provides the least control of all the options available and does no

synchronization. Many applications will have to be significantly changed to execute correctly under these conditions, particularly removing any possible circular knowledge source invocations. Without any synchronization, as soon as an event is created all relevant knowledge sources become active -- no events are added to the eventlist and no focus event is ever selected.

kss-minisync: Add an event to the event list and do minimal computation at the point of synchronization before invoking the next set of knowledge sources. The main computation done is the collection and pruning of similar events, leaving fewer events to activate subsequent knowledge sources.

5.4 CAGE Machine Model

Because CARE is a message passing, distributed memory model, we had to create a shared memory variant of CARE to simulate CAGE execution. Currently we simulate an even number of processors, using half as processor-cache pairs and half as controller-memory pairs. The atomic unit of memory access in CAGE is a blackboard node. Concurrent node access requests are handled by simple spin lock mechanisms.

With CAGE-CARE every step of the simulation, down to a very low level, is measured. For example, one can track the length of the memory queues to get a handle on a major issue in programming concurrent blackboard systems, memory contention. Other measurable factors include the overhead for creating new processes, network communication costs and the cost of creating a new node. Using CAGE-CARE one can experiment with multiprocessors of various sizes and can get a reasonably accurate picture of the parallelism obtainable for a particular application. The only disadvantage for the user is the length of real time it takes to run a simulation on CAGE-CARE. and combinations later.

6 CAGE, Poligon and LAMINA Comparative Experiments

During the past contract period we have been developing application software and machine architecture models to support a series of end-to-end experiments comparing various concurrent programming systems for knowledge-based applications. The goals of these experiments are to:

1. Obtain quantitative comparisons of the performance of the programming systems.

2. Gain insights into how different concurrent programming models lead to different (or similar) application decomposition and organization.

3. Force the refinement of the concurrent programming systems so as to better support application development.

4. Gain insights into the ease or difficulty of writing application code in each of the programming systems.

6.1 The Experiments

The common application for these experiments is Elint [2], a real-time, knowledge-based system for integrating pre-processed, passively acquired radar emissions from aircraft. This Elint application has been implemented in three different concurrent programming systems:

- The concurrent object-oriented programming model supported by LAMINA [3]. LAMINA is the basic, low-level programming interface to CARE, a grid-based, distributed address space, message passing multiprocessor architecture [4].

- The Poligon system [9, 8]. Poligon is a demon-driven system derived from the blackboard model of problem solving.

- The CAGE system [1, 8]. CAGE is a concurrent descendant of the AGE serial blackboard framework.

Each of the implemented applications will be executed and evaluated using various input data sets and varying numbers of processors.

Application code written in either LAMINA or Poligon compiles to code which executes on the CARE architecture. CAGE, however, is targeted toward a single address space, shared variable multiprocessor architecture. CAGE is implemented in QLisp, a concurrent Lisp for shared variable multiprocessors. To support CAGE we had to develop a multiprocessor "blackboard machine" variant of CARE. This blackboard machine models a shared variable architecture and includes the mechanisms and instruments necessary to manage and study memory contention. The architecture implements the blackboard and the control data structures in global, shared memory. It directly supports the CAGE system and application code written in QLisp.

6.2 Experiment Status

During the past contract period we have:

1. Completed the implementation of the the Elint application in each of the three concurrent programming systems.

2. Completed the development of the blackboard machine variant of CARE.

3. Developed an experiment plan for the comparative studies.

4. Developed a new measure of speedup as a function of the number of processors in a multiprocessor system. This measure is useful for evaluating system performance of real time applications and is based on the concept of maximum sustainable input data rate.

5. Completed the first set of experiments for each of the three programming systems.

7 The AIRTRAC Application

AIRTRAC [5] is the primary application driving our development of concurrent knowledge-based system programming methodologies. Also, it is one of the basic applications used for our multiprocessor architecture performance experiments. AIRTRAC is a knowledge-based signal interpretation and information fusion system. The system attempts to identify, track, and predict the future behavior of aircraft. In particular, it attempts to recognize aircraft which might be engaged in covert activity, for example, smuggling. The inputs to AIRTRAC are periodic radar tracking system reports, a priori, filed flight plans for some aircraft, and occasional intelligence reports about suspected covert activity.

AIRTRAC is designed to be sufficiently complex and realistic to adequately test various ideas about concurrent problem solving on multiprocessor machine architectures. The AIRTRAC application involves continuous input data streams, typical of real-time signal interpretation problems. Such problems often require a level of computational power two to three orders of magnitude beyond what is currently available. Moreover, the application uses data-driven, expectation-driven and model-driven styles of reasoning. These reasoning styles encompass a wide range of paradigms in artificial intelligence.

21

7.1 Overall Application System Structure

The overall system consists of radar collection sites and associated trackers, filed flight plan sources, intelligence report sources, and the AIRTRAC system running on a multiprocessor.

Output from each radar is fed to an associated tracker which produces periodic track reports for input to AIRTRAC. A tracker detects aircraft, estimates their positions and velocities, and assigns unique track identifiers. A tracker continues to assign the same identifier if it believes that the received signal is due to the same aircraft which was previously seen. Periodic reports from the tracker include the scantime, track identifier, and the mean and covariance of the position and velocity of the track. Because of tracker limitations, they usually lose a track when the corresponding aircraft makes a significant maneuver such as turning sharply. A tracker assigns different identifiers to the tracks before and after such a maneuver. One of the tasks of AIRTRAC is to connect such "broken" tracks. Another AIRTRAC task is to fuse multiple tracks which represent the same aircraft observed from different radar sites.

A filed flight plan is information regarding the expected position at given times of the flight path of an aircraft. Since filed flight plans are only estimates of actual flight paths, their track information is less precise then actual observed track data. Filed flight plans are usually available for cooperative aircraft. Intelligence reports provide information about possible origins, possible destinations, and possible flight times for aircraft engaged in covert activity. This information typically embodies a "tip-off" about covert activity. Due to the sketchy nature of the information, intelligence reports are even less precise than filed flight plans. AIRTRAC attempts to fuse observed tracks, filed flight plans, and intelligence reports which represent the flight path of the same aircraft.

7.2 AIRTRAC Organization

The AIRTRAC system is partitioned into three major modules. At the lowest level of data abstraction, the Data Association Module accepts as input the periodic output of the radar trackers. The primary task of the module is to abstract the periodic track reports into sequences of straight-line Radar Track Segments that represent (approximately) constant-heading, constant-velocity segments of an aircraft's flight path. Other tasks of this module are to recognize when a track with a new identifier is initiated, determine when sufficient evidence has been collected for a track to confirm its existence with a given probability, and to recognize when a track with a given identifier has been terminated.

The Path Association Module receives the Radar Track Segments from the Data Association Module. It attempts to "connect" the segments into coherent tracks representing the flight paths of the aircraft under observation. It then attempts to fuse the tracks which correspond to the same aircraft observed from different radar sites. The module also accepts as input filed flight plans and intelligence reports, and it attempts to fuse the plans and reports with the observed tracks. The module uses models of aircraft performance characteristics such as velocity, acceleration and maneuverability to help form hypothesized flight paths. The Path Association Module must deal with ambiguous data, and it maintains, if necessary, alternative flight paths for an observed aircraft. For each alternative, hypothesized flight path, the module maintains a measure of confidence in the hypothesis which rises as more evidence is accumulated fitting the hypothesis and which falls if expected behavior consistent with the hypothesis does not materialize.

The primary tasks of the Path Interpretation Module are to predict the future behavior of observed aircraft and to identify aircraft which are engaged or might engage in covert activity. The module takes into account the current and predicted flight paths of the observed aircraft, information about existing airports, known radar shadow regions, known flight corridors, and geographic and/or political boundaries. It uses models of aircraft behavior that embody strategies and goals to help form reasonable hypotheses.

7.3 AIRTRAC Status

The AIRTRAC Data Association Module and associated experiments were completed during Phase One [6]. The experiments were performed using the SIMPLE/CARE multiprocessor simulation system. They demonstrated that almost linear speedup as a function of the number of processors can be achieved (at least up to 100 processors) for a periodic data-driven knowledge-based system such as the Data Association Module.

8 Multiprocessor Load Balancing Studies

One of the more difficult problems in actually realizing high levels of concurrent execution of applications on multiprocessor systems is that of processor and/or memory load balancing. Based on our experiments with concurrent knowledge-based systems, the single largest impediment to achieving high utilization of multiprocessing resources is localized processor and/or memory "hot spots." That is, processors or memory acess queues which are overloaded relative to the rest of the system. Such hot spots result in many of the processors sitting idle awaiting

information from the overloaded resources. This load balancing problem is particularly acute for concurrent applications such as signal interpretation where there is significant dynamic (i.e., run-time) creation and destruction of processes and data structures. This situation is in contrast to well-structured applications such as finite element computations where all processes and data structures are known at load-time.

8.1 Load Balancing Studies Status

Our work to date on load balancing has focused on non-adaptive schemes. That is, schemes in which once a process is allocated to a processing site it remains there throughout its life. In adaptive schemes active processes can migrate between processing sites.

For our earliest ELINT-CAOS experiments [2], we used an extremely simple load distribution scheme based on round-robin assignment of dynamically created objects to processing sites. This scheme resulted in poor resource utilization, for example, at best 25% average processor utilization for a 49 processing site CARE architecture.

We next experimented with various dynamic load distribution schemes employing techniques such as each site keeping track of its (logically) immediate neighbor's loads and using application domain knowledge to predict the lifetime and busyness of dynamically created objects. These schemes resulted in, at best, very marginal improvement over the round-robin scheme.

We then experimented with non-adaptive schemes based on random scattering of dynamically created objects to processing sites. Surprisingly, this scheme performed remarkably well relative to the earlier, more information intensive schemes. We are currently using a variant of the random scattering scheme in which each processing site is assigned an a' priori preference weight with respect to accepting dynamically created objects. These weights are based on the distribution of load-time created objects onto sites. The random distribution of dynamically created objects to sites is skewed so as to respect this weighting.

Although this weighted random distribution scheme provides the most balanced loads that we have achieved to date, it still results in significant underutilization of machine resources. For example, we have achieved, at best, only about 50% average processor utilization on 64 site CARE architectures.

24

References

[1] Nelleke Aiello.
User-Directed Control of Parallelism; The CAGE System.
Technical Report KSL 86-31, Stanford University, April, 1986.
Reproduced as Appendix C in this report.

[2] Harold Brown, Eric Schoen, and Bruce Delagi.
An Experiment in Knowledge-Base Signal Understanding Using Parallel
Architectures.
Technical Report KSL 86-69, Stanford University, October, 1986.
Reproduced as Appendix B in this report.

[3] Bruce A. Delagi, Nakul P. Saraiya, and Gregory T. Byrd.
LAMINA: CARE APPLICATIONS INTERFACE.
Technical Report KSL 86-67, Stanford University, November, 1987.
Reproduced at Appendix F in this report.

[4] Bruce A. Delagi, Nakul Saraiya, Sayuri Nishimura, and Greg Byrd.
An Instrumented Architectural Simulation System.
Technical Report STAN-CS-87-1148, Stanford University, January, 1987.
Reproduced as Appendix E in this report.

[5] J. R. Delaney.
Multi-System Report Integration Using Blackboards.
In *Proceedings of 1986 American Control Conference.* March, 1986.
Reproduced as Appendix D in this report.

[6] Russell Nakano and Masafumi Minami.
Experiments with a Knowledge-Based System on a Multiprocessor.
Technical Report KSL 87-61, Stanford University, October, 1987.
Reproduced as Appendix G in this report.

[7] H. Penny Nii.
An Introduction to Knowledge Engineering, Blackboard Model and AGE.
Technical Report KSL 80-29, Stanford University, December, 1980.

[8] H. P. Nii, N. Aiello, J. Rice.
Frameworks for Concurrent Problem Solving: A Report on Cage and Poligon.
Technical Report KSL 88-02, Stanford University, February, 1988.
Reproduced as Appendix I in this report.

[9] J. P. Rice.
Poligon, A System for Parallel Problem Solving.
Technical Report KSL 86-19, Stanford University, April, 1986.
Reproduced as Appendix A in this report.

[10] J. P. Rice.
Problems with Problem-Solving in Parallel: The Poligon System.
Technical Report KSL 88-04, Stanford University, January, 1988.
Reproduced as Appendix H in this report.

Poligon, A System for Parallel Problem Solving

by
James P. Rice

Knowledge Systems Laboratory
Computer Science Department
Stanford University
Stanford, California 94305

Table of Contents

Summary

The Poligon[1] system is a new, domain-independent language and attendant support environment, which has been designed specifically for the implementation of applications using a Blackboard-like problem-solving framework in a parallel computational environment.

This paper describes the Poligon system and the Poligon language, its salient and novel features. Poligon is compared with other approaches to the programming of parallel systems.

1. Introduction

The larger project of which Poligon is only a small part will not be discussed here in any detail. Design decisions made in other parts of the project will be held to be axiomatic, though some mention of these decisions will be made in order to show the motivation for the features of Poligon. The primary objective of the overall project is to achieve significant speedup of knowledge based systems, particularly those directed at real-time signal understanding.

The purpose of the Poligon language is to express the problem solving behaviour of human experts in order to map them onto a problem solving framework, which will run on simulated parallel hardware.

The fields of knowledge representation and problem solving are rich and complex. This paper will not go into any great detail in describing the problem solving processes involved. Poligon tries usefully to express knowledge both in a declarative and procedural sense, through rules [Davis 77]; and in a structural sense, through the configuration of the solution space. These will be described below.

Some crucial design criteria and early design commitments have affected the development of Poligon, the consequences of which will be described in this paper. These can be summarised as follows.

- Poligon is intended to be a language for both problem solving and the general purpose programming necessary to support it. Unlike most programs, Poligon programs must also address the problems of real-time processing, including asynchronous events and input data backup. Poligon, therefore, must assist in this respect.

- The overall project's strategy is to solve problems significantly faster than existing systems through the exploitation of parallelism. Poligon is targeted at a MIMD, distributed-memory, message-passing machine with ~thousands of processors. This hardware gives direct support for futures, remote objects and such efficient message-passing strategies as *Broadcast* and *Multicast* so as to take full advantage of its processor interconnection network.

- A consequence of the desire to achieve a significant order of parallelism in Poligon programs is that many of the control mechanisms used in serial problem solving systems, such as schedulers and event queues, have been discarded because they are highly serial. Most actions in Poligon programs are, therefore, performed asynchronously. Rules, the primary mechanism in Poligon for describing things and for getting things done, are activated as daemons. Much of the work in Poligon is aimed at providing mechanisms to cope with this chaotic behaviour.

This paper contains the following;

- A discussion of related work in parallel languages.

- A discussion of the design approach guiding the development of Poligon.

- A description of the abstraction mechanisms provided by the Poligon system with some small examples.

- Some concluding remarks.

- References for further reading on the subject.

1.1. Knowledge Representation and Problem Solving in Poligon

The primary purpose of this paper is to discuss the Poligon language. It is, however, not possible completely to divorce this from the underlying hardware and from its purpose; knowledge representation and problem solving.

Poligon can be described loosely as a "Blackboard System". What this means in practice is that the problem solving metaphor of Poligon is one of cooperating experts gathered around a blackboard, posting ideas about their deductions on the blackboard. For an exposition on the term "Blackboard System" the reader is encouraged to read [Nii 86]. Poligon tries usefully to express knowledge both in a declarative and procedural sense, through rules and functions; and in a structural sense, through the configuration of the solution space on the blackboard. In particular, the term "blackboard" will be used to describe the set of all of the nodes in the solution space of the system.

The suggestion that Poligon is a blackboard system is a little controversial. There are a number of respects in which this is not a satisfactory label. This term will, however, be used freely from now on for lack of a better label. The reader is encouraged to substitute for the term "Blackboard system" any term, such as "Frame System" which seems best to fit his mental model of what is being described.

1.2. Poligon's Model of Parallelism

It seems appropriate here to describe Poligon's model of parallelism. In its simplest form this can be thought of as *An Element in the Solution Space as a Processor*.

This gives some idea of the granularity that is being sought. It is, however, by no means the most efficient way to implement Poligon. Poligon programs want to be able to execute rules and parts of rules associated with a particular *Node* in the solution space in parallel. These rule activations need processors, on which to execute.

Thus a modified version of Poligon's model of parallelism could be *A Rule Activation as a Process, with sufficient processors to cope with the parallelism exhibited by the rule during its activation.* This tends towards a mapping of solution space elements onto a cluster of processors to service the rule activations. In practice, however, a number of nodes might be folded over the same set of processors, either because nodes become quiescent or because the load balancing in the system is sub-optimal.

2. Related Work

Work in this field falls into two distinct categories; work on parallel knowledge based systems and work on languages for parallel symbolic computation. The former is, at present, a very sparse field and, will not be discussed here, though some references are given in § 6. The latter is much more highly developed.

Much work is already being done on parallel languages for general computation. Amongst these languages are Actors, MultiLisp and QLisp on the one hand and concurrent logic programming languages and purely functional languages on the other. Often missing from this

work is a thrust toward the investigation of large applications in parallel domains, for instance the development of parallel knowledge representation and problem solving systems. This is, of course, what Poligon attempts to do. This section will discuss briefly Actors, QLisp and Multilisp, since these are the parallel symbolic computation languages which are most relevant to the development of Poligon and the software which lies beneath it.

2.1. Actors

Actors [Hewitt 73] probably come the closest in their behaviour to Poligon, at least at an implementation level. Actors are independent, asynchronously communicating objects. As is the way with purely object oriented systems they communicate only through message passing and have tightly defined operations. The mutual control of Actors an parallelism is achieved by the support of procedure call and coroutine model message passing. The modularity afforded by this sort of programming metaphor may well be especially useful for the programming of distributed-memory, message-passing hardware, since having a close match between the hardware and software metaphors is likely to achieve better performance. It is not in any way surprising that the operating system level software, which underlies Poligon, is founded on many of the same principles as Actors. It has yet to be seen whether this programming methodology is able in practice to extract significant amount of parallelism from problems, though clearly this project hopes that it is.

2.2. MultiLisp and QLisp

MultiLisp [Halstead 84] and QLisp [Gabriel 84] are lumped together because, at least in some senses, they have strong generic resemblances. They are both, at the user level, extensions to existing Lisp dialects which provide mechanisms for the expression of parallelism, such as parallel Let constructs and parallel function argument evaluation (QLet and PCall). It is assumed by both of these systems that the hardware at which they are targeted is a form of shared-memory multiprocessor. Although there is no particular reason why such systems could not be implemented on a distributed-memory system, they are optimised for shared-memory multiprocessors. These are currently the most readily available form of multiprocessor. They would, however, need significant extensions in order to be able to exploit a distributed-memory system as is shown in CAREL [Davies 86], an implementation of QLisp for distributed-memory machines. The assumption of shared-memory, MIMD processors in these systems imposes constraints on the languages. They assume, at least to an extent, that processes will be expensive and that the user must have control over their creation. Poligon assumes quite the opposite.

3. The Design of Poligon

Poligon will be discussed first in terms of the way in which the language relates to the problems being solved and its underlying systems. Next the language will be discussed in terms of the requirements for languages in general and parallel languages in particular.

3.1. Background and Motivation

The philosophy behind the design of Poligon comes from intellectual and pragmatic pressures. It attempts to steer a middle course between the extreme purism of applicativists and the extreme pragmatism of the proponents of side-effects.

From the outset, the project was oriented towards real-time problem solving. Blackboard systems are well known to be of interest as tools in the knowledge engineer's toolkit. Little work has been done to investigate the appropriateness of the blackboard metaphor to parallel execution or the meaning of parallel blackboard systems, though it is frequently claimed that they are full of latent parallelism. The excellent formal properties of pure applicative and logic languages may well be of little use in a system which, for whatever reasons, needs to express side-effects and which has to cope with real-time constraints. Poligon is a system in which

some of the formal rigour of truly applicative systems has been put aside in favour of a pragmatic approach to the exploitation of parallelism.

The BB1 project [Hayes-Roth 85], also a project at the HPP, is an attempt to investigate the behaviour of highly controlled problem solving systems. It attempts to use a great deal of meta-knowledge and makes significant use of globality of reference in order to support an holistic view of its solution space, thus providing a basis for meta-level reasoning. The Poligon project is an attempt to investigate quite the reverse. Poligon has very little support for meta-knowledge and allows no global data or global view of the solution space whatsoever. The purpose of this experiment is to determine whether a system, unconstrained by a great deal of serialising control knowledge, might still be able to find useful answers faster than an highly controlled system, such as BB1, which would be extremely difficult to speed up significantly through parallelism.

The Poligon system pictures the elements in its solution space as processes resident on processors distributed across a grid, with the code necessary for them intimately associated with them. Because no global control is permitted in Poligon the activation of rules is necessarily completely daemon-driven.

The project hopes to achieve significant speed-up through parallelism. This can be done only if much parallelism is extracted from the problem. Ideally, the system would try to achieve its parallelism by exploiting parallelism in the program's implementation at a very fine grain. This can, in principle, extract the maximum amount of parallelism available. On its own it has drawbacks, however. The costs of processes and the problems of synchronisation at a fine grain size make it difficult to exploit such parallelism without the use of hardware mechanisms significantly different from those available with prevailing technologies. This approach is also only part of the story. It neglects the fact that a properly parallel decomposition of the source problem is crucial to finding a lot of parallelism. One could summarise the problems, therefore, as expressing the problem in a sufficiently parallel fashion and the matching of the parallelism in the program to the grain size of the underlying hardware. Poligon addresses these issues.

Parallelism is very hard to find in conventional programs. Applicative systems have an advantage in this respect because of their relative lack of need to express parallelism explicitly. Their unchanging semantics when parallelism is introduced eases matters considerably. Poligon has attempted to learn from this and has pure applicative semantics in a number of areas but takes a different approach to the finding of parallelism in programs. It attempts to execute everything in parallel that it can and leaves it to the programmer to find any serial dependencies.

When the parallelism in a program is user-defined, problems can result from an inappropriate match between the granularity of the parallelism expressed in the program and the granularity of the underlying machine. In systems of the size and complexity of a typical Poligon application such a match would be particularly difficult to find because of the large number of processors involved and because it would be difficult for the user to keep track of the location of his data in the processor array. These characteristics are a consequence of the highly variable and data dependent state of the solution space in such programs. Poligon, because of its structure, should be able largely to obviate such granularity mismatches because parallelism is defined and controlled by the system and the Poligon system is closely matched to the granularity of the underlying system.

It is often thought that problems suitable for solution by means of the blackboard model tend to partition their solution spaces into what look rather like pipe-lines. Pipe-lines are, of course a well known form of parallelism. In practice pipes in such systems are not pipes in the normal sense, since they are more like "leaky" pipes. It is one of the prime objectives of these systems to reduce the amount of data as it percolates up through the abstraction hierarchy of the solution space. Because of the reduction in the data rate flowing in these pipes the contention problems that one might expect when pipes are connected into trees, as they often are, are alleviated.

A significant limitation of the performance of pipelines is that, at best, the parallelism that they can produce is proportional to the length of the pipe. This would typically be only of the order of half a dozen sections. This is clearly not the "orders of magnitude" of performance improvement that we all hope for. In practice, though, given a large enough problem, it is often possible to set up a large number of these pipes side-by-side. It is one of the major objectives of the Poligon language to encourage, facilitate and reward the decomposition of problems so that this form of independence can be exploited, so that such pipes will be created by the system.

3.2. Language Requirements

Poligon is a language which is by no means directed at general computation. It is nevertheless intended to be used for the solution of large, complex problems on distributed-memory parallel hardware. The following is a brief list of the ways in which Poligon attempts to address some of the primary requirements of programming languages.

- The language should provide a tangible method of expressing the ideas of the programmer.

  The Poligon language has been written with considerable input from those with experience in problem solving systems in the application domains at which it is targeted. It is therefore intended to match the ideas of the "Expert", whose knowledge is to be encoded, but in a domain independent way.

- The compiler[2] should provide a mapping between the language and the underlying systems, be they hardware or software.

  Poligon's compiler compiles Poligon language source into code understood by the underlying *Lisp* system and the concurrent object-oriented operating system running on its target hardware.

- The language should abstract the programmer from its underlying systems.

  The Poligon system shields the user from all aspects of the underlying hardware such as the topology of the processor network, the message-passing behaviour of the hardware and the location of any code or data within the network.

- The language should provide mechanisms for the exploitation of the underlying systems to good effect.

  The underlying hardware and software systems are exploited in a number of ways in Poligon. Firstly the language encourages the user naturally to decompose his problem into a form which will map efficiently onto the underlying hardware. Secondly the language offers a number of application-independent, high-level constructs, which are designed to exploit the hardware to the full. These topics are covered more fully in § 4.

- The language should allow the development of software faster than would be the case if it were to be developed in a less abstract form.

  Considerable effort has been spent on making the Poligon language a high level way to describe the solutions to parallel knowledge based system problems. A high level language with such features as infix, user-definable operators and user definable syntax, provides a natural way for the expert to implement his knowledge.

  Much effort has been spent also on integrating the Poligon system cleanly into the program support environment of the Lisp Machines on which it runs. For instance, incremental compilation is supported from within the editor.

---

[2]The term *Compiler* is used in its most general sense here, perhaps an interpreter or a machine which is clever enough to execute the language specified directly.

- The language should assist the development of reliable, maintainable and modular software.

  Language features are provided to minimise the possibility of inconsistent modifications to the source code and the structure of the language and its semantics are defined in a manner which minimises the probability of complex bugs being introduced by asynchronous side-effects.

  A sophisticated set of debugging facilities is provided. A system that emulates the semantics of full, parallel Poligon programs as closely as possible in a serial environment has been produced. The user is able to debug his program serially to remove all possible serial bugs and bugs due to the non-deterministic execution order of Poligon programs before it is ported to the full parallel environment.

In addition to these requirements a language targeted at parallel hardware should have a number of attributes which reflect the parallel nature of the target hardware.

- The language should address the granularity of the hardware.

  Poligon is closely matched to the granularity of the hardware at which it is targeted. It is generally expected that the solution space of the problems addressed by Poligon programs will have of the order of thousands of nodes. This is of the same order as the granularity of the hardware.

- The language should provide a mechanism for the extraction of parallelism from programs and from the programmer.

  Poligon extracts parallelism from programs and the programmer in two main ways. First the decomposition of the problem is encouraged to be as modular as possible. Secondly the semantics of Poligon programs are such that almost all of the program can be executed in parallel without changing their behaviour from that seen during serial execution. This allows the system to execute most operations in parallel if it has the resources to do so.

- The language should, where appropriate, shield the programmer from those details of the hardware which are particular to parallel computing engines, such as topology.

  The hardware, on which Poligon programs runs, causes Poligon programs to have to cope with communication between solution space elements on different processor sites. All such message passing is hidden from the user. In fact the Poligon language has no concept of message-passing at all.

  Futures are used for all remote operations in the user's program. The hardware implements these such that there is no efficiency penalty associated with creating futures for such remote accesses. The Poligon language copes with these invisibly to the programmer.

As can be seen quite easily from the above one of the factors that must be well understood before a language is designed is the general purpose of the language and the level of generality that is expected of programs written in it. A language, whose sole purpose is the expression of solutions to huge matrix problems on systolic hardware might well be justified in expecting the programmer to express, at quite a low level, the mapping of the program onto the hardware provided. This is less likely to be a reasonable expectation of a language targeted at the solution of large, complex problems of an unpredicatable, dynamically-varying or data-dependent nature. Poligon is a fairly general purpose programming language with a very definite bias.

4. Abstractions in Poligon

To cope with Poligon's view of parallelism and with the chaotic execution of rules (see § 1) a number of linguistic abstractions are provided.

Poligon provides abstractions for knowledge representation, control, data, parallelising, real-time and side-effect control. These will be described briefly in this section.

4.1. Knowledge Representation

Knowledge is traditionally represented in blackboard systems in a number of ways, listed below.

- *Declarative Knowledge* is encoded in *Rules*.

- *Procedural Knowledge* is encoded in procedures.

- Knowledge concerning the sequencing of activities is encoded in the scheduling mechanism.

- Knowledge about the structure of the solution space is encoded by the definition of the structure of the blackboard.

- Knowledge about relationships between the objects in the system is often encoded using a Link mechanism.

These all represent knowledge about the application domain. In addition, there is in any program a large body of implicit knowledge concerning the semantics of assignment, sequencing and the system's function as a whole, especially in for systems with poor formal properties. This will not be discussed here. The Poligon language does, however, go to considerable effort to make the semantics of the Poligon system as clear as possible.

4.1.1. Declarative Knowledge

The encoding of *Declarative Knowledge* in blackboard systems is conventionally done in *Rules*[3], which exist within scheduling units known as *Knowledge Sources*. Poligon also has the concept of Rules and Knowledge Sources, though their meaning is somewhat different. Unlike serial blackboard systems, the rules in a Poligon system are activated autonomously and asynchronously.

Existing blackboard systems usually suffer from a confusion and overloading in the semantics and purpose of knowledge sources. It is useful to collect one's knowledge of one subject together into one chunk. These chunks are knowledge sources. Sadly, the implementors of blackboard system frameworks often think of knowledge sources as scheduling units and thus design their scheduling strategies around the idea of the "invocation of knowledge sources", even though it is by no means necessarily the case that it is appropriate to schedule all of knowledge in a chunk at the same time. This has a detrimental effect on the modularity of the system.

In Poligon, knowledge sources are used as linguistic and software engineering abstractions provided for the programmer in order to allow him to collect related knowledge together. There are no scheduling semantics associated with knowledge sources in Poligon. Because of the underlying system's daemon-like rule triggering mechanism the rule writer is allowed completely to decouple the concept of *scheduling* from the concept of *chunks of knowledge*.

Rules are activated as a result of "events" happening to the fields of nodes (see § 4.3.1). These events can be caused either by a write operation to a field, by a semaphore being waved at a field or by the real-time clock.

A powerful *Expectation* mechanism is provided, which allows the dynamic placement and specialisation of rules. An Expectation is a way of expressing model-based knowledge. Given

---

[3]The term *Rule* is used here in the sense of "Pattern/Action pairs". It should be noted that these are quite unlike the structures called rules used, for instance, in Prolog. Pattern/Action rules move towards a solution to their problem by performing side-effects on their environment, in this case the blackboard, not through unification.

a particular model of the behaviour of a system, certain changes might be expected if the model's interpretation of the world is correct. Expectations allow such changes to be watched and even allow their associated rules to be triggered if the changes do not happen in a given time. Such expectations can be placed to watch for events happening, or not happening, in specific places on the blackboard, at specific times. Expectations provide a focussing mechanism[4] and, coupled with the system's ability to trigger[5] rules and "time-out" unsatisfied Expectations on the basis of the real-time clock, Poligon allows complex time-critical knowledge to be expressed and applied simply.

An example rule is shown in figure 4-1.

4.1.2. Procedural Knowledge

Procedural Knowledge is an all encompassing term usually used indiscriminately to describe both knowledge about the relationships between values (*Functions*) and the mechanisms for performing side-effects and for sequencing events (*Procedures*). This is often a result of such systems being built on top of Lisp systems, which fail to draw distinctions between procedures with side-effects and those without. Poligon does not allow the encoding of arbitrary knowledge into procedures. Only side-effect free functions are allowed. Side-effects are permitted only in the bodies of rules, where they can be controlled.

4.1.3. The Sequencing of Activities

In most blackboard systems knowledge of the required sequencing of events at a macroscopic level is expressed by the implementation of the system's scheduler. In many cases, such as AGE [Nii 79] this scheduler has fixed characteristics and the application has a fixed interface to it. In others, such as MXA [Rice 84], the user can specify the characteristics of the scheduling of knowledge sources. Poligon provides no such mechanism. Since all rules are activated as daemons, entirely asynchronously, the only analogue of scheduling is the implicit sequencing of the activation of rules due to some rules causing changes that trigger other's rules.

4.1.4. The Structure of the Solution Space

Poligon is unlike most blackboard systems in this respect. Most blackboard systems partition the blackboard into *Levels*, which represent the hierarchy of abstraction in the solution space. Poligon uses a much more general representation which is like that of some *Frame* systems, providing a "Class" mechanism with user defined classes and metaclasses, and compile-time and run-time inheritance. The functionality of the class mechanism in Poligon is a superset of that of the levels provided by most blackboard systems. The programmer can, of course, represent his solution simply using classes as levels in Poligon if he wishes. Classes are discussed more in § 4.3.1.

4.1.5. Knowledge about Relationships

Relationships between entities in blackboard systems are often expressed by a form of *Link* mechanism. Sometimes this link is not so much a part of the system as a reflection of the fact that fields in nodes can have as their values other nodes in the system. Other systems have more sophisticated mechanisms that express links explicitly and allow property inheritance along links, e.g. BB1, or the propagation of likelihood, e.g. MXA.

Poligon has a number of system defined relationships; "Is an Instance of", "Is a part of" and "Is a subclass of". The user can define arbitrary relationships between nodes on the blackboard. These links allow property inheritance and are, themselves, represented as nodes and so

---

[4]It should be noted that the term *Focussing mechanism* is used in a more general sense than by many blackboard systems. There can be any number of such foci all acting in parallel in a Poligon program. The expectation mechanism is another way of applying knowledge in order to take advantage of some local circumstances in order to solve a problem more efficiently or cleanly.

[5]A rule is said to have been *Triggered* when it is activated so that it tries to evaluate its preconditions and body.

The following is a trivial example rule, which shows a small set of the features of Poligon. This rule could be interpreted as saying: "*If the most recent two phonemes that have been seen are "oo" and "ph" then the word is "foo"*". Having concluded this the rule finds the set of sentence components, which represent potential conclusions of the word "foo", and sets them so that they are no longer marked as hypothetical. It also makes a *Sentence-Component* type node, which represents the word "foo", which has been found.

```
Rule : Find-the-word-Foo
    Class : Phoneme
        { Class of nodes with which the rule will be associated }
    Field : uncorrelated-phonemes
        { Try to activate this rule when this field is changed }

Definitions :
    all-phonemes-in-order ≡ The-Phoneme⊕↑uncorrelated-phonemes

        { The operator "⊕↑" returns all values in a field in }
        { time order.  The-Phoneme represents the node, that }
        { triggered this rule }
    most-recent-phoneme ≡ all-phonemes-in-order·Head
    next-most-recent-phoneme ≡ all-phonemes-in-order·Tail·Head
        { Head and Tail are like CAR and CDR only they operate }
        { on lists, Lazy lists and Bags }

Condition Part :
    When : all-phonemes-in-order·length-of-list ≥ 2
        { The "When" part is a locally evaluable precondition }
    If   : most-recent-phoneme·Sound = "oo"
           And next-most-recent-phoneme·Sound = "ph"
        { The precondition for the Rule }

Action Part :
    Definitions :
        new-sentence-component ≡ New Instance of Sentence-Component
        { The creation of the new Sentence-Component node }
        hypothetical-foos ≡
        { A Bag of words, which are "foo" }
            Subset of Words which satisfies

                λ(a-word)

                    a-word·hypothetised And a-word·letters = [ f o o ]

                Endλ

        { Process all elements in the Bag hypothetical-foos }
    Changes :
        In Parallel for each a-word in hypothetical-foos
            Change Type    : Update
            Updated Node   : a-word
            Updated Fields : hypothetised ← nil

        { Set fields of new sentence component in }
        { parallel with updating the elements in the Bag }
    Changes :
        Change Type    : Update
        Updated Node   : new-sentence-component
        Updated Fields : letters ← [ f o o ]
                         constituents ← List(next-most-recent-phoneme,
                                             most-recent-phoneme)
```

All of the actions taken by this rule are performed in parallel, since they are independent of one another, though there is, of course, a serial dependency between the condition part and the action part of the rule.

Figure 4-1:   An example Poligon rule

can have attributes in the same way that any other nodes can. Links are therefore first-class citizens in Poligon and they allow Poligon programs to act like semantic nets.

4.2. Control Abstractions

The flow of control is a rather evanescent concept in a Poligon program. Any rule can be triggered at any time. It is important not to think of the control flow in a Poligon program in the same terms as that of a conventional serial program. There is a well defined flow of control within rules; the action part of a rule is activated after the condition part, upon which it is predicated. Apart from this, however, there is no flow of control in any normal sense. It should be noted also that what little flow of control there is only specifies the strict ordering of activities. The execution of a sequence of actions can be interrupted at any time. The size of the atoms for Poligon's atomic actions is very small.

The triggering of rules is controlled by the user associating rules with particular fields of nodes or classes of nodes on the blackboard. The triggering of rules occurs when a field, which is being watched in such a manner, is updated or is semaphored. A semaphore mechanism is provided to allow rules to be triggered without a field being updated. This provides a form of explicit event-based programming, if it is needed.

Clearly one of the objectives of the design of the Poligon language is to provide a language in which it is simple to express logically distinct pieces of knowledge, independent of other such pieces of knowledge. The decomposition of the problem in this manner causes the system to appear to iterate towards the solution of its problem by small, simple and discrete steps, rather than by complex, giant leaps.

4.3. Data Abstractions

Poligon provides a number of distinct data abstractions. One is characteristic of other blackboard systems, one of pure functional languages and one is rather novel.

- The structure of the blackboard is characterised by being made of *Nodes*, elements in the solution space. These have a user-defined, record-like structure.

- Lazy evaluation is supported.

- Bags are supported as data structures, which parallelism enhancing.

Numerous operations are defined for these data abstractions, particularly a number of generic operations which can be applied to lists, lazy lists and bags, which shield the user from the underlying data structures used by the system or by other segments of his program.

4.3.1. The Structure of the Solution Space

The most obvious data abstraction provided by Poligon is similar to that provided by conventional blackboard systems, that is, the *Node* on the blackboard as an element in the solution space. Such nodes are record-like internally. They have named fields, which can often contain multiple values to be associated with that name. Poligon provides this but also goes beyond it.

Conventional blackboard systems, such as AGE, tend to provide nodes on a blackboard divided into groups, often called "Levels". "Levels" themselves are not represented. Arbitrary use of global data, held in global variables, distinct from the blackboard is also allowed.

Poligon has a much more regular representation for data. The nodes are represented as instances of *Classes*. The Classes themselves are represented as Nodes, which "control" their instances. Knowledge concerned with classes as a whole can be associated with these nodes. Shared, global variables are not allowed in Poligon.

Poligon also provides;

Superclasses   Classes that provide characteristics to the instances of classes. These can be thought of as templates for the instances.

Metaclasses   Classes that provide characteristics to the classes themselves. These can be

thought of as templates for the classes.

Thus the classes are themselves instances of metaclasses, which can be user defined, such that instances of a given class can have any number of superclasses, i.e. component templates, and any number of metaclasses, i.e. component templates for their parent class. It is possible to instantiate classes any number of times, as well as their instances.

Automatic property inheritance allows shared data to be located on locally central nodes, which are immediately visible to the interested parties. This distributes shared data in such a manner as will, hopefully, minimise hot-spotting.

An example class declaration, the specification of a template for a class of nodes, is shown below. The declaration defines a class of nodes called *Words*, each instance of which has two fields (slots) called *Letters* and *Sound*.

```
Class Words :
    Fields :
        Letters
        Sound
```

Extensions to this sort of syntax allows the definition of superclasses and metaclasses within class declarations. The following example defines the class *Sheep*. Each instance of the class *Sheep* will have the characteristics defined for sheep and for mammals. The class called *Sheep* (an instance, in fact, of the class *Meta-Sheep*) has the characteristics of *types of animals*.

```
Class Types-of-animals :
    Fields :
        Rate-Of-Breeding

Class Mammals :
    Fields :
        Colour-of-fur
        Number-of-legs : 4

Class Sheep :
    Metaclasses  : Types-of-animals
    Superclasses : Mammals
    Fields :
        Thickness-of-wool
        Flock
```

4.3.2. Lazy Evaluation

Lazy Evaluation is supported in the guise of *Lazy Lists*, *Lazy Function Arguments* and in the form of the lazy association of expressions with names. The following is an example of the lazy association of a name with a value. The name *A-Meaningful-Name* is associated with the value of the call to the function *An-Expensive-Function*[6].

```
Definitions :
    A-Meaningful-Name ≡
        An-Expensive-Function(an-arg, another-arg)
```

The value of an item defined in a *Definitions* construct is always a future if it is possible to evaluate it as a future.

---

[6]Suitable *Force* operations are provided so that the time of evaluation can be controlled by the program if necessary. These force operators allow the program to perform *Eager Evaluation* if it is needed.

4.3.3. Bags

One abstraction suited particularly to the parallel mode of execution of Poligon programs is the *Bag* data type. Bags are implemented in Poligon so that they are formed as the result of efficient parallel operations and can be processed in parallel efficiently. Even when the elements of Bags are processed serially they perform efficiently. The lack of a defined ordering · in the Bag means that the system can always return the first satisfied Future out of a Bag of Futures, causing minimum waiting for values. Similarly, when a program attempts to extract an element from a bag and there are no satisfied elements the process in which this happens will go to sleep until the next available future is satisfied.

A Bag is generated, for instance, as the value of the following expression. It is a Bag, which contains all of the *Words*, whose *Sound* is "phoo"[7].

```
Subset of Words For Which Element · Sound = "phoo"
```

4.4. Parallelising Abstractions

Poligon supports data representations which are designed to give the user a high level handle on the exploitation of parallelism. Most values computed in Poligon are derived as *Futures*. Computation is decoupled from the expressions which reference values. Futures are, however, completely invisible to the user in Poligon. It understands which functions are strict in their arguments and so waits for the satisfaction of a Future only when it is required. The programmer can, of course, declare his own non-strict functions and operators. All *DeFuturing* coercions are performed automatically by the Poligon system. Thus the following expression will deliver a list with two elements, one of which is the value of *a* and one of which is the sum of *b* and *c*. The first will be a future, if *a* is. The second will be the DeFutured value *b+c*.

```
List(a, b+c)
```

The efficient use of the bandwidth of the processor interconnection network is enhanced by the use of *Broadcast* and *Multicast* operations. Broadcast messages allow messages to be sent to every node in the system in a single operation. Multicast messages allow messages to be sent to a collection of nodes in a single operation. The Poligon system uses these extensively in the processing of the Bag data type and in the execution of groups of actions in parallel. It uses the same mechanisms to provide an efficient implementation for searching a collection of nodes on the blackboard for patterns, which tends to cause significant slowing of serial implementations because of the combinatorial nature of such searches. It allows the blackboard to be searched for bags of matching nodes in a single, fast operation. This provides a significant improvement over the serial construction of such collections.

4.5. Real-time processing

Real-time processing brings its own problems. Poligon provides a simple and regular mechanism for defining the interface between the Poligon system and its signal data. This data can be from an arbitrary number of different types of sources and is posted on the blackboard asynchronously.

Poligon also provides a mechanism by which each datum is timestamped from the time that it enters the system. These timestamps are propagated automatically by the system so that it is trivial for the programmer to manipulate time-ordered collections of values. This mechanism is required because the conventional implicit time ordering of data in lists cannot apply here

---

[7]The expression "Element · Sound" denotes extracting one of the values associated with the "Sound" field of the potential element in the bag. "·" is an operator that selects which of the values associated with the field is to be delivered.

and the non-ordered nature of Bags is sometimes not sufficient.

4.6. The control of assignment

Assignment is something which is likely to cause significant problems in any parallel system. Poligon constrains assignment in a number of ways. Side-effects are only permitted on the fields of nodes. All side-effects can be monitored by rules that might be interested in the changes to values. This removes the possibility of the knowledge base getting confused because of surgical side-effects to data structures at arbitrary times and at arbitrary places in the processor network. Assignment is also constrained so that all of the updates to the fields of a given node are done atomically, before any rules which might be triggered by these changes are allowed to trigger. Such atomicity helps to preserve the consistency of the system.

An example of a collection of updates to fields of a given node is given below. In this example the node *an-instance-of-words* is having two of its fields updated; *Sound* and *Letters*. Operators, such as "←", allow different sorts of modifications to be made to fields. Such operations might be "add this value to the values in this field" or "replace all of the values in the field". This avoids complex and potentially expensive expressions in the old value of the field being evaluated non-locally.

```
Change Type     :  Update
Updated Node    :  an-instance-of-words
Updated Fields  :  Sound    ← "phoo"
                   Letters ← [ f o o ]
```

5. Conclusions

This paper has described Poligon, a language and system for the investigation of problem solving on distributed-memory, parallel hardware. The language was described in the context of related work in the field and in terms of the abstraction mechanisms provided. No significant description of the underlying run-time support has been given.

The Poligon system is still young. Only recently have applications been mounted on it in earnest. Two distinct applications in the field of real-time signal processing are now being implemented and more applications are likely to be started in the near future. Poligon has proved to be well suited to these applications as far as they have gone. No results from the simulation process regarding the performance of Poligon programs are yet available. Significant problems have been found in the simulation of the fine-grained parallelism required by the Poligon metaphor. Such simulations are very time consuming, prone to bugs in the underlying system software and simulator, and are difficult to debug. It is for these reasons that Poligon also has a serial version, Oligon, which accurately emulates the behaviour of the parallel system but without true parallelism. A simulated processor array of 256 processors has recently been made available to the users of Poligon. This simulation will allow more satisfactory investigation of the properties of Poligon programs in the future.

6. Further Reading

For a significantly more detailed treatment of the Poligon language and system the reader is encouraged to consult [Rice 86].

The following topics were not described or discussed but are relevant to the work described above. The reader is encouraged to consult the following for further information;

- [KSL 85] for a description of the Advanced Architectures Project of which Poligon is a part.

- [Delagi 86] for a description of CARE, the hardware simulator used by Poligon, and of the particular hardware being simulated.

- [Schoen 86] for a description of CAOS, the concurrent object oriented system running on the CARE machine, which Poligon uses as its operating system.

- [Ensor 85], [Lesser 83], [Aiello 86] and [Fennel 77] for other approaches to parallel problem solving using blackboard systems.

References

[Aiello 86]   Aiello, Nelleke.
                The Cage User's Manual.
                Technical Report KSL-86-23, Heuristic Programming Project, C. S. Dept.,
                    Stanford University, 1986.

[Davies 86]   Davies, Byron.
                Carel: A Visible Distributed Lisp.
                Technical Report KSL-86-??, Heuristic Programming Project, C. S. Dept.,
                    Stanford University, 1986.

[Davis 77]   Davis, R. and J. King.
                An Overview of Production Systems.
                In E.W. Elcock and D. Michie (editor), *Machine Intelligence 8: Machine
                    Representation of Knowledge,* . John Wiley, New York, 1977.

[Delagi 86]   Bruce Delagi.
                CARE User's Manual
                Heuristic Programming Project, Stanford University, Stanford, Ca. 94305, 1986.

[Ensor 85]   Ensor, J. Robert and Gabbe, John D.
                Transactional Blackboards.
                Proc. of IJCAI 85 :340 - 344, 1985.

[Fennel 77]   Fennel, R. D. and Lesser, V. R.
                Parallelism in AI problem solving: a case study of Hearsay-II.
                IEEE Trans on Computers, C-26 :98-111, 1977.

[Gabriel 84]   Gabriel, Richard P. and McCarthy, John.
                Queue-based Multi-processing Lisp.
                Proceedings of the ACM Symposium on Lisp and Functional programming :25
                    - 44, August, 1984.

[Halstead 84]   Halstead, Robert H. Jr.
                Implementation of Multilisp: Lisp on a Multiprocessor.
                Proceedings of the ACM Symposium on Lisp and Functional programming :9
                    - 17, August, 1984.

[Hayes-Roth 85] Barbara Hayes-Roth.
                Blackboard Architecture for Control.
                Journal of Artificial Intelligence 26:251 - 321, 1985.

[Hewitt 73]   Hewitt, C., P. Bishop, and R. Steiger.
                A Universal, Modular Actor Formalism for Artificial Intelligence.
                Proceedings of IJCAI-73 :235 - 245, 1973.

[KSL 85]   Knowledge Systems Laboratory.
                *Knowledge Systems Laboratory 85, incorporating the Heuristic Programming
                    Project.*
                KSL, Dept of Computer Science, Stanford University, 1985.

[Lesser 83]   Lesser, Victor R. and Daniel D. Corkill.
The Distributed Vehicle Monitoring Testbed: A Tool for Investigation Distributed Problem Solving Networks.
The AI Magazine Fall:15 - 33, 1983.

[Nii 79]   Nii, H. P. and N. Aiello.
AGE: A Knowledge-based Program for Building Knowledge-based Programs.
Proc. of IJCAI 6 :645 - 655, 1979.

[Nii 86]   Nii, H. P.
Blackboard Systems.
AI Magazine 7:2, 1986.

[Rice 84]   Rice, J. P.
The MXA user's and writer's companion
Systems Programming Ltd, The Charter, Abingdon, Oxon, UK, 1984.

[Rice 86]   Rice, J. P.
The Poligon User's Manual.
Technical Report KSL-86-10, Heuristic Programming Project, C. S. Dept., Stanford University, 1986.

[Schoen 86]   Schoen, Eric.
The CAOS System.
Technical Report KSL-86-22, Heuristic Programming Project, C. S. Dept., Stanford University, 1986.

An Experiment in Knowledge-based Signal Understanding Using Parallel Architectures

by

Harold D. Brown, Eric Schoen, and Bruce A. Delagi

Knowledge Systems Laboratory
Computer Science Department
Stanford University
Stanford, California 94305

Table of Contents

List of Figures

List of Tables

Abstract

This report documents an experiment investigating the potential of a parallel computing architecture to enhance the performance of a knowledge-based signal understanding system. The experiment consisted of implementing and evaluating an application encoded in a parallel programming extension of Lisp and executing on a simulated multiprocessor system.

The choosen application for the experiment was a knowledge-based system for interpreting pre-processed, passively acquired radar emissions from aircraft. The application was implemented in an experimental concurrent, asynchronous object-oriented framework. This framework, in turn, relied on the services provided by the underlying hardware system. The hardware system for the experiment was a simulation of various sized grids of processors with inter-processor communication via message-passing.

The experiment investigated the effects of various high-level control strategies on the quality of the problem solution, the speedup of the overall system performance as a function of the number of processors in the grid, and some of the issues in implementing and debugging a knowledge-based system on a message-passing multiprocessor system.

In this report we describe the software and (simulated) hardware components of the experiment and present the qualitative and quantitative experimental results.

1. Introduction

This report documents an experiment investigating the potential of a parallel computing architecture to enhance the performance of a knowledge-based signal understanding system. This experiment was done within the Expert Systems on Multiprocessor Architectures Project of Stanford University's Knowledge Systems Laboratory.

The computational characteristics of complex knowledge-based systems are poorly understood, especially in parallel computational environments. Our Architectures Project is performing a number of experiments to try to gain some understanding of these characteristics and, in particular, of the potential for concurrent execution of such systems. A primary goal of the project is to develop software and hardware system architectures which exploit this concurrency to increase the performance of knowledge-based signal understanding and information fusion systems.

The Architectures Project is organized according to a hierarchy of computational abstraction levels as shown in Table 1-1. Each experiment represents a narrow, vertical slice through these levels and consists of a specific system choice for each level.

For the reported experiment, the choosen application is a knowledge-based ELINT (ELectronics INTelligence) system for interpreting processed, passively acquired radar emissions from aircraft. The ELINT application is implemented in CAOS, an experimental concurrent, asynchronous object-oriented framework built on Zetalisp [1]. The CAOS framework, in turn, relies on the services provided by the underlying hardware system environment. For this experiment, the hardware system environment is a simulation of a parallel architecture, called CARE [2]. CARE simulates a communications grid of processing sites where each site contains a Lisp evaluator, private memory, and a communications and process scheduling subsystem. Message-passing is the only means of inter-site communication. CARE is simulated using a general, event-based simulator, SIMPLE [3]. SIMPLE is written in Zetalisp and executes on a Symbolics 3600 or a Texas Instruments Explorer Lisp machine.[1] Figure 1-1 illustrates the relationship between the various software components of the experiment.

The ELINT-CAOS-CARE experiment investigated both qualitative and quantitative aspects of the performance of the overall system. The CARE architecture uses dynamic, cut-through (as

---

[1] A version of the SIMPLE simulator which runs on a local area network of multiple Lisp machines has also been implemented [4].

Table 1-1:   Computational levels.

Level	Research questions
Application	Where is the potential concurrency in knowledge-based signal understanding tasks?
	How does the problem solver recognize and express application dependent concurrency?
Problem-solving framework	What are suitable framework constructs for organizing and encoding concurrent signal understanding tasks?
	What are appropriate granularities for knowledge, knowledge application and data to maximize concurrency?
	What types of strategies for control of knowledge application are needed to assure acceptable solution quality without introducing excessive execution serialization?
Knowledge representation and management	What kinds of knowledge representation mechanisms are suitable for exploiting concurrency in inference and search?
System programming language	How can general-purpose symbolic programming languages be extended to support concurrency and help manage the resource allocation and reclamation tasks on a distributed memory multiprocessor?
Hardware system architecture	What multiprocessor architectures best support the organization and concurrency in knowledge-based signal understanding applications?

opposed to store and forward) routing through the communication grid for interprocessor message transmission. Message transmission time is indeterminate. As a consequence, without the imposition of significant message sequencing protocols (and the corresponding serialization of execution), operations are intrinsically non-deterministic in the sense that two executions of the same program on the same input data can result in different problem solutions depending on different message arrival orders. For many knowledge-based systems, in particular, the ELINT system, there is no such thing as *the* correct problem solution but only *satisficing* (i.e., acceptable) problem solutions. One primary objective of the experiment was to investigate the trade-offs between the imposition of various synchronizations (and the resulting loss of concurrency) and the quality of the problem solution. A second primary objective was the more usual investigation of the speedup of the overall system performance as a function of the number of processing sites in the CARE grid. A third objective was to gain some understanding of the difficulties in implementing and debugging a reasonably complex knowledge-based system on a multiple address space, message-passing multiprocessor system such as that represented by CARE.

ELINT	Interpretation of radar emissions from aircraft
CAOS	Concurrent, asynchronous object system
Zetalisp+	Zetalisp plus locality and communication constructs
CARE	Grid-based, message-passing multiprocessor specification
SIMPLE	Hardware specification system and event-driven simulator
Zetalisp	
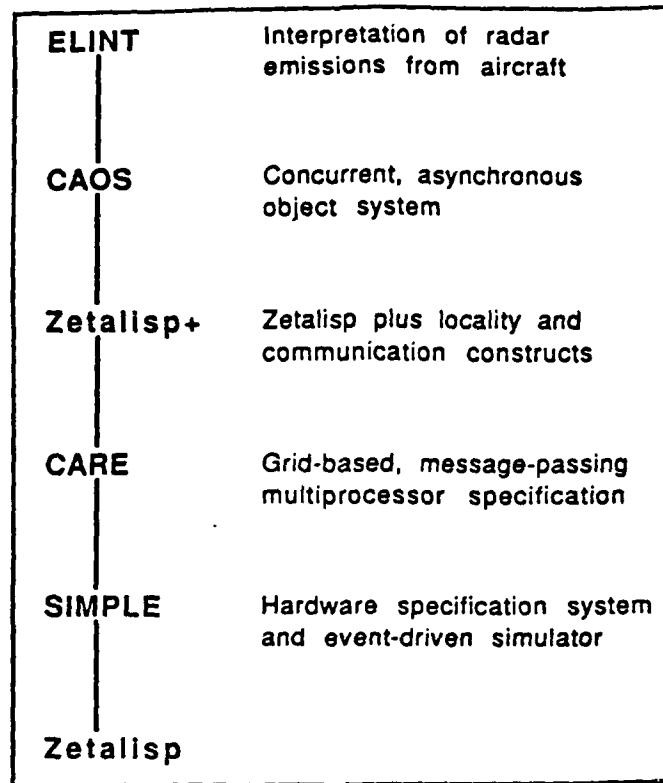
Figure 1-1: The software component hierarchy of the experiment.

In the following sections we describe, in decreasing hierarchical order, each component of the experiment. Section 2 describes the ELINT application. Section 3 gives an overview the CAOS programming framework and its approach to concurrency. ELINT's implementation in CAOS is described in Section 4, and Section 5 describes the salient features of the CARE architecture and its simulation environment. In Section 6 we present the results of the ELINT-CAOS-CARE experiment.

2. The ELINT Application

The driving application for our vertical slice experiment is a prototype, knowledge-based ELINT system for interpreting processed, passively acquired, real-time radar emissions from aircraft. This ELINT system is one component of a multi-sensor information fusion system, TRICERO [5] developed several years ago. ELINT was originally implemented in AGE [6], an expert system development tool based on the blackboard paradigm [7, 8]. ELINT is a relatively simple, but non-trivial, knowledge-based system. Much of its knowledge is implemented procedurally. However, if ELINT had been implemented as a production rule

system, we estimate that its knowledge base would consist of about one thousand rules.[2]

ELINT's basic analysis technique is to correlate a large number of passively observed radar emissions into the smaller number of individual radar emitters producing those emissions. It then correlates the emitters into the yet smaller number of clusters of co-located emitters. ELINT maintains the track and activity histories of the clusters

2.1. ELINT's Inputs

The inputs to the ELINT system are multiple, time-ordered streams of processed observations from multiple collection sites. Each observation is presented in a record format. The fields of an input observation record are shown in Table 2-1.

Table 2-1: Elint observation record.

Field	Contents
Observation-Time	An integer time-tag indicating when the radar emission was sampled
Observation-Site	The symbolic name of the collection site acquiring the observation
Site-Location	The positional coordinates of the collection site at the time of observation
Emitter-Identifier	An integer identifing the radar emitter producing the emission
Line-of-Bearing	The line of bearing from the collection site to the observed emitter
Emitter-Type	A symbolic radar emitter type designator
Emitter-Mode	The operational mode of the emitter at the time of observation
Signal-Quality	A symbolic indicator of the signal quality of the observed emission

The Site-Location field is necessary since the collection sites can be mobile. The Emitter-Identifier is a unique integer identifier assigned by the collection sites to each distinct observed emitter. This identifier is used by the collection sites to indicate multiple observations of the same emitter both over time and from different collection sites. In particular, two concurrent observations of the same emitter from different collection sites

---

[2]In general, there are currently no adequate metrics for measuring the complexity of knowledge-based systems. One crude measure used for rule-based systems is the number of rules. Although the number of rules does somewhat indicate the amount of knowledge, it does not give much indication of the complexity of the reasoning.

should have the same identifier. Both the intra-site and inter-site determination of whether two observed emissions are from the same emitter are based on the electronic characteristics of the emissions and on signature analysis. This determination may be in error, and the ELINT system must cope with such identifier errors. The **Emitter-Type** of a radar emitter indicates the functional class of the emitter, for example, Air-Intercept (AI), Navigation (NAV) or Identification-Friend-Or-Foe (IFF), and, if known, the equipment type class of the emitter. Certain classes of emitter types can have multiple operational modes. The **Emitter-Mode**, if applicable, is emitter-type specific. For example, an AI radar can be either in Search Mode or Lock-on Mode depending on whether it is scanning for a target or whether it is automatically tracking a specific target. The **Signal-Quality** of an observation is a subjective, qualitative measure of the strength of the observed emission, for example, *strong, normal,* or *fading*.

All of the input information required for the ELINT system is obtainable from the raw radar signal data using current, passive radar signal collection and processing techniques. These techniques are largely automated and employ special-purpose hardware.

2.2. ELINT's Outputs

The primary outputs of the ELINT system are periodic status reports about the tracks and activities of clusters of emitters in the area under surveillance. A cluster is defined as a collection of emitters which are co-located over time. That is, two emitters are in the same cluster if for some given minimum number of consecutive time units (three in the current ELINT system) their corresponding time-tagged locational fixes are within a distance determined by the line-of-bearing resolution of the observation site equipment (one degree resolution in the current ELINT system). Conceptually, two emitters are in the same cluster if if they are on the same aircraft or are on two tactically associated and co-located (over time) aircraft, for example, a lead aircraft and his wingman.[3]

The periodic output reports contain, for each cluster, information about the cluster's current

---

[3]An aircraft can be operating with some (or all) of its radars off. In general, it is impossible to distinguish between, for example, two co-located aircraft, one with an AI radar on and one with a NAV radar on, and one aircraft with both its AI and NAV radars on. Hence, our ELINT system does its assessments based on emitter clusters rather than aircraft.

heading, position and track; an estimate of the number and types of aircraft in the cluster;[4] an indication of the cluster's current activity; and an indication if the cluster represents an immediate threat, for example, if it is within a certain proximity of a friendly aircraft, if its AI radar is in Lock-on Mode, or if its missile guidance radar is on.

2.3. ELINT's Processing Flow

The basic reasoning strategy used by the ELINT application is data-driven accumulation of evidence for the existence, the tracks, and the activities of emitters and clusters based on input observations and infered information. The primary processing flow is a kind of pipeline where the pipeline stages are observations, emitters and clusters.

Upon receipt of a new observation, the system first determines if the observed emission *matches* (i.e., has as a source) a known emitter (i.e., an emitter on ELINT's "situation board"). This match is based on the Emitter-Identifier assigner by the collection site to the observation, and it is verified using the emitter's characteristics and its track and heading histories. Depending on the outcome of the match, one of the following actions is taken:

1. If the observation does not match a known emitter, then a new emitter which is the source of the observed emission is hypothesized on the situation board and initialized from the information contained in the observation.

2. If the observation does match an emitter on the situation board and the match is verified, then the information contained in the observation is used to update the attributes of the matched emitter, including increasing the confidence level of the hypothesis that the emitter represents. Moreover, if the new observation is the second (or greater) observation of the emitter for the current time and it is from a different collection site than the previous observation(s) at that time, then a locational fix for the emitter is computed using the observed lines of bearing. If, in addition, the Emitter-Type and/or Emitter-Mode indicate a near-term threat to a friendly aircraft, then a threat report is output.

---

[4]Knowledge relating an aircraft type, for example F-15 or MIG-3, with the number and types of radars it carries is available. Using this knowledge and the identified emitter types in a cluster, it is possible to roughly estimate bounds on the number and types of aircraft in the cluster.

3. If the observation matches a known emitter but fails the match verification test, then an error in the Emitter-Identifier is indicated and the situation board is modified so as to undo any incorrect inferences based on the error. Also, an identifier error report is output to the collection sites.

On a periodic basis, the status of each emitter on the situation board is evaluated and various actions are taken:

1. If there have been no recent observations of the emitter, then the confidence level of the emitter is reduced. If, as a consequence of this reduction, that level falls below a given *no-confidence* threshold, then the emitter and all of the consequences infered from it (including cluster association) are deleted from the situation board.

2. If the confidence level is above a given *full-confidence* threshold and the emitter is not currently associated with a known cluster, then an attempt is made to *match* the emitter with a cluster on the situation board. This match is based on the track and heading histories and the type attributes of the emitter and the cluster. If a match is made, then the emitter is associated with the matched cluster and the emitter's current attributes are used to update the attributes of the cluster. If the match fails, then a new cluster is hypothesized on the situation board and the emitter is associated with it.

3. In the remaining case of a recently observed emitter with an associated cluster, the current attributes of the emitter are used to update the attributes of its associated cluster.

Also on a periodic basis, the state of each hypothesized cluster on the situation board is examined. If all of the emitters associated with the cluster have been deleted, then the cluster is deleted from the situation board. Otherwise:

1. The cluster is checked to see if it should be *split* into two (or more) clusters based on the currrent locations of its associated emitters. If so, new clusters with the appropriate associated emitters are hypothesized on the situation board.

2. The track history, heading history, speed history and activity history of the cluster are updated; and, if any new emitters have been recently associated with the cluster, an estimate of the types and numbers of aircraft comprising the cluster is derived.

3. A current status report for the cluster is output.

The ELINT processing flow lends itself naturally to concurrent execution. The parallel implementation of ELINT using CAOS is described in Section 4. The CAOS system itself is described in the following section.

3. The CAOS Programming Framework

CAOS is a framework which supports the encoding and the execution of multiprocessor expert systems. It represents an early attempt to bridge the gap between the application specification and the multiprocessor system programming primitives. The design of CAOS is predicated on the belief that many highly parallel architectures (e.g., hundreds of processors) will emphasize limited communication between processor-memory pairs rather than uniformly shared memory. We expect that such an architecture will favor relatively coarse-grained problem decomposition with little synchronization between processors. CAOS is intended for use in real-time, data interpretation applications such as continuous speech recognition and radar and sonar signal interpretation (see, for example, [9, 10]). CAOS is based on an object-oriented programming paradigm, and it draws many of its ideas from the Flavors system [1] and the Actors paradigm [11].

A CAOS application consists of a collection of communicating, active *agents*, each responding to a number of application-dependent, predeclared messages. An agent retains long-term local state. Each agent is a multi-process entity, that is, an arbitrary number of processes may be active at any one time in a single agent.[5] Conceptually, an agent can be thought of as virtual, multiprocess processor and memory pair. It responds to externally sent messages, and these message responses can alter the state of its local memory and can include the sending of messages to other agents.

CAOS is designed to express parallelism at a relatively coarse grain-size. For example, in the ELINT experiment, the message handlers (i.e., the *methods*) which implement the message responses are written as Lisp procedures, each averaging about one hundred lines of primitive Lisp code. CAOS supports no mechanism for finer-grained concurrency such as within the execution of agent processes, but neither does it rule it out. We could easily imagine message

---

[5] The active processes in an agent are not scheduled preemptively. Instead, an executing agent process either runs to completion or until it is "blocked" awaiting some remote service (see Section 5).

methods being written, for example, in QLisp [12], a concurrent dialect of CommonLisp which supports finer-grained concurrency.

3.1. CAOS' Approach to Concurrency

A CAOS application is structured to achieve high degrees of concurrency in the application execution in two principal manners: *pipelining* and *replication*. Pipelining is most appropriate for representing the flow of information between levels of abstraction in an interpretation system. Replication provides means by which the interpretation system can cope with arbitrarily high data rates.

3.1.1. Pipelining

Pipelining is a common means of parallelizing tasks through a decomposition into a linear sequence of concurrently operating stages. Each stage is assigned to a separate processing unit which receives the output from the previous stage and provides input to the next stage. Optimally, when the pipeline reaches a steady-state, each of the processors is busy performing its assigned stage of the overall task.

CAOS promotes the use of pipelines to partition an interpretation task into a sequence of interpretation stages where each stage of the interpretation is performed by a separate agent. As data enters one agent in the pipeline, it is processed, and the results are sent to the next agent. The data input to each successive stage represents a higher level of abstraction.

Sequential decomposition of a large task is frequently very natural. Structures as disparate as manufacturing assembly lines and the arithmetic processors of high-speed computing systems are frequently based on this paradigm.

Pipelining provides a mechanism whereby concurrency is obtained without duplication of mechanism (i.e., machinery, processing hardware, knowledge, etc.). In an optimal pipeline of n processing elements, the throughput of the pipeline is n times the throughput of a single processing element in the pipeline.

Unfortunately, it is often the case that a task cannot be decomposed into a simple linear sequence of subtasks. Some stage of the sequence may depend not only on the results of its immediate predecessor, but also on the results of more distant predecessors, or worse, some distant successor (e.g., in feedback loops). An equally disadvantageous decomposition is one in which some of the processing stages take substantially more time than others. The effect of either of these conditions is to cause the pipeline to be used less efficiently. Both these

conditions may cause some processing stages to be busier than others. In the worst case, some stages may be so busy that other stages receive almost no work at all. As a result, the n-element pipeline achieves less than an n-times increase in throughput. We discuss a partial remedy for this situation below.

3.1.2. Replication

Concurrency gained through replication is ideally orthogonal to concurrency gained through pipelining. Any size processing structure, from an individual processing element to an entire pipeline, is a candidate for replication. Consider a task which must be performed on the average in time t, and a processing structure which is able to perform the task in time T, where $T > t$. If this task were actually a single stage in a larger pipeline, this stage would then be a bottleneck in the throughput of the pipeline. However, if the single processing structure which performed the task were replaced by T/t copies of the same processing structure, the effective time to perform the task would approach t, as required. Replication is more costly than pipelining, but it does avoid some of the problems associated with developing a pipelined decomposition of a task.

Our work leads us to believe that such replicated computing structures are feasible, but not without drawbacks. Just as performance gains in pipelines are impacted by inter-stage dependencies, performance gains in replicated structures are impacted by inter-structure dependencies.

Consider a system composed of a number of copies of a single pipeline. Further, assume the actions of a particular stage in the pipeline affects each copy of itself in the other pipelines. In an expert system, for example, a number of independent pieces of evidence may cause the system to draw the same conclusion. The system designer may require that when a conclusion is arrived at independently by different means, some measure of confidence in the conclusion is increased accordingly. If the inference mechanism which produces these conclusions is realized as concurrently operating copies of a single inference engine, the individual inference engines will have to communicate between themselves to avoid producing multiple copies of the same conclusion rather than a composite conclusion. Any consistency requirement between copies of a processing structure decreases the throughput of the entire system, since a portion of the system's work is dedicated to inter-system communication. Examples of this situation are shown in Section 4 where we describe the CAOS agent types for the ELINT application.

3.2. Programming in CAOS

CAOS is basically a package of operators on top of Lisp. These operators are partitioned into three major classes -- those which declare agent classes, those which initialize agents, and those which support communication between agents. We now describe briefly the CAOS operators for each of these classes. A more complete description of these operators is given in [13].

3.2.1. Declaration of Agents

Agents classes, like most object-oriented classes, are declared within an inheritance network. Each agent class inherits the attributes of its (multiple) parents. The root CAOS agent class, vanilla-agent, contains the minimal attributes required of a functional CAOS agent. All other CAOS agents have the vanilla-agent as a parent, either directly or indirectly. Another CAOS-declared agent class, process-agenda-agent, is a specialization of vanilla-agent, and includes a priority mechanism for scheduling the execution of messages. The vanilla-agent schedules its messages in a FIFO manner only.

Application agent classes are declared by augmenting the following primary attributes of CAOS-declared or other ancestral agent classes:

Local-Variables: An instance agent's local variables store its private state. The agent's message handlers may refer freely to only those variables declared locally within the agent. Each local variable may be declared with an initial value.

Messages-Methods: The only messages to which an agent may respond are those declared in the agent's class declaration. Associated with each declared message name is the name of the message's *method* (i.e., the message's message handler). In CAOS, a method name must refer to a defined Lisp procedure. This declaration simplifies the task of a resource allocator which must load application code onto each CARE site.

Clocks-Methods: An agent may periodically invoke actions based on internal clock "ticks." For example, the periodic update of emitter agents and the periodic output of cluster status reports are invoked by clock ticks. A clock is defined by its tick interval. Whenever an internal agent clock ticks, the set of methods associated with that clock are scheduled for execution.

Critical-Methods: This attribute declares certain sets of methods as being mutually "critical

regions" for their owning agents.[6] Each such set of critical methods has an associated *lock*. Before an owning agent agent executes a critical method, this lock is checked. If it is unlocked, the agent locks it and executes the method. Upon completion of the method, the agent unlocks the lock. If the lock is locked, the method is queued in a FIFO queue awaiting the unlocking of the lock.

There are a number of additional basic agent attributes. However, most of these are used only internally by CAOS.

3.2.2. Initialization of agents

An initial CAOS configuration is specified by a two-component initialization form. The first component of the form creates the *static* agent instances. Some agent instances are created during system initialization and exist throughout a CAOS run. Such agent instances are called static agents as opposed to *dynamic* agents which are created (and possibly deleted) during program execution. For programmer convenience, we allow code in agent message handlers and default values of local-variables to reference such static agents by name. Before an agent instance begins running, each symbolic reference to the declared static agents is resolved by the CAOS runtimes.

The second component of the form is a list of expressions to be evaluated sequentially when CAOS's static agent instantiation phase is complete. Each expression is intended to send a message to one of the static agents declared in the first part of the form. These messages serve to initialize the application. For example, in the ELINT application the initialization messages open log files and start the processing of ELINT observations.

Agent instances may also be created dynamically during execution. The creation operator accepts an agent class name and a location specification.[7] The *remote-address* of the newly-created agent instance is returned. The remote-address of an agent includes the CARE site coordinates where the agent resides and a pointer to the agent in the address space of that

---

[6]A design goal for ELINT in CAOS was to avoid the use of critical methods, and our ELINT implementation does not use any. The CAOS initialization routines, however, do use such methods.

[7]Currently, agents may be created only "at" or "near" specified CARE sites. CAOS makes no attempt at dynamic load balancing.

site. A dynamically created agent may *not* be referenced symbolically, however, its remote-address may be exchanged freely.

3.2.3. Communications Between Agents

Agents communicate with each other by exchanging messages. CAOS does not guarantee when messages reach their destinations. Due to excessive message traffic or processing element failure, messages may be delayed indefinitely during routing. It is the responsibility of the application program to detect and recover from such delayed messages.

Two classes of messages are defined: those which return values, called *value-desired* messages, and those which do not, called *side-effect* messages. The value-desired messages are made to return their values to a special cell called a *future* which represents a "promise" for an eventual value.[8] Processes attempting to access the value of a future are blocked until that future has had its value set. Futures are first-class data types, and they may be manipulated by non-strict Lisp operators (e.g., list) even if they have not yet received a value. It is possible for the value of a CAOS future to be set more than once, and it is possible for there to be multiple processes awaiting a future's value to be set.

The CARE primitive **post-packet**, which sends a packet from one process to another, is employed in CAOS to produce three basic kinds of message sending operations:

post: The **post** operator sends a side-effect message to an agent. The sending process supplies a remote-address to the target agent (or its name in the case of a static agent), the message's routing priority, and the message's name and arguments. The sender continues executing while the message is delivered to the target agent.

post-future: The **post-future** operator sends a value-desired message to the target agent. The sending process supplies the same parameters as for **post**, and it is immediately returned a local pointer to the future which will eventually receive a value from the target agent. As for **post**, the sender continues executing while the message is being delivered and executed remotely. A process may later check the state of the future with the **future-satisfied?** operator or access the future's value with the **value-future** operator. This latter operator will *block* the process (i.e., suspend its execution and "swap it out") if the future has not yet received a value. When the

---

[8]Futures are also used in Multilisp [14]. The HEP Supercomputer [15] implemented a simple version of futures as a process synchronization mechanism.

future finally receives a value, the blocked process is rescheduled for resumed execution.

post-value: The **post-value** operator is similar to the **post-future** operator except that the sending process is immediately blocked until the target agent has returned a value. This operator is defined in terms of **post-future** and **value-future**, and it is provided for programming convenience.

It is possible to detect delay of value-desired messages by attaching a timeout to the associated future. The operators **post-clocked-future** and **post-clocked-value** are similar to their untimed counterparts but allow the caller to specify a *timeout-period* and *timeout-action* to be performed if the future is not set within the timeout-period. Typical timeout-actions include setting the future's value to a default value or resending the original message using the **repost** operator.

There also exist versions of the basic posting operators which allow the same message to be sent to multiple agents simultaneously. These versions exploit the multicast facilities of CARE (see Section 5).[9]

Multipost sends a side-effect message to a list of agents while **multipost-future** and **multipost-value** send value-desired messages to lists of agents. In the latter two cases, the associated future is actually a list of futures, and the future is not considered satisfied until all the target agents have responded. The value of such a message is an association-list where each entry in the list is composed of an agent's remote-address or name and the returned message value from that agent. There exist clocked versions of these operators (called, naturally, **multipost-clocked-future** and **multipost-clocked-value**) to aid in detecting delayed multicast messages.

3.3. The Runtime Structure of CAOS

CAOS is structured around three principal levels: site, agent, and process. Two of these levels, site and process, reflect the organization of CARE. The remaining agent level is an artifact of CAOS. We describe here only briefly the runtime structure of CAOS. This structure is described in greater detail in [13].

---

[9]Neither CAOS nor CARE currently support a "predicated multicast" mode wherein messages would be sent to all agents satisfying a particular predicate. Messages can only be multicast to a fully-specified list of agents. Receiving agents can, of course, apply arbitrary predicates to the message in order to determine their consequent action.

The implementation of CAOS described in this report is written in Zetalisp [1] and the primitive CARE operators using Zetalisp's object-oriented programming tool, Flavors[1].

Each CARE site contains a CAOS Site-Manager. A Site-Manager is realized as a Flavors instance. Its instance variables store site-global information needed by all agents located on the site. In addition, each Site-Manager includes CARE-level processes which perform the functions of creating new agents on its site and translating static agent symbolic names into agent addresses.

Each CAOS agent is also realized as a Flavors instance. A CAOS agent is a multiprocess entity. Most of the processes are created in the course of problem-solving activity. These processes are refered to as user processes. At runtime, however, there are always two special processes associated with each CAOS agent -- the *agent input monitor process* and the *agent scheduler process*. The agent input monitor process watches the CARE stream by which the agent is known to other agents. It handles request messages and responses from value-desired messages from these agents. CAOS user processes are created in response to request messages from other agents or clocked methods. The agent scheduler process collaborates with the CARE site's operator processor in the scheduling of these user processes (see Section 5).

4. ELINT's Implementation in CAOS

We describe now the agent types and their organization for the ELINT application as implemented in the CAOS framework. This implementation illustrates some of the benefits and some of the drawbacks of the framework. As discussed in Section 2, ELINT is an expert system whose domain is the interpretation of passively-observed radar emissions. ELINT is meant to operate in real time. Emitters appear and disappear during the lifetime of an ELINT run. The primary flow of information in ELINT as implemented in CAOS is through a pipeline with replicated stages. Each stage in the pipeline is an agent. The basic ELINT agent pipeline is illustrated in Figure 4-1
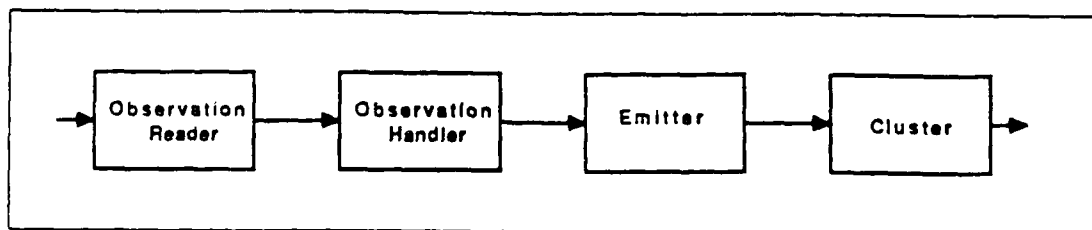
Figure 4-1: The basic ELINT agent processing pipeline.

4.1. ELINT Agent Types

The ELINT agent types described here are those used by the CT control strategy version of ELINT in CAOS (see Section 6).

Observation-Reader Agent

Observation-reader agents are an artifact of the simulated environment in which our ELINT implementation runs. Their purpose is to feed radar observations into the system. Observation-readers are driven off system clocks. At each clock "tick" (one ELINT time unit), they supply all observations for the associated time interval to the proper observation-handler agents. This behavior is similar to that of radar collection sites in an actual ELINT setting.

Observation-Handler Agent

The observation-handler agents accept radar observations from associated radar collection sites. Of course, in the simulated environment the observations actually come from observation-reader agents. There may be several observation-handlers associated with each collection site. The collection site chooses to which of its observation-handlers to pass an observation based on some scheduling criteria, for example, round-robin.

The contents of an ELINT observation was described in Section 2. In particular, each observation contains an identifier number assigned by the collection site to distinguish the source of the observation from other known sources. This source identifier is usually, but not always, correct. When an observation-handler receives an observation, it checks the observation's identifier to see if it already knows about the emitter which is the observation's source. If it does, it passes the observation to the appropriate emitter agent which represents the observation's source. If the observation-handler does not know about the emitter, it asks an emitter-manager agent to create a new emitter agent and then passes the observation to that new agent.

Emitter-Manager Agent

There may be many emitter-manager agents in the system. An emitter-manager's task is to respond to requests from observation-handlers to create new emitter agents with associated source identifier numbers. If there is no such emitter agent in existence when the request is received, the manager will create one and return its remote-address to the requesting

observation-handler agent. If there is such an emitter agent in existence when the request is received, the manager will simply return its remote-address to the requestor. This situation arises when one observation-handler requests an emitter that another observation-handler had previously requested. Emitter-managers must also handle the case of "almost concurrent" requests for the same emitter. This case occurs when a request is received for an emitter agent which is currently being created by another process on another CARE site in response to a slightly earlier request.

The reason for the emitter-manager's existence is to reduce the amount of inter-pipeline dependency with respect to the creation of emitters. When ELINT creates an emitter it is similar to a typical expert system drawing a conclusion based on some evidence. ELINT must create its emitters in such a way that the individual observation-handlers do not each end up creating copies of the "same" emitter, that is, creating multiple emitter agents with the same associated source identifier (see Section 3.1.2). Consider the following strategies that the observation-handler agents could use to create new emitter agents:

1. The handlers could create the emitter agents themselves immediately as needed. Since the collection sites may pass observations with the same source identifier to any observation-handler, it is possible for multiple observation-handlers to each create its own copy of the same emitter. This strategy is not acceptable.

2. The handlers could create the emitter agents themselves, but inform the other handlers that they have done this. This scheme breaks down when two handlers try simultaneously (or almost simultaneously) to create the same emitter.

3. The handlers could rely on a single emitter-manager agent to create all emitters. While this approach is safe from a consistency standpoint, it is likely to be impractical as the single emitter-manager could become a processing bottleneck

4. The handlers could send requests to one of many emitter-managers chosen by some arbitrary method. This idea is nearly correct, but does not rule out the possibility of two emitter-managers each receiving creation requests for the same emitter.

5. The handlers could send requests to one of many emitter-managers chosen through some algorithm which is invariant with respect to the source identifiers.

This last strategy is the one used used in our implementation of ELINT. The algorithm for choosing which emitter-manager to use is based on a many-to-one mapping of source identifiers to emitter-managers.[10]

Emitter Agent

Emitter agents hold the state and history of the observation sources they represent. As each new observation is received by an emitter agent, it is added to a list of new observations. On a periodic basis, this list of new observations is scanned for interesting information. In particular, after enough observations are received, the emitter may be able to determine the heading, speed, and location of the source it represents. The first time it is able to determine this information, it asks a cluster-manager agent to either match the emitter to an existing cluster agent (as described in section 2.3) or create a new cluster agent to hold the single emitter. Subsequently, it sends an update message to the cluster agent to which it is associated indicating its current heading, speed, and location.

Emitters maintain a qualitative confidence level of their own existence (*possible, probable, positive* and *was-positive*). If new observations are received often enough, the emitter will increase its confidence level until it reaches *positive*. If an observation is not received by an emitter in the expected time interval, the emitter lowers its confidence by one step. If the confidence falls below *possible*, the emitter deletes itself, informing its manager and any cluster to which it is associated of its deletion.

Cluster-Manager Agent

The cluster-manager agents play much the same role in the creation of cluster agents as the emitter-manager agents play in the creation of emitter agents. However, it is not possible to compute an invariant to be used for a many-to-one mapping between emitters and cluster managers. If ELINT were to employ multiple cluster-managers, any strategy for which of the many managers an emitter agent chooses to request a cluster match could still result in the creation of multiple instances of the "same" cluster (i.e., multiple cluster agents representing the same physical cluster of emitters). Thus, we have chosen to implement ELINT using only a single cluster-manager. Fortunately, new cluster creation is a relatively rare event, and the

---

[10]The algorithm simply computes the source identifier modulo the number of emitter-managers and maps that number to a particular manager.

single cluster-manager has never been observed to be a processing bottleneck.

As described above, requests from emitters to associate themselves with clusters are specified as match requests over the extant clusters. Emitters are matched to clusters on the basis of their location, speed, and heading histories. However, the cluster-manager does not itself perform this matching operation. Although it knows about the existence of each cluster it has created, it does not know about the current state of those clusters. Thus, the cluster-manager asks all of its clusters to (concurrently) perform a match.

If none of the clusters responds with a positive match, the cluster-manager creates a new cluster for the emitter. If one cluster responds positively, the emitter is added to the cluster and it is so informed of this fact. If more than one cluster responds positively, this usually indicates that there is not yet sufficient resolution of the emitter's history to uniquely associate it with a cluster. In this case the emitter to cluster matching operation is tried again after more observations of the emitter have been processed.

Cluster Agent

The radar emissions from a cluster of emitters often indicate the activities of the aircraft represented by that cluster. For example, emissions from a missile guidance radar indicate that an air-to-air attack is imminent. Each cluster agent periodically applies heuristics about types of radar signals to try to determine the current activities of its represented aircraft, and, in particular, if these activities represent a threat to friendly aircraft. This activity information, the aircraft type information, and the merged track parameters of the emitters associated with each cluster are the primary outputs of the ELINT system. Also, each cluster periodically checks to see if all constituent emitters have been deleted. If so, it deletes itself.

Time-Manager Agent

Many of the knowledge-based actions taken by an ELINT agent make use of the agent's *last-observed* time, that is, the time stamp of the most recent observation associated directly or indirectly with the agent. For example, if an emitter agent determines that it has received no new associated observations for several data time intervals (i.e., that it is "out-of-date"), it will consider itself as no longer exisiting and it will delete itself and all of its relational links from ELINT's situation board.[11]

---

[11]This action reflects the expectation knowledge that if an emitter within the area of observation is observed at time *t*, then it is expected that it will be observed at time *t*+1.

In an asynchronous message passing system such as CARE, it is difficult for an agent to determine whether it is out-of-date because it has not been observed recently or because messages to it which would result in an update of its last-observed time are delayed due to overall system load or local load imbalances. One solution to this problem would be for each observation-handler agent to send an "end-of-observation-time-interval" message to each of its known emitter agents whenever it observes the crossing of an observation time interval boundary.[12]

This solution was rejected for the reported implementation of ELINT because of a perceived excessive message overhead.[13] Instead, our ELINT experiment uses a time-manager agent. Whenever an observation-handler agent observes a new input observation time stamp, it reports this new time to the time-manager via a message. The time-manager maintains a conservative, global current observation time which is the minimum of the the reported time stamps. Whenever any agent considers taking a drastic, non-reversible action which is based on its being out-of-date (e.g., deleting itself), it requests a confirmation from the time-manager that its (the requesting agent's) last-observed time is sufficiently older than the time-manager's global current observation time. The requesting agent does not perform its considered action until it receives the confirmation. If in the interim, the requesting agent receives any messages which result in an update of its last-observed time, the confirmation is ignored.

Reporter Agent

Instances of the reporter agent class are used to asynchronously output various ELINT reports to displays and/or files, for example, threat reports and periodic situation board reports. In addition, instances of a specialization of the reporter class, **debug-trace-reporter**, are used during application program debugging to asynchronously output debugging traces in a manner that minimally impacts system timing dependencies.

---

[12]Since each input observation stream is in observation-time sequential order, each observation-handler eventually knows when such a time boundary is crossed.

[13]This overhead may be more perceived than actual. A more recent implementation of ELINT uses such "end-of-observation-time-interval" messages. Initial results seem to indicate that the associated cost is not excessive (see [16]).

4.2. ELINT Agent Organization

The ELINT agents are basically organized as a pipeline with replicated stages where each stage is an agent. Inter-pipeline dependencies and dependencies between replicated stages are managed by emitter-manager and cluster-manager agents. The amount of replication (i.e., the number of agents) at each pipeline stage is a function of that stage. For some stages, the number of replicated agents at that stage is fixed during system initialization. For example, the numbers of observation-handler agents, emitter-manager agents, and cluster-manager agents are pre-determined based on the number of collection sites and their output data rates. The numbers of emitter stages and cluster stages vary during the course of execution since the corresponding emitter agents and cluster agents are created and deleted as the radar emitters and collections of radar emitters which they represent appear and disappear over time.

The overall organization of the ELINT agents is illustrated in Figure 4-2
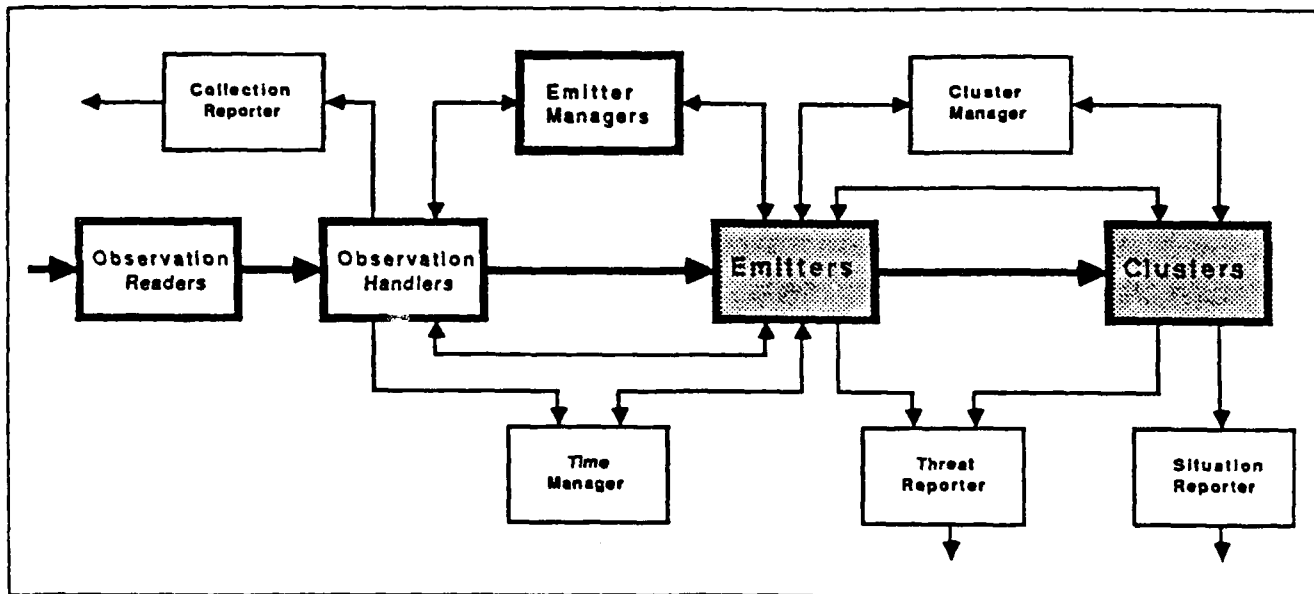
Figure 4-2: The overall ELINT agent communication organization.

5. An Overview of CARE

The CARE architectural specification and its simulation environment provide a parameterized and instrumented multiprocessor simulation testbed designed to aid research in alternative parallel architectures. The testbed executes within SIMPLE, a hierarchical, event-driven simulator [3].

A CARE architecture is a grid of tens to hundreds of processing sites interconnected via a

dedicated communications network. The network uses dynamic, buffered, cut-through routing, and it supports multicast inter-site message transmission. The ELINT experiment, for example, was performed on various square CARE grids of hexagonally connected sites, that is, each site, excluding those at the edges of the grid, is connected to six of its eight nearest neighbors.

As shown in Figure 5-1, each CARE site consists of an *evaluator*, a general-purpose processor-memory pair; an *operator*, a dedicated communications and process scheduling processor which shares memory with the evaluator; and network interfaces -- *net-inputs* and *net-outputs* -- that accomplish pipelined message transmission, flow control, deadlock avoidance, and routing. Each net-input at a site may establish a connection with a net-output at any site, and all such connections at a site may be simultaneously active.
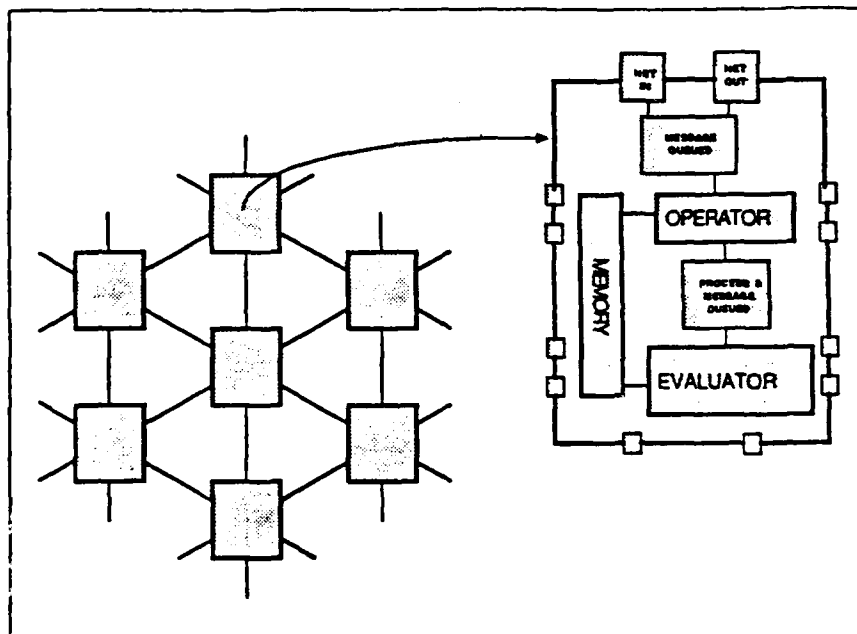
Figure 5-1:   A hexagonally connected CARE grid.

Application-level computations take place in the evaluator. The operator performs two duties. As a communications processor, it is responsible for initiating and receiving messages. As a scheduling processor, it queues application-level processes for execution in the evaluator. Message routing is performed by the net-input and net-output network interfaces.

In our simulation of CARE, the evaluator is treated as a "black box" Lisp processor. None of its internal operation is simulated. The Lisp machine hosting the simulation serves as the evaluator in each processing site. The operator, however, is functionally simulated, and the network interfaces are simulated and instrumented in great detail.

CARE allows a number of parameters of the processor grid to be adjusted. Among these parameters are: the speed of the evaluator, the speed of the communications network, the network routing algorithm, and the speeds of the process creating and switching mechanisms. By altering these parameters, a single processor grid specification can be made to simulate a wide variety of actual multiprocessor architectures. For example, we can experiment with the optimal level-of-granularity of problem decomposition by varying the speed of both process-switching and communications. Alternative network topologies can be studied by using SIMPLE's graphic interfaces and composition operators to configure CARE components into any topology that can be wired.

The CARE simulation environment provides detailed displays of such information as evaluator, operator, and communication network utilization, and process scheduling latencies. This instrumentation package informs developers of CARE applications of how efficiently their systems make use of the simulated hardware.

A more detailed description of CARE is given in [16], and the technology considerations underlying the CARE architecture are discussed in Appendix I.

6. Results and Conclusions

The CARE architectural simulation testbed and the CAOS system we have described have been fully implemented, and they are in use by several groups within our Architectures Project. CAOS-CARE executes on the Symbolics 3600 family of machines as well as on the Texas Instruments Explorer Lisp machine. ELINT, as described in Sections 2 and 4, has also been fully implemented, and we have analyzed its performance on various size CARE grids.

6.1. Evaluating CAOS

CAOS is a rather special-purpose environment, and it should be evaluated with respect to the programming of concurrent, real-time signal interpretation systems. In this section, we explore CAOS's suitability along the dimensions of expressiveness, efficiency, and scalability.

6.1.1. Expressiveness

When we ask that a language be suitably expressive, we ask that its primitives be a good match to the concepts the programmer is trying to encode. The programmer should not need to resort to low-level "hackery" to implement operations which ought to be part of the language. We believe we have succeeding in meeting this goal for CAOS (although to date, only CAOS's designers have written CAOS applications). Programming in CAOS is essentially programming

in Lisp using objects but with added features for declaring, initializing, and controlling concurrent, real-time signal interpretation applications.

6.1.2. Efficiency

CAOS has a very complicated architecture. The lifetime of a message involves numerous processing states and scheduler interventions. Much of this complexity derives from the desire to support alternate scheduling policies within an agent. The cost of this complexity is approximately one order of magnitude in processing latency. For the common settings of simulation parameters, CARE messages are exchanged in about 2 to 3 milliseconds, while CAOS messages require about 30 milliseconds. It is this cost which forces us to decompose applications coarsely, since more fine-grained decompositions would inevitably require more message traffic.

We conclude that CAOS does not make efficient use of the underlying CARE architecture. This conclusion has lead to an evolution of both CAOS and CARE which is described briefly in Section 6.3 and in detail in [16].

6.1.3. Scalability

A system which scales well is one whose performance increases commensurately with its size. Scalability is a common metric by which multiprocessor hardware architectures are judged. For example, does a 100-processor realization of a particular architecture perform ten times better than a 10-processor realization of the same architecture? Does it perform only five times better, only just as well, or does it perform even worse? In hardware systems, scalability is typically limited by various forms of contention in memories, busses, etc. The 100-processor system might be no faster than the 10-processor system because all interprocessor communications are routed through an element which is only fast enough to support ten processors.

We ask the same question of a CAOS application. Does the throughput of ELINT, for example, increase as we make more processors available to it? This question is critical for CAOS-based, real-time interpretation systems. Our only means of coping with arbitrarily high data rates is by increasing the number of processors.

We believe CAOS scales well with respect to the number of available processors. The potential limiting factors to its scaling are increased software contention, such as the inter-pipeline bottlenecks described in Section 3, and increased hardware contention, such as overloaded processors and/or communication channels. Software contention can be minimized by the

design of the application. Communications contention can be minimized by executing CAOS on top of an appropriate hardware architecture such as that afforded by CARE. CAOS applications tend to be coarsely decomposed. They are bounded by computation, rather than communication, and communications loading was not a problem in our ELINT-CAOS-CARE experiment.

Unfortunately, processor loading remains an issue. A configuration with poor load balancing in which some CARE sites are busy while others are idle does not scale well. Increased throughput is limited by contention for processing resources on overloaded sites while resources on unloaded sites go unused. The problem of automatic load balancing is not addressed by CAOS as agents are simply assigned to processing sites on a round-robin basis with no attempt to keep potentially busy agents apart. We currently have no solution to the problem of processor load balancing beyond that of carefully "hand crafting" a site allocation strategy for each application and then "tuning" that strategy via succesive refinement.

6.2. Evaluating ELINT Under CAOS

The input data set used for most of our ELINT-CAOS runs was based on a scenario involving 16 aircraft mounting a total of 88 radar emitters with between 4 and 45 emitters active and observed during any one data time interval. The scenario takes place in a 60 by 80 mile area over 36 time units, and it involves 1040 separate emitter observations.

Our experience with ELINT indicates that the primary determiner of throughput and solution quality is the strategy used in making individual agents cooperate in producing the desired interpretation. Of secondary importance is the degree to which processing load is evenly balanced over the processor grid. We now discuss the impact of these factors on ELINT's performance.

The following three "control" strategies were used in our experiment:

1. NC: This "no control" strategy represents limited inter-agent control. Agents initiate actions independently. Whenever an agent wants to perform an action, it does so as soon as processing resources are available. For example, whenever an observation-handler agent needs a new emitter agent, it simply creates it with no attempt to coordinate this creation with other observation-handlers. As a result, multiple, non-communicating copies of an emitter may be created, and each copy receives a only portion of the input data it requires. The NC strategy was expected to produce qualitatively poor results, and it was primarilly intended only as a

baseline against which to compare more realistic control strategies. What was surprising was that the strategy also produced quantitatively poor results (see below).

2. CC: In this strategy, agents cooperate in the creation of new agents via manager agents as described in Section 4. The manager agents assure that only one copy of an agent is created, irrespective of the number of simultaneous creation requests. All requestors are returned a reference to the single new agent. Originally, we believed the CC (for "creation control") strategy would be sufficient for ELINT to produce satisficing high-level interpretations. Our experiment results showed that this was not always the case (see below).

3. CT: The CT ("creation and time control") strategy was designed to additonally manage the skewed views of real-world time which develop in agent pipelines. For example, this strategy prevents an emitter agent from deleting itself when it has not received a new observation in a while even though some observation-handler agent has sent the emitter an observation which it has yet to receive. The agents corresponding to the CT strategy are those described in Section 4.

Table 6-1 illustrates the qualitative effects of the various control strategies and grid sizes. The table presents the six major performance attributes by which the quality of an ELINT run is measured. Since the input data for the ELINT experiment were generated from known scenarios, it was possible to compare the results of an ELINT run with "ground truth."

Table 6-1: ELINT Solution Quality Versus Control Strategies and Grid Sizes.

Qualitative performance attribute	Control strategy/grid size					
	NC/16	CC/16	CC/36	CT/4	CT/16	CT/36
False alarms	1%	0	0	0	0	0
Reincarnation	49%	42	2	0	0	0
Confidences	19%	20	90	89	93	95
Fixes	48%	42	99	100	100	100
Threats	65%	63	81	87	87	90
Fusion	0%	0	77	85	88	89

The major qualitative performance attributes are:

False Alarms: This attribute is the percentage of emitter agents that ELINT should not have

hypothesized as existing with respect to the total number of emitter agents hypothesized.

ELINT was not severely impacted by false alarms in any of the control configurations in which it was run as the knowledge used for hypothesizing new emitters was quite conservative. That is, the knowlege was such that it prefered missing a true, but low confidence, emitter to creating a false alarm emitter.

Reincarnation: This attribute is the percentage of recreated emitter agents, that is, emitters which had previously existed but had erroneously deleted themselves due to lack of recent observations, with respect to the total number of emitters created. Large numbers of reincarnated emitters indicate some portion of ELINT is unable to keep up with the data rate. This can be caused by the data rate being too high globally so that all emitters are overloaded or by the data rate being too h·²h locally due to poor load balancing so that some subset of the emitters are overloaded.

The **CT** control strategy was designed to prevent reincarnations. Hence, none occurred when CT was employed on any size grid. When the CC strategy was used, only the 36 site grid was large enough for ELINT to sufficently keep up with the input data rate so that emitters were not erroneously deleted due to overload.

Confidence Level: This attribute is the percentage of correctly deduced confidence levels for the existence of an emitter with respect to the total number of times such confidence levels were determined.

For each hypothesized emitter, ELINT maintains a dynamic confidence level for the existence of the emitter based on accumulating evidence (see Section 4.1). The correct calculation of confidence levels depends heavily on the system being able to cope with the incoming data rate. One way to improve confidence levels was to use a large processor grid. The other was to employ the **CT** control strategy.

Fixes: This attribute is the percentage of correctly-calculated positional fixes of emitters with respect to the total number of times fixes could have been determined from the ground truth data.

A fix can be computed whenever an emitter has seen at least two observations from different collection sites in the same data time interval. If, for example, an emitter is undergoing reincarnation, it will not accumulate enough data to regularly compute fixes. Thus, the approaches which minimized reincarnation tended to maximize the correct calculation of fix

information.

Threats: As described in Sections 2 and 4, certain emitter and cluster events represent immediate threats. This attribute is the percentage of recognized threats with respect to the total number of threat events based on the ground truth data.

Fusion: This attribute is the percentage of correct clustering of emitter agents to cluster agents. The correct computation of fusion appeared to be related, in part, to the correct computation of confidence levels. The fusion process is also the most knowledge-intensive computation in ELINT, and our imperfect results indicate the extent to which ELINT's knowledge is incomplete.

The overall goal of the control strategy experiments was to see if it was possible to determine strategies where the quality of the output results were relatively insensitive to grid size and load balance but still achived significant concurrency.

We interpret from Table 6-1 that the control strategy has the greatest impact on the quality of results. The CT strategy produced high-quality results irrespective of the number of processors used. The CC strategy, which is much more sensitive to processing delays, performed nearly as well only on the 36 si⁻. grid. We believe the added complexity of the CT strategy, while never detrimental, is primarily beneficial when the interpretation system might be overloaded by high data rates or poor load balancing.

Table 6-2 gives the simulated execution times for the ELINT runs used to derive the data in Table 6-1, and Table 6-3 gives the total CAOS message counts for these runs.

Table 6-2: Simulated ELINT execution times for various control strategies
and grid sizes.

Control strategy	Grid size		
	4	16	36
NC		>11.19 sec.	
CC		10.87	5.12
CT	11.80	8.10	4.17

Tables 6-2 and 6-3 clearly show that the processing cost of added control is far outweighed by the benefits in its use. Far less message traffic is generated, and the overall simulated time is reduced. Note that for the runs whose execution times are shown in Table 6-2, the input daṛ

Table 6-3: CAOS message counts for ELINT executions with various control strategies and grid sizes.

Control strategy	Grid size		
	4	16	36
NC		>16118 msg.	
CC		7375	4823
CT	4516	4703	4616

rate was .1 seconds per ELINT time unit. Since the input data set used for these runs spanned 36 time units, the last observation was fed into the system at 3.6 (simulated) seconds. Hence, this is the minimum possible simulated execution time for these runs.

Table 6-4 and Figure 6-1 show the quantitative effect of processor grid size when the CT control strategy is employed. These results were produced with the input data rate set ten times higher (.01 seconds per ELINT time unit) than that used to produce Table 6-2. The minimum possible simulated execution time for the runs used to produce Table 6-4 is 0.36 seconds.

Table 6-4: Simulated ELINT execution time versus grid size for production runs using CT control strategy.

Grid size	Execution time
1	9.476 sec.
4	3.237
9	1.517
16	.761
25	.541
36	.557

As shown in Figure 6-1, the speedu₊ achieved by increasing the processor grid size is nearly linear in the 1 to 25 processor site range. However, the 36 site grid was slightly slower than
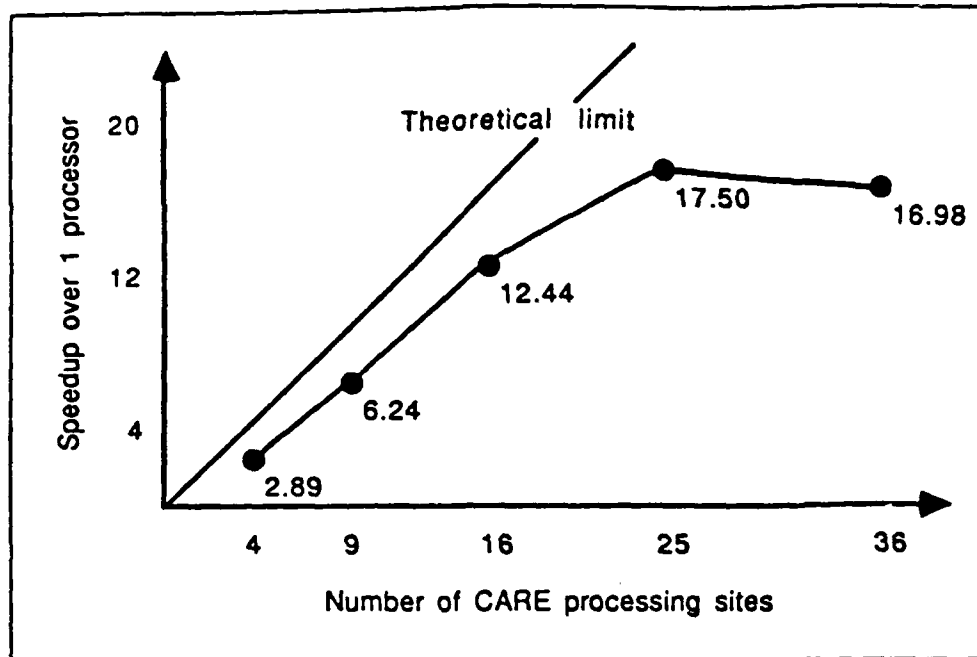
Figure 6-1: The relative speedup of ELINT executions on various size CARE grids.

the 25 site grid.[14]

In this last case, there was not sufficient data per ELINT time interval to warrant the additional processors. That is, there was not enough concurrency to exploit 36 processors. This can be seen from Table 6-5 which gives timing results for larger data sets with more emitters and observations during each time interval and, hence, more potential for concurrency.

Table 6-5: Simulated ELINT execution times and speedup for larger data sets.

Number of Observations	1-site grid execution time	36-site grid execution time	Speedup of 36 over 1
1040	9.476 sec.	.557 sec.	17.0
2080	25.10	.948	26.5
4160	55.87	2.259	24.7

As shown in this table, for an input data set representing twice as many emitters and

---

[14]Because of the intrinsic non-determinism of a CARE architecture, we observed variations in the solution qualities and the run times between different runs of the same input data set on the same size CARE grids. For such runs, the variations in solution qualities never exceeded a fraction of a percent. However, the varitions in run times where as much as five percent. This accounts for the slightly longer execution time on 36 versus 25 processors.

observations than the basic data set, the 36 site grid achived a speedup factor of 26.5 (as opposed to a speedup of 17.0 for the basic data set) over a single processor. However, for a data set four times larger than the basic data set, the speedup factor was only 24.8. This was because this larger, and hence more concurrent, data set saturated the 36 site grid. That is, the 2080 observation data set already provided enough concurrency to fully exploit the 36 site grid.

6.3. Some Open Questions

CAOS has been a suitable framework in which to construct concurrent signal interpretation systems, and we expect many of its concepts to be useful in our future computing architectures. Of principal concern to us now is increasing the efficiency with which the underlying CARE architecture is used. In addition, our experience suggests a number of questions to be explored in future research:

- What is the appropriate level of granularity at which to decompose problems for CARE-like architectures?

- What is the most efficient means to synchronize the actions of concurrent problem solvers when necessary?

- How can flexible scheduling policies be implemented without significant loss of efficiency? What is the impact on problem solving if alternate scheduling policies are not provided?

- Are there efficient mechanisms for dynamically balancing processor loads?

We have started to investigate these questions in the context of a new CARE environment. One of the primary difference between the original environment and the new environment is that the process is no longer the basic unit of computation. While the new CARE system still supports the use of processes, it emphasizes the use of *contexts* which are computations with less state than those of processes.

When a context is forced to suspend to await a value from a remote service, it is aborted, and restarted from scratch later when the value is available. This behavior encourages more fine-grained decomposition of problems written in a functional style where individual methods are small and consist of a binding phase followed by an evaluation phase.

In addition, CARE now supports arbitrary prioritization of messages delivered to streams. As

a result, it is no longer necessary to include in CAOS a complex and expensive scheduling strategy. Early indications are that the new CARE environment with a slightly modified CAOS environment performs around two orders of magnitude faster than the configuration described in this paper. The evolution of CARE and CAOS based on the results of our ELINT-CAOS -CARE experiment is described in greater detail in [16].

Acknowledgements

Our thanks to Russell Nakano, Sayuri Nishimura, James Rice and Nakul Saraiya who helped implement and maintain the CARE environment. Also, we wish to thank the staff of the Symbolic Systems Resources Group of the Knowledge Systems Laboratory for their excellent support of our computing environment. We express special graditude to Edward Feigenbaum. His continued leadership and support of the Knowledge Systems Laboratory and the Architectures Project made it possible us to do the reported research.

I. Technology Considerations Underlying the CARE Architecture

The CARE simulation testbed can be used to simulate shared memory as well as message passing multiprocessor architectures. For example, it has been configured to simulate a single address space, shared global memory architecture where the processors (and their local cache memories) are connected to the shared memory's controllers via a switching network. However, the intended focus of the CARE testbed is on message passing, multiprocessor architectures where each processor has significant local memory. This focus is based on technology considerations -- primarily communication versus processing costs.

The base for development of general purpose multiprocessor systems, as for computer systems generally, is given by the design constraints and opportunities established by evolving semiconductor design and manufacturing processes. The VLSI design medium brings a new perspective on cost -- switches are cheap while wires are expensive. Communication costs dominate those associated with logic. Communication is currently the resource in shortest supply, and it will become more of a constraint rather than less as semiconductor lithographies decrease.

The consequence of relatively expensive communication is that performance is enhanced if the design establishes that whenever a lot of information has to move in a short time, it does not have to move far. Significant locality of high bandwidth links is a design goal. Among the highest bandwidth links in a computer system are those connecting the processor and memory. Thus, close coupling of processors with local memory is preferred.

To reduce demand on the communications resource to supportable levels, local memory sizes for multiprocessors can be expected to increase to the 100K byte level and beyond, and block transfers between backing store and such several hundred kilobyte local memories will be used to make the most efficient use of both memory structures and communications facilities. Moreover, the functionallity of memory controlers will expand to include, for example, management of request queues, the dispatching of results, and execution of synchronization primitives; and thus, the distinctions between a memory controller and a small, simple processor will become blurred.

The proportion of area for a simple, high performance processor to the total area of a site with, for example, 256K bytes of local storage can be reasonably estimated at around 15%. From (i) this estimate of the incremental cost of adding a processor to a block of memory, (ii) the significant size of the total local storage in the system, (iii) the blurring of distinctions

between fast, simple processors and memory controllers of increasing complexity, and (iv) the tendency towards block tranfers between local memory and backing store, it follows that the level of the storage hierarchy now labeled as "random access memory" is likely to be subsumed by a combination of large local memories and fast, block access backing stores in multiprocessor systems.

The performance of the available communication resource merits special attention in the design of multiprocessor systems. For example, dynamic routing which selects available inter-site links as needed is useful in balancing load, and thus it allows more of the communication resource of the system to be exploited throughout a computation. Cut-though routing which makes a routing decision on the fly as a packet is received reduces buffer requirements in the system and minimizes latency experienced in network transit. Flow control via signalling transmission delays back to the source based on local blockage information together with single "word" buffering and transmission validation at each network input and output port allows the source to complete a transmission in a time that does not depend on the size of the network. Point to point multicast which sends (approximately) the same packet to multiple targets using common resources to the largest degree possible can significantly enhance overall communication performance. A communication resource with these features provides a multiprocessor system with "virtual busses" that are established precisely as and when they are needed.

These technology considerations have led us to focus our attention on the class of multiprocessor hardware system architectures exemplified by CARE.

References

1. Weinreb, D. and Moon, D. (1981) Lisp machine manual, 4th ed. Artificial Intelligence Laboratory, Massachusetts Institute of Technology.

2. Delagi, B., et al. (1986) CARE user's manual. Technical Report, Knowledge Systems Laboratory, Stanford University.

3. Saraiya, N. (1986) Simple user's manual. Technical Report, Knowledge Systems Laboratory, Stanford University.

4. Saraiya, N. (1986) AIDE: A distributed environment for design and simulation. Technical Report, Knowledge Systems Laboratory, Stanford University.

5. Williams, M., Brown, H. and Barnes, T. (1984) TRICERO design description. Technical Report ESL-NS539, ESL, Inc.

6. Aiello, N., Bock, C., Nii, H. P. and White, W. (1981) Joy of AGEing. Technical Report, Heuristic Programming Project, Stanford University.

7. Nii, H. P. (1986) Blackboard systems: The blackboard model of problem solving and the evolution of blackboard architectures. AI Magazine, vol. 7, no. 2, pages 38-53.

8. Nii, H. P. (1986) Blackboard systems part two: Blackboard application systems. AI Magazine, vol. 7, no. 3, pages 82-106.

9. Erman, L., Hayes-Roth, F., Lesser, V. and Reddy, D. R., (1980) The HEARSAY-II speech understanding system: Integrating knowledge to resolve uncertainty. ACM Computing Surveys, vol. 12, pages 213-253.

10. Nii, H. P., Feigenbaum, E., Anton, J. and Rockmore. A. (1982) Signal-to-symbol transformation: HASP/SIAP case study. AI Magazine, vol. 3, no. 3, pages 23-35.

11. Lieberman, H. (1981) A preview of Act1. Artificial Intelligence Laboratory Memo 625., Massachusetts Institute of Technology.

12. Gabriel, R. and McCarthy, J. (1984) Queue-based multiprocessing Lisp. In Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming. Austin, Texas.

13. Schoen,E. (1986) The CAOS system. Technical Report, Knowledge Systems Laboratory, Stanford University.

14. Halstead, R. H., Jr. (1984) MultiLisp: Lisp on a multiprocessor. In Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming. Austin, Texas.

15. Denelcor, Inc. (1981) Heterogeneous element processor: Principles of operation. Boulder, Colorado.

16. Delagi, B., et al. (1986) Lamina: Streams and objects for concurrency. Technical Report, Knowledge Systems Laboratory, Stanford University.

User-Directed Control of Parallelism; The CAGE System

by

Nelleke Aiello

Knowledge Systems Laboratory
Computer Science Department
Stanford University
Stanford, California 94305

Table of Contents

Abstract

CAGE provides a framework for building and executing application programs as concurrent blackboard systems. The user controls which constructs of the blackboard system are executed in parallel.

1. Introduction

CAGE[1], Concurrent AGE[2], provides a framework for building and executing application programs as a concurrent blackboard system. With CAGE, the user can control which parts of the blackboard system are executed in parallel. A blackboard application an be implemented and debugged serially on CAGE. Once the serial version is debugged, concurrency can be introduced to different parts of the system, allowing the user to experiment with various configurations. We believe this incremental approach will facilitate the construction of concurrent problem solving systems and will teach us much about programming in a parallel environment. This paper describes the design of the CAGE system and gives detailed instructions for implementing an application, using the CAGE language and compiler [Rice 86]. We have included advice, warnings, and caveats based on our experience using CAGE.

The target parallel system architecture for the CAGE system is currently the same as that of QLAMBDA, a queue-based multi-processing Lisp ([Gabriel 84]and McCarthy) on which the parallel simulation is based. We are assuming a shared memory and a large number of processors. The user can specify his CAGE application in an extension of the L100 language, called the CAGE language, and use the CAGE compiler to generate CAGE code. CAGE runs on LOQS, a functional simulator for QLAMBDA. CAGE is implemented in ZETALISP for Symbolics 3600 machines and TI Explorers.

2. Overview of CAGE Design

CAGE is a blackboard framework system. In addition to the basic AGE [Nii 79] functionality, CAGE allows user-directed control over the concurrent execution of many of its contructs. The basic components of a system built using CAGE are:

1. A global data base (the blackboard) in which emerging solutions are posted. The elements on the blackboard are organized into levels and represented as a set of attribute-value pairs (a frame).

2. Globally accessible lists on which control information is posted (e.g. lists of events, expectations, etc.).

3. An indefinite number of knowledge sources, each consisting of an indefinite number of production rules.

4. Various kinds of control information that determine (a) which blackboard element is to be the focus of attention and (b) which knowledge source is to be used at any given point in the problem solving process.

5. Declarations that specify what components (knowledge sources, rules, condition and action parts of rules) are to be executed in parallel, and when to force synchronization. During the execution of the user's application CAGE will run these specified components in parallel.

Using the concurrency control specifications, the user can alter the simple, serial control loop of CAGE by introducing concurrent actions. CAGE allows parallelism ranging from concurrently executing knowledge sources all the way down to concurrent actions on the right- or left-hand-sides of the rules. The serial execution and parallel executions possible in CAGE are summarized below.

in KS Control
    serial: pick one event and execute associated KSs

---

parallel:
1. as each event is generated execute associated KSs in parallel[3]
2. wait until several events are generated then select a subset and execute relevant KSs for all subset events in parallel

in KS
serial: 1. evaluate bindings
2. evaluate LHS then execute RHS of one rule whose LHS matches (in written order)
3. evaluate all LHS then execute all RHS whose LHSs match

parallel:
1. evaluate bindings*
2. evaluate all LHSs in parallel
   a. then synchronize (i.e. wait for all LHS evaluations to complete) and choose one RHS(pick one in order)
   b. then synchronize and execute the RHSs serially (in written order)
   c. execute RHS as LHS matches*

in Rule
serial: evaluate each clause then execute each action

parallel:
evaluate clauses in parallel then execute actions in parallel*
(first nil clause --> no match; first all non-NIL clauses --> match)

in clause
serial: Lisp code

parallel: Qlambda code

For more information about the concurrent options available in the CAGE System and how to specify them refer to Section IV of this paper.

3. Building applications in CAGE

In each of the following sections we will outline the application data that must be supplied by the user and how that information should be structured for use by the CAGE System. The CAGE System provides a CAGE language with which the user can write his application. The type of user-supplied information is similar to that required for applications constructed in the original AGE system. However, the structure of the user information is somewhat different from that of an AGE application.

---

[3]The starred options indicate the greatest use of concurrency.

3.1. Blackboard Data Structure

There are two major components in the CAGE blackboard structure, the hypothesis *classes* (frequently called levels in hierarchical blackboard structures) and the hypothesis *nodes*. The user must specify the classes that make up his application's blackboard structure. For each class, the user must define the fields to be associated with the nodes created in that class. Nodes are created in those classes, either a priori by the user or dynamically while executing the user's rules. The following example shows the definition of several classes and their fields in the CAGE language.

```
Class Definitions for Model "example" :

Class name-of-levela :
    attribute1
    attribute2
    attribute3
    ...

Class name-of-levelb :
    attribute4
    attribute5
    ...
```

This will compile into two macro calls, DEFHYPOTHESIS-STRUCTURE and DEFLEVEL, which the CAGE System will in turn compile into the appropriate hypothesis structure.

```
(defhypothesis-structure
  user-hypothesis-structure
  (application-system-root)
  name-of-levela
  name-of-levelb
  name-of-levelc
            ...)

(deflevel name-of-levela
  ((attribute1 nil)
   (attribute2 nil)
   (attribute3 nil)
            ...))
```

Each of the levels(or classes) will be defined as an object with the attributes as instance variables and with the nodes as instances of those objects as they are created. (The user can define methods for the level objects which are generally used for printing information contained in the nodes on those levels.)

Definitions:

user-hypothesis-structure: A name the user gives the application's blackboard structure.

application-system-root: A handle on the above hypothesis structure for user access, generally a node where the input data, or a massaged version of the input data will reside, or the top level of a hierarchical hypothesis structure.

name-of-level: Each level or class must have a user supplied name.

node: An instance of a level, created either before or during the execution of the application, inheriting all the attributes of that level, but no values.

attribute: For each level the user must specify the names of the slots, which will become a template for the instance nodes, which in turn will contain the values used by the KSs. These values are initially NIL.

link: The user may also define links for connecting nodes. These links are defined

in the knowledge sources which use them and consist of a link name and an optional, opposite link. The value of a link on a node is the name of another node.

value: The value of an attribute depends on what was stored there by the rules and its structure depends on how it was stored. Values can be modified only by the user's initialization function and by the application rules. The structure of the values is arbitrary. How values are added or changed is explained in the knowledge source section.

3.2. Control Structure

All CAGE control information is referenced through the Control-Structure object. The major components of the Control-Structure are:

User-Initialization: This is a user-defined function, handling any initialization needed for the user's program, e.g. setting-up the appropriate blackboard structure (on top of the predefined hypothesis framework) from the input data.

Termination-Condition: Another user-defined function, which determines when the application should be terminated. The Termination-Condition can access the step-lists for events or expectations, perhaps checking for a significant event; or the blackboard, checking a particular node or nodes. It should return a non-nil value when the application is to be terminated.

User-Post-Processor: When the termination condition is true, a user supplied post processing function is invoked. This function can be used to print out the application's results in a readable form, or to handle any other post processing details.

Event-Info: This is a pointer to the Event-Information object which contains both the user-specified information on how events should be scheduled, and run-time data including the event list and the current focus event.

Expect-Info: Similar to the Event-Info pointer, this object keeps track of the expectations generated by the application and information specifying how those expectation should be scheduled.

Control-Rules: A list of of control rules defined by the user to determine when to execute which control step (event or expectation). The control rules are defined using the DEFCONTROL-RULE macro. Each control rule consists of a condition, an arbitrary LISP expression and a steptype, either event or expect. The following example of a control rule says that if there are any events pending on the event list (steplist of event-info is not null), then do an event next.

```
Example:

Control Rule : Crule-1
    Condition Part:
        If   : event-info⊕steplist
    Action part : event
```

LHS-Evaluator: The default function for evaluating the conditions of a rule if the knowledge source containing that rule has no left hand side evaluator over-riding this default. For most applications the CAGE provided function QAND will suffice. It is a serial or concurrent boolean AND depending on the parallel options selected by the user.

3.2.1. Event-Information

A blackboard system can be executed in several ways, the simplest being event-driven. This means that each time a rule action is executed the system records that change to the blackboard

as an event. Each event is added to a list called the *event list*. The scheduler selects an event from the event list to become the next *focus event*. The type of focus event is matched against the preconditions of the knowledge sources, and all the matching knowledge sources are activated. The rules of the activated knowledge sources are evaluated, those rules with satisfied conditions are executed and the cycle repeats until the termination is true.

To run a blackboard model with an event-driven control structure, certain control information must be supplied by the user.

> **selection-method**: a function that determines which event to select from the event list. The user can write his own *best-first* selection method or use one of the CAGE provided functions, FIFO, LIFO, or AGENDA. If the AGENDA selection method is chosen, the user must also specify the agenda and an order.

> **agenda**: An ordered list of event types supplied by the user. (See knowledge source specification for definition of event type.)

> **order**: LIFO or FIFO order in which to check the agenda. There may be several different events of the same type on the event list.

> **collection rules**: In some applications many events of the same type and the same node are generated and added to the event list. If the user specifies that type of event as a collection rule, then only one event is pursued and the others are *collected* and deleted from the event list.

3.2.2. Expect-Information

In an expectation-driven system, a rule may specify an expected result or change on the blackboard as one of the actions of that rule (called an expectation rule). When an expectation rule is executed, the expectation part of the rule is added to the *expectation list*. Later, when the control rules specify that an "expect" step should be executed, a focus is selected from the expectation list. If a change has occurred on the blackboard that satisfies the expect portion, actions associated with the expectation rule are executed.

Much of the information required to execute an expectation-driven system is similar to that of an event-driven system. The user must supply a selection-method, possibly including an agenda and order, and collection rules. Some additional information is required to execute expectation.

> **matcher**: a function which defines how to match expectations to the blackboard. CAGE provides on default, PASSIVEMATCH, which simply evaluates the expectation portion of the expectation rule to see if its value is non-nil.

3.3. Knowledge Sources

CAGE knowledge sources are a partitioning of the application knowledge into sets of rules. Each knowledge source consists of some declarative information and a set of rules.

3.3.1. Knowledge Source Declarations

The definition of a knowledge source consists of more than just groups of rules. In order to properly interprets those rules, CAGE needs to know certain knowledge source control information, e.g.,

1. Under what circumstances should this knowledge source be invoked?

2. How should the rule conditions be evaluated,

3. what levels of the blackboard structure will be changed?

4. Which one or all of the rules whose conditions are true should be executed?

5. Are there any local variables or links to be defined for this KS?

The following features are available for the user to tailor a knowledge source to his own specifications:

:

Preconditions: A list of tokens, representing the *event types* used in rules. If the focus event has an event type that matches one of the knowledge source's preconditions, then that knowledge source is activated.

Levels: A list of pairs of blackboard levels or classes. The user must specify between which levels of his hypothesis structure a knowledge source makes inferences.

Links: If a knowledge source adds links between nodes on the blackboard, they must be defined here. The definition consists of a list of pairs of link names, a link and its inverse.

Hit Strategy: There are two main hit strategies available in CAGE, SINGLE and MULTIPLE. When a knowledge source with a single hit strategy is interpreted the rules of that KS are evaluated, in order, until one rule's condition evaluated to true. Then that rules actions are executed and no other rules are even considered. With a multiple hit strategy, the conditions of all rules of a knowledge source are evaluated and then all the actions of rules which successfully evaluated executed. In conjunction with either single or multiple hit strategies, the user can also specify ONCEONLY. This will cause a rule to be marked when its conditions are successfully evaluated. Its actions will be executed and it will never be evaluated again during that run of the application.

Definitions: A list of local definitions, available to all the rules of a knowledge source. The definitions are an efficiency feature to avoid the repeated calculation of the same value by all the rules. The structure is similar to that of LET, a list of pairs, a variable name and an expressions to be evaluated and assigned to the the variable. If the value is NIL it can be omitted.

Rule Order: A list of rule names, representing the rules of the knowledge source. This is the order in which the rules will be evaluated serially. Because the rules are actually defined as methods of the knowledge source to which they belong, each name should begin with a colon (:).

LHS Evaluator: The user can optionally specify a left hand side rule evaluation function for each knowledge source. There is also a default LHS evaluator specified for the entire application in the Control data. The evaluator specified here will override the default evaluator for this specific knowledge source. The LHS evaluator is a function which determines how the rule conditions are evaluated. CAGE provides several built-in functions which the user can select, including AND, for a simple boolean AND of the conditions and QAND for a concurrent boolean AND.

The following is an example of the definition of a knowledge source from the CRYPTO system written in the CAGE language.[4] The name of this knowledge source is "combine-weights", it has two preconditions, makes inferences from the Cryptoletter level of the hypothesis structure to the alphabet-letter level, defines a pair of bi-directional links, and uses the single-hit rule selection strategy. The combine-weights knowledge source also makes two definitions, possible-values gets the value NIL and lhs-evaluator the value QAND.

---

[4]The colons in the CAGE language are separators when separated by spaces from other words in the language. Colons indicate keywords when they directly precede a word.

```
Knowledge Source : combine-weights
   Preconditions : Confirmation, Contradiction
   Classes : Cryptoletter : alphabet-letter
   Links : Possible-Value-of : possible-Letters
   Rule Selection : Single

Definitions :
    possible-values ≡ nil
    lhs-evaluator ≡ qand
```

This compiles to the following CAGE macros.

```
(defknowledge-source COMBINE-WEIGHTS
   :preconditions (confirmation contradiction)
   :levels ((cryptoletter alphabet-letter))
   :links((possible-value-of possible-letters))
   :hit strategy (single)
   :bindings ((possible-values))
   :rule-order (:letters )
   :lhs-evaluator qand)
```

3.3.2. Rules

CAGE rules consist of three major parts; definitions, conditions, and actions. Here is an example from CRYPTO in CAGE.

```
Rule :  letters {3}

Definitions :
    possible-values ≡
        possible-values(focus-node⊕
                            possible-letters)

Condition Part :
    If      : qand(focus-node·is-cryptoletter,
                   possible-values)

Action Part :
    Changes :
            Change Type    :  Update
            Updated Node   :  focus-node
            Event Type     :  possible-assignment
            Updated Slots :
                possible-letters ←← possible-values


;Combine the weights of identical possible
;values.
```

CAGE also provides a macro for defining rules called DEFRULE, to which the above will compile.

```
(defrule (combine-weights :letters)
  ((possible-values
     (possible-values
       ($value focus-node :possible-letters
                 :all)))) 
  ((is-cryptoletter focus-node)
    possible-values )
  ((propose :EVENT-TYPE 'possible-assignment
            :CHANGE-TYPE 'supersede
            :HYPOTHESIS-ELEMENT focus-node
            :LINK-NODE nil
            :ATTRIBUTES-AND-VALUES
              '((possible-letters
                   ,possible-values))
            :SUPPORT 'combine-weights)
  ))
```

After specifying the knowledge source to which a rule should be added and the name of the rule, preceded by a colon, the user must specify the three major parts of the rule.

Definitions: The definition part of a rule is similar to a LET in structure. The local variables set here are available only to this rule, both in the condition and action parts, as well as other definitions of this rule. This is an optional component of a rule, and can be NIL.

Conditions: The second part of a rule contains the conditions. These can be one or more arbitrary LISP expressions which will be evaluated according to the left hand side evaluator as specified in the local knowledge source or at the control level. The conditions can reference both local variable definitions or variables bound at the knowledge source level. The CAGE system provides several access functions for retrieving values from the hypothesis structure, which can be used in the conditions of rules. It is important when writing the conditions of rules for a CAGE application to keep in mind the feasibility of running those clauses concurrently, i.e. keeping them independent of each other.

Actions: The action clauses make up the final part of a CAGE rule. These clauses have a very specific structure as evidenced by the preceding examples. The actions specify what changes are to be made to the hypothesis structure by a rule and how those changes should be made. The user must specify what node and attributes on the blackboard are to be changed, what the new links or values are, and how those changes are to be made (possibly deleting some old values). The user must also specify an event type, a name representing the type of change this action makes to the blackboard. If and when the event created by this action is selected as a focus event, this token will be matched against the preconditions of the knowledge sources to determine which KS to invoke next.

3.4. Initialization
There are two types of initialization which can occur at the beginning of a CAGE run. First CAGE must create the instances of all the application defined flavors which will constitute the executable form of the user's system. In addition, the user can do any other initialization he feels appropriate by defining his own initialization function, the name of which should be stored in the application's control structure. Since the major components of the application are defined as flavors, initialization can be done by defining :initialize or :after :init methods.

3.5. Input Data
The user must define two functions to handle his input data.

1. INPUT-PROCEDURE(Record, Time) : Given an input record, retrieved automatically at the correct time by CAGE, do what ever should be done with that input,e.g. add it to the blackboard.

2. TIME-OF-INPUT-RECORD(Record) : Given an input record, return the time stamp.

At the beginning of each run the user will be asked to specify an input data file by typing in the file name or selecting a file from a menu of pre-specified input data file names. The data file consists of records that can be read by the above two functions. A time stamp is mandatory on each input record.

4. Specifying Concurrency

CAGE supports the concurrent evaluation of pieces of knowledge. Once an application has been debugged in serial mode, the user can specify one or several knowledge source components to be executed in parallel. For example, the user might specify that the rules of the knowledge source be evaluated concurrently, or perhaps just the actions of the rules or a combination of the available options. With a minimum amount of recompilation, the user can change his parallel specifications and experiment with many different configurations.

In general more speed-up should occur as more components are run in parallel. But for some applications the overhead of setting up the new processes and inter-process communication costs will be greater than the speed-up gained by executing particular components concurrently. For example, if most or all of the knowledge sources of an application contain only one rule, then it would not be efficient to evaluate rules in parallel since for any one KS invocation there would only be one item to evaluate.

4.1. Concurrent Components

The use of knowledge sources to partition the knowledge in blackboard systems and, in particular, the structure of the knowledge sources in CAGE provide several obvious places for concurrency. The knowledge sources group the domain knowledge into independent modules, which theoretically, could be invoked independently and concurrently. Within each knowledge source the rules provide another source of parallelism, and within each rule, the clauses of the condition and action parts provide yet another. Of course not all clauses, rules or even knowledge sources are actually implemented totally independently of each other and some serialization may be necessary to correctly solve the application problem.

The following are the options for parallelism available in CAGE, grouped according to their allowed use in combination.

Clause level: can be used in combination with each other or any other parallel option.

actions: Execute the RHS action clauses of a rule in parallel. Note: When running RHS actions concurrently a non-deterministic system may result if both destructive (Supersede in CAGE) and constructive (Modify) actions occur to the same object in parallel. (Same object and attribute) A QLOOP macro is used to initiate the parallelism for loop actions, requiring recompilation of the rules containing loop actions.

lhs: Evaluate the LHS condition clauses of a rule in parallel. Note: Use the rule bindings to set any local variables tested here, insuring that the lhs clauses will be independent. A QAND macro is provided as the LHS-evaluator to initiate the concurrency for the conditions, requiring recompilation when this option is used.

rule-bindings: Evaluate the definitions of a rule in parallel. Again, these definitions should be independent of each other if their concurrent evaluation is to result in an actual speed-up.

Rule level: bindings can be used in combination with any of the other options, but only one of the rule options, single, multiple, sync or nosync can be used at a time.

> **bindings**: Concurrently evaluate the definitions at the beginning of a knowledge source.

> **rules-single**: Evaluate all of the conditions of the rules of a knowledge source concurrently, but only execute the actions of one successfully evaluated rule.

> **rules-multiple**: Evaluate all of the conditions of the rules of a knowledge source concurrently, then serially execute the actions of all the successfully evaluated rules.

> **rules-sync**: Evaluate all of the conditions of the rules of a knowledge source concurrently, then concurrently execute the actions of all applicable rules.

> **rules-nosync**: Begin evaluating the conditions of the rules of a knowledge source in parallel and execute the actions of each rules as soon as the conditions are known to be true. With this option there is no synchronization between the left and right hand sides of rules.

Knowledge source level: Only one of the knowledge source options can be set at any one time.

> **kss**: Invoke all the applicable knowledge sources concurrently at step selection, synchronizing by waiting for all knowledge sources to complete execution and add events to the event list before concurrently invoking a new set of kss.

> **kss-nosync**: Invoke all applicable knowledge sources as soon as a new event is created. This option provides the least control of all the options available and does no synchronization. Many applications will have to be changed slightly to execute reasonably under these conditions, particularly removing any possible circular knowledge source invocations. To implement the parallel execution of knowledge sources without any synchronization, the control loop of CAGE was drastically altered from that described at the beginning of this paper. (See CAGE Overview.) Without any synchronization, as soon as an event is created it immediately allows all relevant knowledge sources to be invoked. No events are added to the eventlist and no focus event is ever selected. A timed loop was added to the top level control to re-invoke the user's initial knowledge source in case the system exhausts all previous events before the termination condition is satisfied.

> **kss-minisync**: Add an event to the event list and do minimal computation at the point of synchronization before invoking the next set of knowledge sources. The main computation done is the collection and pruning of similar events, leaving fewer events to activate subsequent KSs. The mini-sync and no-sync options are different from the parallel kss option in that they don't use the serial step-selection procedure.

4.2. How to specify and change parallel components

A function, SELECT-PARALLEL-OPTIONS is provided to allow the user to quickly change the selected parallel options. SELECT-PARALLEL-OPTIONS has no arguments. A menu of parallel options will pop-up on the screen and the user can select new options or delete old ones.

5. Design Details

CAGE is currently implemented in an object-oriented style, using the Flavors feature of ZETALISP. The top level object in CAGE is called the BLACKBOARD. From the Blackboard object there are pointers to each of the principle components of the system, as follows

control-structure: all control information specified before compilation is stored here, as well as pointers to run-time control structures.

hypothesis-structure: the blackboard solution space, which must be structured by the user.

knowledge-source-list: names of the knowledge sources containing the production rules of the user's application.

user-functions: optional, user-defined functions invoked by the rules

information-structure: optional, user-defined, static data structures

A separate data structure, Parallel-Specifications, is used to store the parallel options selected by the user.

The DEFKNOWLEDGESOURCE macros will create, at compile time, an object for each knowledge source, and a set of associated methods. During the initialization process an instance of each knowledge source object is created. Other instances may be created during system execution if one of the concurrent knowledge source options is selected. One of the associated methods, SETUP-AND-START, evaluates the knowledge source definitions and initiates the rule interpretation when a knowledge source is invoked.

Each rule is created as three methods, EVALUATE-DEFINITIONS, EVALUATE-CONDITION, and EVALUATE-ACTION, associated with the rule's name using the :case method-combination feature of Flavors. The keywords of the action clause listed above are keywords in the method definitions, and therefore must be preceded by colons in the macro definition of a rule.

CAGE utilizes a global variable, PARALLEL-SPECIFICATIONS, whose value is a list of the current parallel options specified by the user. It is initially NIL and is updated using SELECT-PARALLEL-OPTIONS.

During execution CAGE prints out messages indicating the state of the execution and uses some simple graphics to help the user observe the simulation of concurrency. A set of small windows will appear on the right side of the screen, one for each process initiated by CAGE. Any state messages generated by the parallel process will appear in one of these associated windows, instead of the main terminal i/o window. There is only room to display 12 of these small i/o windows at the same time and still have them large enough and leave them up long enough to be readable. If more than 12 processes are active at the same time, the windows will overlap.

6. Future Directions

The next step for CAGE will be a reimplementation on CARE. The instrumentation in CARE will provide us with the needed tools for measuring the speed-up gained from each of the various concurrent options in the CAGE System. CAGE users will be able to implement and debug their applications in the current CAGE-on-LOQS system with its fast simulation time. Once an application is debugged it could then be run on the CAGE-CARE system for complete and accurate measurements.

References

[Gabriel 84]   Gabriel, Richard P. and McCarthy, John.
Queue-based Multi-processing Lisp.
Proceedings of the ACM Symposium on Lisp and Functional programming :25
- 44, August, 1984.

[Nii 79]   Nii, H. P. and N. Aiello.
AGE: A Knowledge-based Program for Building Knowledge-based Programs.
Proc. of IJCAI 6 :645 - 655, 1979.

[Rice 86]   Rice, J. P.
The L100 Language and Compiler Manual.
Technical Report KSL-86-21, Heuristic Programming Project, C. S. Dept.,
Stanford University, 1986.

Multi-System Report Integration Using Blackboards

by

J. R. Delaney

Knowledge Systems Laboratory
Computer Science Department
Stanford University
Stanford, California 94305

Table of Contents

List of Figures

ACKNOWLEDGEMENT

ABSTRACT

Blackboards are an AI problem solving methodology. A blackboard system consists of a structured data base (the blackboard) holding input and derived inferences and a collection of procedures for deriving inferences (knowledge sources). Each knowledge source is specialized to operate on some portion of the blackboard. The knowledge sources are invoked opportunistically as the information on the blackboard increases.

ne best known applications of the blackboard methodology have been in speech understanding and passive sonar data interpretation. The inputs in these cases were a single form of raw sensor data. But the methodology is also well suited to integrating multiple streams of fully reduced and qualitatively different data such as active radar track reports, passive electronic intelligence reports, and human intelligence reports about enemy intentions.

This paper sketches the nature of the blackboard problem solving methodology with an emphasis on those features suiting it to such applications. The sketch is illustrated with examples from a relatively simple multi-system report integration problem. Relevant applications currently under development at Stanford's Knowledge Systems Laboratory are also described.

1. INTRODUCTION

"Multi-System Report Integration" is an odd phrase. An alternative would have been "Sensor Data Fusion". But that phrase often implies a less reduced form of information to integrate than is intended here. The reporting systems in this paper are presumed to reduce the data they sense as fully as is practical with only that data available. The degree of processing can vary from system to system. For a radar tracking system, the reports would be samples of on-going tracks integrating all measurements up to the present. For an ELINT system dealing with intermittent emissions, the reports might be just current emitter and bearing characteristics. And for a human intelligence gathering system, the reports might be informed guesses about near-term enemy intentions.

"Sensor Data Fusion" also usually implies that the information to be integrated appears at comparable time intervals or is static. But the reporting systems in this paper are presumed to provide reduced data over a wide range of time intervals. The radar, ELINT, and "humint" systems mentioned above could produce reports at very different intervals with very different degrees of regularity. Assuming that some reports are locally of comparable frequency while others are locally static information is Procrustean.

"Blackboards" refers to a particular AI problem solving methodology. The best known applications of the blackboard methodology are HEARSAY-II, a speech understanding system (2), and the HASP/SIAP sonar data interpretation system (4,5). These applications effectively processed regular streams of data from a single sensor, treating any other information as locally static. But the blackboard methodology is more generally applicable. In particular, it provides a convenient framework for integrating maximally reduced information from multiple sources with different temporal characteristics. Just what is needed for multi-system report integration.

In the first section below, the fundamental features of blackboard systems are described abstractly. A consistent set of examples are used in the following section to clarify those features in context of multi-system report integration. The next section reviews those aspects of the blackboard methodology particularly suited to multi-system report integration. The last section briefly describes work in progress at Stanford's Knowledge System Laboratory on two more ambitious examples. It also explains how that work is embedded in a larger effort.

2. NATURE OF BLACKBOARDS

The blackboard problem solving methodology originated approximately 10 years ago and has been evolving ever since. The hallmarks of a blackboard system are:

- A global data store holding input data and hypotheses about the solution of the problem derived from that data. Related information is kept together. This data store is known as the blackboard.

- A collection of procedures for deriving hypotheses about the solution of the problem from the input data and/or from other hypotheses. Each procedure is specialized to operate on a particular portion of the blackboard. These procedures are known as knowledge sources.

- A mechanism for invoking a knowledge source on relevant parts of the blackboard. A knowledge source is invoked on a particular piece of the blackboard when the invocation would incrementally advance the solution of the problem. This mechanism is known as the control structure.

Each of these hallmarks is described abstractly in the remainder of this section with simple examples appearing in the next.

The blackboard holds the state of the problem solving system as the solution evolves. In conventional terms, the dimensionality of the state varies with time. The elements may be discretely or continuously valued. And the elements change values at discrete times. But such observations miss the most significant feature of the blackboard. It structures the information it holds.

Closely related input data or hypotheses are collected together in the form of blackboard nodes having certain attributes and values for those attributes. Related nodes form blackboard levels. All the nodes in a given level having the same attributes but (potentially) different attribute values. Levels can in turn form hierarchies of analysis or abstraction, usually with input data nodes at the base of each hierarchy. The most common nodal attributes are links between nodes on different levels. Such links connect hypotheses to input data or other hypotheses which support them. They can be links up and down levels within a hierarchy or they can be across hierarchies.

Knowledge sources transform the state of the problem solving system by adding nodes to the blackboard, by removing them, or by modifying their attribute values. Knowledge sources are effectively parametric procedures for transforming the state. A knowledge source could be invoked on any node at a given level or a tuple of nodes at one or more levels. It operates only on the node(s) upon which it is invoked plus those nodes linked directly or indirectly to them. Knowledge sources are also effectively typed procedures; a knowledge source can be invoked only on a node of a particular level or on a tuple of nodes, each of a particular level. This feature of knowledge sources provides them with a degree of modularity. In particular, knowledge sources do not interact directly.

The procedure carried out by a knowledge source expresses knowledge of how to advance the problem solution. It is expressed in the creation, modification, and/or elimination of particular sorts of hypotheses in the form of nodes of particular levels. In this sense, a knowledge source is a specialist in the solution of some part of the overall problem. The details of the procedure can be expressed in any form. A typical form is a set of production rules and a policy for using them.

Each production rule specifies a logical condition on the attribute values of the node(s) upon which the knowledge source is invoked and an action to be carried out if that condition is true. Both the condition and action can be compound. The value of a compound condition is TRUE if the values of all its component conditions have TRUE values. A compound action is simply a sequence of individual nodal creations, deletions, or modifications. Evaluating a logical condition or modifying a node may require the application of complex numeric functions to attribute values. In this way, production rules mix symbolic and numeric computations.

Different policies for using a set of production rules allow at most one action to occur, or

multiple actions but never the same one twice, or the same one repeatedly. In the first case, the rules are scanned in order of definition with the scan terminating immediately if a rule's action is carried out. In the second case, the logical conditions of the rules are all tested before any actions take place. Then any actions are carried out in parallel. The third case is simply the second case repeated until no logical condition is TRUE. While this style of programming many seem bizarre at first, it has proved quite successful in past and existing blackboard systems.

A knowledge source describes the procedure by which it changes the blackboard when invoked. It also describes when it is invocable. The most general form of this description is a (possibly compound) logical condition on attribute values of the node(s) upon which it could be invoked. In this manner, a knowledge source resembles a production rule. The condition is parametric in the same sense that each knowledge source is parametric. As a result, the same knowledge source may be invocable on several nodes or tuples of nodes simultaneously. Each such combination of a knowledge source and a node or tuple of nodes is called a potential invocation. At any time, there are typically many potential invocations. The control structure determines the set of potential invocations, picks one, and causes it to be carried out.

Many blackboard systems do not use the most general form to describe when a knowledge source is invocable. They use events and logical combinations thereof. An event is a summary of a blackboard change. A knowledge source posts the appropriate event or events when it completes. A pointer to the affected node is associated with each event. These systems may also use events for an additional purpose as explained below.

The control structure is intended to operate in an opportunistic manner analogous to the manner in which people solve jigsaw puzzles. Initially, the puzzle solver scans for pieces with singular small-scale characteristics. If two such pieces have similar characteristics, they are tested for fit. Gradually, clusters of pieces accrete as the puzzle solver continues to scan through the unused pieces. Once the clusters become sufficiently large, scanning the pieces is replaced by searches for specific pieces to extend a cluster. But pieces plausibly belonging another cluster are tested for fit there if they are chanced upon during a search. Eventually, large clusters are recognized as connected on the basis of large scale characteristics and are jointed. If progress while searching for specific pieces bogs down, the puzzle solver reverts to scanning for pieces with similar characteristics for a time. It choses that activity which, at the moment, seems likely to make the best contribution to the overall solution of the problem.

A variety of techniques are used by the control structures of different blackboard systems to decide which potential invocation would, if carried out, make the best contribution to the overall solution. The topic is being actively researched. One system has an additional blackboard for handling hypotheses about the best choice (3) and another allows all potential invocations to be carried out in parallel (6).

Several blackboard systems use events in their control structures. After a particular event or sequence of events, particular knowledge sources are preferred to others. And they are prefered for invocation on the affected node or nodes. These same systems also use events to describe when a knowledge source is invocable. So the control structures of these systems need only attend to events and not to the blackboard nodes themselves.

Some of these blackboard systems also use expectations in their control structures. Expectations are posted by knowledge sources just as events are posted. Generally speaking, they are instructions to invoke a particular knowledge source on a particular node or nodes when, if ever, a certain event or pattern of events occurs involving the node(s). Expectations can also be negative. Such expectations cause a particular knowledge source to be invoked if a certain event or pattern of events does not occur within a specified time interval.

3. BLACKBOARDS ILLUSTRATED

Consider the problem of producing a situation map of aircraft flying over an area of interest. The situation map is based on track reports from an air surveillance radar tracking system, emitter/bearing reports from an ELINT system sensing airborne radar emissions, and warnings from a h man intelligence system. The warnings are that particular aircraft or grouos

of aircraft may soon enter the area of interest with particular objectives in mind. The situation map should identify the type of each aircraft as well as its current position and velocity. The radar track reports are regular for aircraft in the area of interest. The ELINT reports are intermittent by comparison. There are no reports unless an emitter is on. And the detection range of an active emitter can depend on its type and, in some cases, on the aircraft's aspect. ELINT reports are also less accurate geometrically than radar reports. Intelligence reports are generally less frequent than the ELINT reports, but can be updated rapidly on occasion.
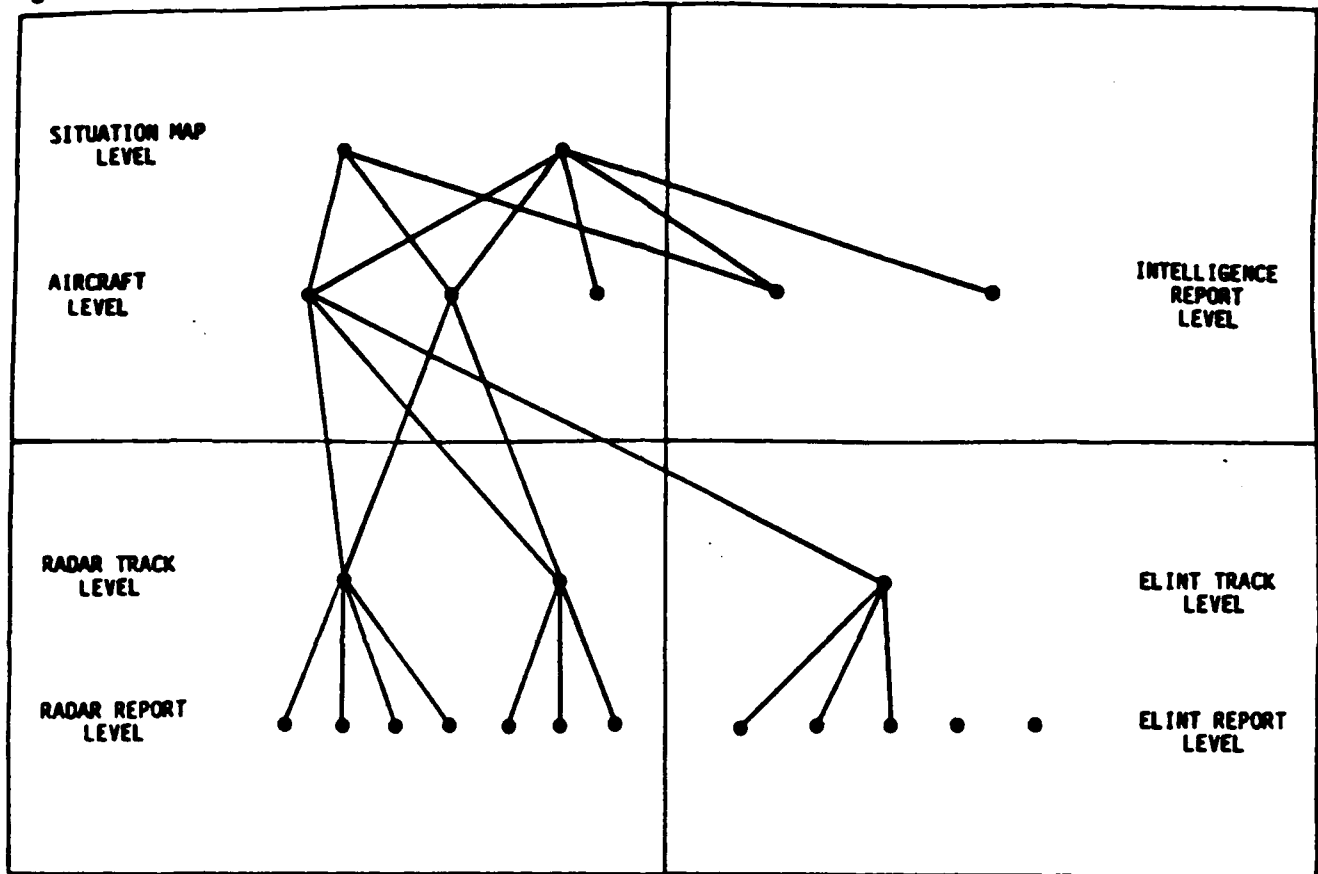
Figure 3-1: A Blackboard with 7 levels of nodes in 4 hierarchies

Figure 3-1 illustrates a possible blackboard configuration during the course of solving this problem. There are seven levels on the blackboard, a typical number. The situation map and aircraft levels form one hierarchy of levels. Nodes on these two levels hierarchically express a'ternative hypotheses about the map of aircraft in the area of interest. Two situation map hypotheses exist in this case, both including the same two hypothetical aircraft and one including a hypothetical third aircraft as shown by links between the corresponding nodes in the figure. One attribute of a situation map node is thus a set of component aircraft nodes. Hypothesis credibility is also a situation map node attribute. *A posteriori* probability would be a reasonable credibility measure. The value of that attribute is a function of the credibilities of the supporting aircraft hypotheses.

The intelligence report level is treated as a separate, degenerate hierarchy in the figure. The figure shows two intelligence report nodes. Links indicate that one of these reports supports both situation map hypotheses while the second report supports only one of them. The credibility attribute value of each situation map node is also a function of the credibility of each intelligence report node linked to it.

The radar track and radar report levels form another hierarchy. So do the ELINT track and ELINT report levels. A sequence of report nodes is linked to a corresponding track node to represent the hypothesis that they were all caused by the same object, aircraft or emitter.

Similarly, the links between the aircraft nodes and both kinds of track nodes represent the hypothesis that the tracks are all of the same aircraft. The credibility of an aircraft hypothesis is a function of the credibilities of the two kinds of track hypotheses supporting it.

It will prove useful later to have explicit definitions of certain attributes of radar report and radar track nodes. We do so in pseudo-computerese as follows:

```
Level: radar-report
Attributes: report-time
            track-identifier
            state-estimate
                North position
                East position
                North velocity
                East velocity
            state-covariance
            ...
            associated-tracks


Level: radar-track
Attributes: last-associated-report
            report-history
            track-credibility
```

The names of the attributes suggest their intended meanings. But attributes are given pragmatic meaning by the way the attributes are manipulated by knowledge sources. They are analogous to the elements of a state vector in this sense.

Knowledge sources embody knowledge about how to solve a problem. Consider the following fragment of knowledge about radar tracking:

A sequence of radar reports caused by a particular aircraft usually have the same track identifier. An exception may occur if two aircraft approach closely at some time, in which case the track identifiers are swapped at roughly the time of closest approach.

It can be converted into the following fragments of knowledge about collecting radar reports into radar tracks:

Given a radar report node that is not associated with any radar track node and given a radar track node, if the radar report node's track identifier is the same as that of the radar track node's last associated radar report node, then associate them.

Given two radar track nodes, if their histories of associated radar report nodes indicate a close approach, then create two new radar track nodes with histories composed by splitting the original track nodes' histories at the time of closest approach and rejoining them with the track identifiers swapped after that time.

A knowledge source based on the first of these fragments is expressed in pseudo-computerese as follows:

```
Applies-to:
        a-radar-track , a-radar-report

Invocation-condition:
        associated-tracks of a-radar-report =
                empty-set

Use-policy:
        all-true-once

Production-rule 1:
Condition:
        track-identifier of last-associated-report
```

                    of a-radar-track =
            track-identifier of a-radar-report

        Action:
                last-associated-report of a-radar-track
                        := link to a-radar-report ;
                report-history of a-radar-track
                        :== link to a-radar-report ;
                associated-tracks of a-track-report
                        := link to a-radar-track

Here ":=" symbolizes assignment, ":==" signifies addition to a set, and ";" sequences simple actions in a compound one.

The knowledge source is quite simple, with just one production rule. That is atypical. Knowledge sources using production rules typically employ between ten and thirty production rules. A knowledge source realizing the second fragment would be more complex. It would include one or more production rules used to determine whether a possible close approach occurred and when.

The details of any particular control structure are complex. And the motivation for that complexity is no. apparent in an example involving just one or two knowledge sources and a few nodes. So no attempt is made to include control structure details in this illustration. A sketch of the blackboard changes one would prefer under particular circumstances provides a better feel for the control structure's gross behavior. It also illustrates how the different components of a blackboard system can come together to solve a problem.

Assume that no reports have been received of any sort by the blackboard system. Then one situation map node exists with no links to aircraft nodes. This represents the hypothesis that no aircraft are in the area of interest. Then an intelligence report is posted on the blackboard. It warns that some number of aircraft of a particular type or types are expected to enter the area during a specified time interval across a specified portion of the area's boundary. Aircraft nodes are then created with the appropriate types, all linked to a new situation map node. The credibility of this new situation map node is the same as that of the intelligence report. The credibility of the old situation map node is appropriately adjusted downward.

The radar track attribute of each new aircraft node is not filled in at this point. There are no radar track nodes yet. But an expectation is established that later examines newly created radar track nodes. If one is created in the appropriate time interval and the appropriate place, a link to that radar track becomes the value of the associated track attribute. If the expectation goes unsatisfied, the aircraft node is deleted and the credibility of each associated situation map is reduced. Whenever the credibility of a situation map node slips below a certain level, that node is also deleted. Any aircraft nodes linked only to that situation map node are also deleted. The credibilities of all remaining situation maps are then re-normalized.

Receipt of the first few radar track reports causes them to be posted on the blackboard, but no more. Only when three report nodes having the same track identifier appear on the blackboard is a radar track node created to represent the hypothesis that they are from a single aircraft. In this manner, the creation of false radar track nodes based on radar false alarms is largely avoided. The resulting node may then be linked to an existing aircraft node by the aforementioned expectation.

Failing that, a new aircraft node is created to which the new radar track node is linked. Then the cross-product is formed of the old situation map hypotheses and the pair of hypotheses that the radar track was or was not caused by an aircraft. One new situation map node is created corresponding to each existing one. The new situation map nodes are copies of the old nodes, each with a link to this aircraft node added. Some portion of the credibility of each old situation map hypothesis must also be transferred to the corresponding new hypothesis. At this point, the knowledge source which removes insufficiently credible situation map nodes is again applied to reduce the number of situation map hypotheses maintained.

The accretion of ELINT reports into ELINT tracks is similar to that of radar reports into radar tracks. But the creation an of ELINT track does not satisfy any expectations or trigger

the creation of an aircraft node. Rather it triggers a search for aircraft nodes of a type which could produce the sensed emission and which has a history of estimated positions (implicit in the radar tracks' report history) consistent with the ELINT track's history of bearings (similarly implicit). The ELINT track node is linked with any and all such aircraft nodes. The credibility of any such aircraft nodes is increased appropriately to reflect evidence that the hypothesis it represents is correct. Such a credibility increase must also be propagated up to the situation map nodes. Creation of a new aircraft node triggers a similar search for supporting ELINT tracks.

Prioritization among the knowledge sources carrying out the aforementioned actions can be relatively simple. The arrival of a new input datum should trigger a locus of activity on the blackboard which propagates up the network of levels, with pauses to spread down along different hierarchies as appropriate. All of the activity directly triggered by one datum should be completed before the next input datum is posted. To keep the amount of inter-input processing reasonable, the diversity of hypotheses created in the normal course of processing must be limited. Thus as additional radar reports arrive, the posted nodes are simply associated with radar tracks on the basis of track identifiers as in the above knowledge source example. It would be possible to create track nodes expressing all possible hypothetical combination of track reports without regard to track identifiers. But the processing required to create, qualify, and eventually delete most of these nodes would be wasteful given the number of possible combinations.

But when should the control structure invoke the knowledge source which tests for a close approach of two aircraft and creates new track nodes to reflect a possible confusion of track identifiers? One answer would be after the completion of every invocation of the knowledge source associating a new radar report with an existing radar track. But that would mean frequent invocations, usually producing no change. An alternative is to invoke that knowledge source only when some other, less frequent, occurrence suggests the possibility of a close approach by two aircraft and consequent track identifier confusion be considered.

In the scheme described above, ELINT tracks are associated with an aircraft if they are consistent with the aircraft's hypothesized type and with the radar track. If the tracks are geometrically consistent but the nature of the tracked emission is inconsistent with the aircraft type, one possibility is that the aircraft hypothesis was wrong with regard to type and should be discarded or modified. But another possibility is that the radar track history actually corresponds to two different aircraft at two different times due to a track identifier confusion during a close approach. If ELINT tracks are already linked with the aircraft node as support for the hypotheses, the possibility of a close approach should be investigated first.

The above sketch does not reflect the only manner in which the example problem might be solved. It reflects various options for incrementally advancing the problem solution. Choosing which option to use in a particular situation can require subtlety if one wishes to be computationally efficient. Not illustrated are the additional subtleties of advising the control structure how to achieve that sequencing. Experience is required to make such choices wisely. Experience is also important in the construction of knowledge sources, the choice of blackboard levels, and the selection of nodal attributes. Simple examples can only suggest the subtleties involved.

4. SUITABILITY OF BLACKBOARDS

The above sketch of possible blackboard changes illustrates a major reason why the blackboard problem solving methodology is suitable for multi-system report integration. The ordering of changes adapts appropriately to the arrival of very different sorts of input data in different orders.

If any intelligence report involving a particular aircraft arrives after radar track reports corresponding to it, the hypothesis that it exists will still have been formed. The credibility of the situation map hypotheses supported by that aircraft hypothesis will be increased once the intelligence report is incorporated into the support for those situation map hypotheses. ELINT reports are not discarded immediately if they do not confirm an existing aircraft hypothesis. They are saved for possible confirmation in the future. And exceptional occurrences need be

considered only when evidence suggests they occur. The close approach of two aircraft leading to track identifier confusion being the case in point.

This adaptability in the operation of a blackboard system is a consequence of the control structure's opportunistic invocation of knowledge sources, the knowledge sources' modularity of forming or altering hypotheses, and the blackboard's structured composition of hypotheses. Any knowledge source can be invoked after any other completes, depending on the state of the blackboard, i.e., of the problem's solution, at that point in time.

The blackboard methodology also provides a means for managing the complexity of large multi-system report integration problems. Knowledge sources are modular in their applicability to all nodes of a given level, or tuples of given levels, but only to those nodes. Modularity is also achieved by expressing a partial problem solution as hypotheses supported by a hierarchy, or a set of linked hierarchies, of sub-hypotheses ultimately based on input data. Solution to individual parts of a particular multi-system report integration problem can be conceptualized and implemented without dwelling on the details of how the results of solving one part are used in the solutions of other parts.

Standard algorithms can be used where appropriate to solving part of the problem. But special pre- or post-processing may be required. Such pragmatic features of a standard algorithm's use in a particular context can be isolated from the algorithm itself by encapsulating them in separate knowledge sources. Explicitly separating formal and heuristic aspects of a problem's solution can highlight the heuristic aspects. It illuminates the assumptions, explicit or implicit, upon which they are based. Modifying the heuristic aspects without compromising the formal aspects also becomes easier.

5. WORK IN PROGRESS

The Heuristic Programming Project Group of Stanford's Knowledge System Laboratory is trying to

- realize a new generation of software architectures using parallel computation to speed up AI applications and

- specify multiprocessor system architectures for carrying out those computations efficiently.

Among the issues being investigated are

- recognition of opportunities for parallelism in the solution to a problem and

- expression of that potential parallelism in a problem solving framework that can exploit it.

In particular, this effort is focusing on signal understanding problems and blackboard-like frameworks.

Blackboard systems appear to be intrinsically parallel. At any time, there can be many potential invocations of knowledge sources. Those involving different nodes seem eligible for parallel execution. Within knowledge sources, production rule conditions could be evaluated in parallel. And some production rule actions could be safely executed in parallel. Currently two different blackboard systems are under development, each investigating a different approach to expressing opportunities for parallel computation or requirements for serial computation. Applications of these experimental systems used in evaluating their effectiveness.

The focus on signal understanding problems follows in large part from the focus on blackboard systems. The two mate well. But signal understanding problems are important in their own right. When signal understanding is defined broadly, it includes sensor data fusion and multi-system report integration. That class of problems is large and of considerable interest to the military.

Two signal understanding problems have been investigated so far as part of the current

project. They are referred to as the TRICERO/ELINT and AIRTRAC problems. While generally similar, each problem is expected to push the research into recognizing opportunities for, and expressing, parallel computation in different directions.

In the TRICERO/ELINT problem, streams of ELINT emitter/bearing measurements must be combined to estimate the flight paths and operating modes of non-cooperating aircraft. The problem is named after ESL's TRICERO blackboard system for solving a problem of which this one is just a component. The knowledge of how to solve the TRICERO/ELINT problem has already been worked out, albeit without attention to opportunities for parallel computation. So work on this problem is further along.

The AIRTRAC problem is recognizing aircraft flying across a national border and heading for particular airfields used by smugglers. The smugglers' aircraft must be picked out of the normal air traffic across that border. To solve the problem, aircraft destinations must be recognized, not just flight paths and types. Streams of radar reports from multiple radar systems are available. But the low altitude coverage of those radars is assumed to be limited and the smugglers are assumed to know the coverage limits. So smugglers can try to avoid detection. They can also maneuver their aircraft evasively to disrupt tracking. Such behavior is a sure sign of a smuggler's aircraft, but makes the recognition of a destination difficult.

To complicate the AIRTRAC problem further, distributed aeroacoustic tracking systems using modest batteries of acoustic sensor arrays(1,7) are placed across large holes in radar coverage. These systems provide tracking reports within their limited coverage. Because such systems are passive and readily moved, the smugglers are assumed to be unaware of their coverage and so unable to avoid detection by these systems. These systems also use acoustic signature information to provide aircraft class estimates along with tracking reports.

Initial solutions to both problems should be completed in both experimental blackboard systems by the end of the year. Moreover, each solution should have been applied to several problem scenarios on realistic simulated multiprocessors. These experiments will determine how much parallelism was realized and may suggest alternative ways of realizing more parallelism.

6. REFERENCES

(1) J.R. Delaney and R.R. Tenney, "Broadcast Communication Policies for Distributed Aeroacoustic Tracking", Proceedings of the 8th MIT/ONR Workshop on C³ Systems, Cambridge, MA, July, 1985, pp.195-199.

(2) L.D. Erman, F. Hayes-Roth, V.R. Lesser, and D.R. Reddy, "The HEARSAY-II Speech Understanding System: Integrating Knowledge To Resolve Uncertainty", Computing Surveys, v. 12, December 1980, pp. 213-253. Also reprinted in (8).

(3) B.Hayes-Roth, "A Blackboard Architecture for Control", Artificial Intelligence, vol. 26, no. 3, July 1985, pp. 251-321.

(4) H.P. Nii and E.A. Feigenbaum, "Rule-Based Understanding of Signals", in D.A. Waterman and F. Hayes-Roth, Pattern-Directed Inference Systems, Academic Press, San Francisco, 1978. pp. 483-501.

(5) H.P. Nii, E.A. Feigenbaum, J.J. Anton, and A.J. Rockmore, "Signal-to-Symbol Transformation: HASP/SIAP Case Study", AI Magazine, vol. 3, no. 2, Spring 1982, pp. 23-35.

(6) J. Rice, "POLIGON: A System for Parallel Problem Solving", Knowledge Systems Laboratory Technical Report 86-19, Stanford University, 1986

(7) R.R. Tenney and J.R. Delaney, "A Distributed Aeroacoustic Tracking Algorithm", Proceedings of the 1984 American Control Conference, San Diego, CA, June 1984, pp. 1440-1450.

(8) B.L. Webber and N.J. Nilsson (eds.), Readings In Artificial Intelligence, Tioga Press Company, Palo Alto, 1981.

An Instrumented Architectural Simulation System

by

Bruce A. Delagi, Nakul Saraiya, Sayuri Nishimura,

and Greg Byrd

Knowledge Systems Laboratory
Computer Science Department
Stanford University
Stanford, California 94305

Table of Contents

List of Figures

ABSTRACT

AN INSTRUMENTED ARCHITECTURAL SIMULATION SYSTEM

Simulation of systems at an architectural level can offer an effective way to study critical design choices if (1) the performance of the simulator is adequate to examine designs executing significant code bodies -- not just toy problems or small application fragments, (2) the details of the simulation include the critical details of the design, (3) the view of the design presented by the simulator instrumentation leads to useful insights on the problems with the design, and (4) there is enough flexibility in the simulation system so that the asking of unplanned questions is not suppressed by the weight of the mechanics involved in making changes either in the design or its measurement. A simulation system with these goals is described together with the approach to its implementation. Its application to the study of a particular class of multiprocessor hardware system architectures is illustrated.

1 INTRODUCTION

Simulation systems are quite often developed in the context of a particular problem. To a degree, this is true for SIMPLE, an event based simulation system, and CARE, the computer array emulator that runs on SIMPLE.[1] The problem motivating the development of both SIMPLE and CARE was the performance study of 100 to 1000-element multiprocessor systems executing a set of signal interpretation applications implemented as "1000 rule equivalent expert systems" [2].

A set of constraints pertinent to this problem governed the design of SIMPLE/CARE. The applications represented significant bodies of code and so simulation run times were expected to be an important consideration. Moreover, the issues involved with the interactions of multiprocessor system elements were sufficiently unexplored prior to simulation that simplifications in the CARE system model, specifically with respect to element interactions, were suspect. This need for detail was, of course, in tension with the need for simulation performance. The ways that simulated system components would be composed into complete systems was initially difficult to bound. Further, it was clear that the models of these components would be elaborated over time and would undergo substantial change as design concepts evolved. It was also clear that the ways of examining the operation of these components would change independently (and at a great rate) as early experience indicated what alternative aspect of system operation *should* have been monitored in any given completed run.

The design goals that emerged then were (1) that the simulation system should support the management of substantial flexibility with regard to simulated system structure, function, and instrumentation and (2) that, in order to accomplish runs in acceptable elapsed times, the detail of simulation should be particularly focused on the communications, process scheduling, and context switching support facilities of the simulated system -- that is, on just those aspects of system execution critical to multiprocessor (as opposed to uniprocessor) operation.

1.1 Design Time Interaction And Run Time Operation

Encapsulation of the state of design components with the procedures that manipulate that state is one clear way to manage design evolution. Such encapsulation partitions the design along well defined boundaries. Components (by and large) interact with other components only through defined *ports*. Connections between components terminate at such ports. When a system simulation is initialized, connections are traced so that for every port, the simulator knows the connected (terminating) ports together with their containing components. Once such initialization is complete, that is, throughout the simulation run, assertions about the state of a port of one component can be directly translated to assertions about the state of connected ports of other components.

Partitioning issues of system **structure**, **component behavior**, and **instrumentation** into separate domains of consideration helps in managing a design that is both fluid and complex. **System structure**, that is, the relationship between components, can be specified through use of an interactive, graphics structure editor and is largely independent of component function per se. **Component behavior** is encapsulated in a set of definitions pertinent to the given class of component. Each component in a SIMPLE simulated system is a member of a class defined for that component type. **Instrumentation** is automatically and invisibly made part of the definition of each simulated component that is to be monitored during a run. This is done by arranging that the class of every component to be monitored is a specialization of the general *instrumented-box* class. The basic data structures and procedures for monitoring simulated components and maintaining the organizational relationships between each component and its related instrumentation are inherited through this general, ancestral class and are thus made a separate, substantially independent consideration in the design.

---

[1]SIMPLE and CARE were developed by the authors at the Knowledge Systems Lab of Stanford University. SIMPLE is a descendent of PALLADIO [1] optimized for the subset of PALLADIO's capabilities relevant to hierarchical design capture and simulation. It is written in Zetalisp [4] and currently runs on Symbolics 3600 machines and TI Explorers.

A further partitioning of concerns is employed to separate out the definition of the application programming language interface and its support (as provided by CARE) from the underlying information flow control governing component behavior. The behavioral descriptions of components (which are expressed as sets of condition/action rules) deal generically with gating information, independently of the structure of the information, between ports of the component and its internal state variables. This is separated in the component model definitions from the functions performed to create and manipulate the information so gated. The simulated implementation of the application programming language support facilities, on the other hand, relies only on the specifics of the information and its structure and plays no part in gating it between the components of the system. Changing the definition of the application language is thus done independently of changing component flow control behavior. The application programmer and the implementer of the application language interface may use whatever data structures seem suitable to them, be they numbers and keywords or procedure bodies and execution environments. The simulation system doesn't care.

The *component probe* definitions, that is, the specifications of what information should be captured for each component type, are separated from the descriptions of the behavior of such components. In designing for flexibility in the instrumentation system, it turned out to be important to further divide the information presentation from the information collection issues. The mapping from particular component probes to particular *instrument panels* and the transformations to be applied to the information as it passed from a given kind of probe to a given panel (and between panels) is captured in the *instrument specification*. This is a definition of what kinds of panels are included in an *instrument*, how they fit on an *instrument screen*, how they are labeled and scaled, and what information from which kinds of probes are displayed on each panel. The instrument specification also indicates what kinds of probes are to be connected to which kinds (that is, which classes) of components in the system.

Figure 1: Design Time Interactions and Run Time Representations

Putting together all the definitions of components, component probes, panels, instruments, applications interfaces, and inter-component relationships is done in a set of design time interactions by a system architect. These interactions are used by the simulation system to generate efficient run time representations so that simulation performance goals can be met. Figure 1 illustrates the partition between design time interactions and simulation run time operation. Structure editing pulls together components from the component library to produce a *circuit*. Associated with some components in the library, there are definitions for the syntax and underlying mechanisms of a multiprocessor applications language. These specify the

interface used to provide the program input to the multiprocessor system being simulated.[2] The definitions used to generate component probes are associated with each library component to be monitored. There may be several such definitions, each appropriate to measuring a different aspect of the associated component's operation. An instrument specification selects from these definitions, elaborates them with selections from a set of *probe operation modules* to include any pre-processing (for example, a moving average) to be calculated by the probe, and indicates under what conditions what information from the probe is to be sent to which panels of the instrument and how it is to be transformed and displayed there. Instrument specifications also partition the screen among the panels of the instrument. The end product of these design time interactions is an *instrumented circuit* and an *instrument*. The instrument comprises a set of instrument panels and a set of constraints relating them to the instrument screen. The instrumented circuit ties together instances of components, probes, and panels for a simulation run.

For each defined class of component and its associated probes, the design time interactions produce code bodies that accomplish simulation operations during a run. It is an attribute of the underlying Lisp base of the simulation system that changes in these definitions have immediate effect even during a simulation run -- an important capability during debugging.

2 STRUCTURE AND COMPOSITION

Design time interactions to specify a system include the establishment of component relationships. Such specifications can be said to accomplish the composition of the system from its components and so define its structure. SIMPLE supports hierarchical composition: components may be described in terms of a fixed set of relationships among their subcomponents. Additionally, such composite components may have function beyond what can be inferred strictly from their composition. All this can then be included a higher level composite (as shown in figure 2) and so on indefinitely until the top level "circuit", the system structure, is reached.
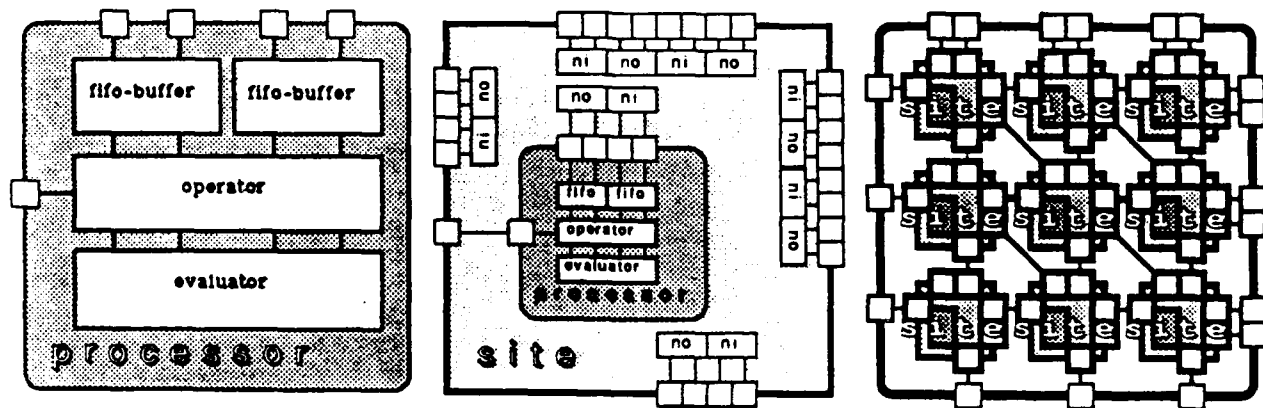
Figure 2:   Hierarchical Composition

The behavior induced on a composite component from its parts changes according to the behavior of its parts. Thus, for example in figure 2, if at any time during a simulation the function of CARE *operator* components is changed by redefining their operation, the behavior

---

[2]The language primitives supplied can be used to define multiprocessor language interfaces for either shared-variable or value-passing paradigms. As supplied, the language interface built on these primitives supports value-passing on streams between objects but alternative interfaces can be (and have been) easily defined in terms of the given primitives.

of the nine-site grid is in immediate correspondence.[3]

Composition is described graphically and interactively in SIMPLE by picking a previously specified component type from a menu, placing it in relationship to other components with "mouse" movements, and, through the same means, specifying the connections between its selected ports and those of other components (as indicated in figure 3).
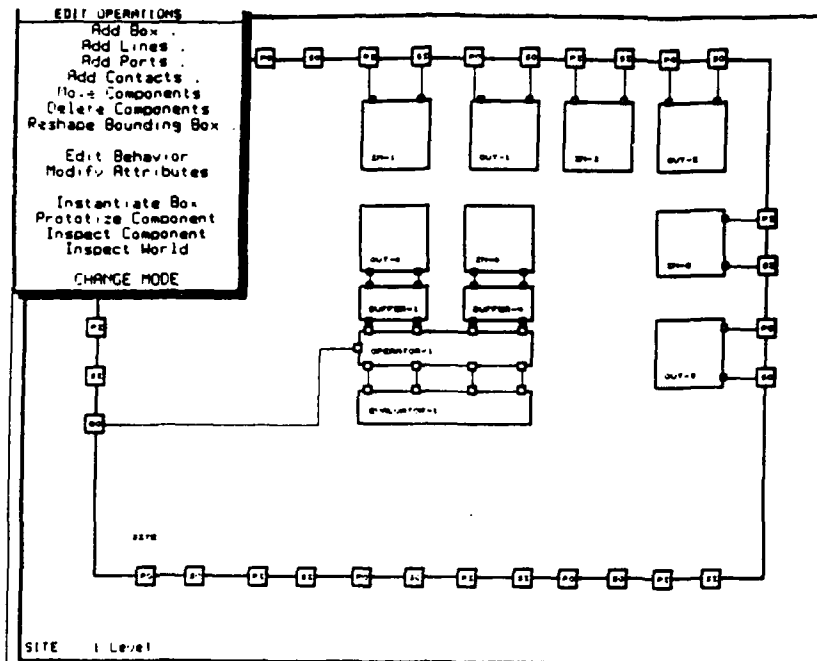
Figure 3: Graphic Structure Specification

Through another menu selection, ports can be defined for the new composite component so that it, in turn, can be fitted into yet higher level structures. Such external ports can be connected directly to ports of sub-components "within" the composite. If this is done, information appearing on that external port will be the responsibility of the connected sub-component. By this same means, a component previously described as a base level component, can be redefined as a composite of yet lower level elements as its design is elaborated with further details.

Components and (internal) connections can also be deleted from a library component and replaced with substitute components. After all sub-components and connections have been added, deleted, elaborated, and replaced as required, the completed structure can then be entered into a library of components and used in turn to compose higher or equivalent level components.

2.1 CARE Base Components

CARE supplies a small library of system level base component types. Currently these are the *net-input*, the *net-output*, the *fifo-buffer*, the *operator*, and the *evaluator*. The net-input, net-

---

[3] However, for reasons concerning simulation performance and because of their relatively low frequency, changes in the number and names of the internal state variables of components and the structural relationships between sub-components of a composite are not reflected in an already instantiated circuit. Changes in the internal structure of a CARE *site* library component, for example, will be reflected only in circuits instantiated after the change took effect. For this reason and to reduce long term storage requirements and load time for the fundamentally iterative circuits that we primarily study, we do not keep files of instantiated circuits. They are instantiated as needed from a high level library component with the same prototypical structure.

output and fifo-buffer accept (or block), route, and buffer transmissions. They do so in accordance with a dynamic, flow-controlled, multicast, cut-through communications protocol as described in [3]. The evaluator does the real work of the application: evaluating the application of functions to their parameters. The operator does the overhead work associated with such evaluations: for example, scheduling processes and sending and receiving (but not routing) messages.

In keeping with the objective of focusing simulation cycles on the aspects of the simulation particularly relevant to multiprocessor operation, the behaviors of the net-input, net-output, and fifo-buffer component classes are defined in fair detail, that is, at the register transfer level. Routing operations are described procedurally and assumed to occur within a time set by a parameter to the simulation. As indicated previously, the simulation of the operator and evaluator is broken into two aspects: the control of the flow of information and the functions performed on that information. The former is described in terms of SIMPLE behavior rules (as documented in section 3), register transfer by register transfer. The latter is described directly in terms of procedures and the simulated time taken by such procedures is modeled. In the case of the operator, this is done as a function of the number of storage cells manipulated during an operator procedure. In the case of the evaluator, this is done as a function of the execution time used by the machine executing the simulation, that is, the simulation vehicle.

2.2 CARE Composite Components

The prototypical composite component supplied with CARE is the *site*. As supplied, it includes net-inputs and net-outputs for up to eight "neighboring" components (generally other sites), a net-input and a net-output with associated fifo-buffers for local receptions and transmissions, and, finally, an operator and evaluator as described above. Specializations of the site, for example, the *torus-site*, exist in the library to fit the site into alternative topologies by supplementing the site routing and wiring procedures as appropriate to the topology.

2.3 Automatic Composition in CARE

Although any connection of components can be created by the means noted previously, for some repetitive, well patterned systems of connections, composition can be automated. The CARE library includes a component, the *iterated-cell*, which represents a template for the creation of composite components by iteration of a unit cell. The unit cells (for example, the torus-site) are specializations of other components (for example, the site) as just discussed. The specializations include a method for responding to a request to provide a wiring list. Such a list associates each source port of a cell with the corresponding destination port (in terms of port names) and the position of the destination cell relative to the source cell in the iterated structure. The iterated cell component uses this information to make the required connections between each of its constituent cells.

3 SPECIFYING BEHAVIOR

SIMPLE is an event based simulator. The behavior of a simulated component is described in terms of responses to the events pertinent to that component. A component's response may include consequent events to be handled by the simulator as well as direct operations on component state. Assertion of consequent events and the responses to them (involving further consequences) drives the simulation. When there are no more events to handle, the simulation is complete.

To maintain modularity in a simulation system, responses to simulation events should be local to the affected component and its defined ports, that is, its connection to the remainder of the simulated system. The composition system of the simulator maintains the relationship between ports of one component and those of other components connected to them. Assertions

relative to a port of a component are thus systematically translated to events pertinent to components connected to it. This is the general mechanism for event propagation between components. In a limited number of cases, a direct operation on a related component may be appropriate. With fair warning about its possibility of abuse, a facility is provided to accomplish this.

3.1 Behavioral Rules

The behavior of a component is described in terms of its responses to pertinent events. Each event stipulates the component affected, its port or state variable signalled with an assertion, the asserted value, and the simulated "time" of the event. The time of an event may be thought of as the "current" simulation time. Differences in event times represent the temporal relationship between events. Event times in SIMPLE simulations are monotonically increasing.

For each type of component, there is a procedure to handle pertinent events. The arguments to the procedure are those stipulated by the event (as just described). The procedure tests for conditions and, as satisfied, asserts or directly effects consequent actions. The conditions may include arbitrary predicates on the event parameters and the state variables of the component.

Event based simulators are based on the assumption that state and port variables remain unchanged until explicitly modified. Synchronous designs, that is, those in which the opportunities for state change are temporally quantized to a clock, can be modeled in such implicitly asynchronous, event based simulators by asserting the clock signal on a port of each and every clocked component of the simulated system. If only some of the components in a system need take action on each clock signal, there is an obvious inefficiency in this approach that is crippling for systems with even a modest number of components.

If, however, event times in an event based simulator are restricted to integers, the clock can be assumed. All that is needed is a way to detect the event for which a boolean combination of conditions as strobed by an assumed clock is first met. Primitive condition predicates are supplied for detecting an "edge" (a value changed by the current event) with a coincident "level" (a value set before the current event) of two ports or state variables of a component in either of the two possible event sequences. The predicate both-states in the example evaluator behavior rule shown in figure 4 has these semantics.

```
;;If the evaluator is ready and there is at least one runnable process...
((or (both-states Evaluator-Status⁴ 'ready Evaluator-Queue-Status 'some)
     (both-states Evaluator-Status 'ready Evaluator-Queue-Status 'full))
;;... make it current, start evaluation, and adjust status as per removal.
 (setq Evaluator-Status 'busy)                              ;block rule
 (assert-state Evaluator-Status 'busy now)                  ;next event
 (setq Current-Evaluation (queue-take Evaluator-Queue))     ;note process
 (user-evaluate Current-Evaluation now)                     ;execute it
 (send self :evaluator-queue-decreased now))                ;note change
```

Figure 4: Example Condition/Action Behavior Rule

Figure 4 illustrates the generality of SIMPLE behavioral descriptions. The underlying object-oriented programming system, Flavors [4], in which SIMPLE is implemented provides for direct reference of component state variables. The conditions and actions of behavior rules for a component then need only name the component's port or state variable (as stipulated in the definition of that component type) to get or change the appropriate value in the component instance for which the event is pertinent. Actions may include arbitrary procedures: for example, the procedures user-evaluate and queue-take in the given example.

---

[4]By convention, component state variables are written in capitalized form.

3.2 Using Methods

The environment for the execution of the procedures defining responses to events includes the state variables and ports of the component instance for which the event is pertinent. These procedures are Flavor *methods* [4] (in this case corresponding to the :ApplyRules message) of the component type and, as just noted, refer implicitly to the state variables of the component instance handling the event. Other methods may be defined for simulated components: for example, the :evaluator-queue-decreased method invoked in figure 4. Such methods have proved to be a natural way to realize the functional operations of components not described by behavior rules.

The composition system leaves information about the enclosing and contained component instances for each simulated component in system defined state variables of that component. With this information, methods directly referencing the ports and state variables of such related components may be invoked as needed. This is a useful but sharp-edged facility. The warning about loss of modularity given previously applies here.

4 INSTRUMENTATION

The results of a simulation are primarily the insights it provides into the operation of the simulated system. The "insight" we frequently experienced using an early version of the simulation system was that more interesting results could have been produced by the run just completed if only the instrumentation had been different. With this in mind, the design for the current version of the simulation instrumentation system was aimed at flexibility. This was attained without significant performance impact by building efficient run-time system structures before each run, as outlined in section 1.1, from the declarations defining the instrumentation.

The organization of the instrumentation system is pictured in figure 5. The simulator interacts with component instances through assertions, that is, calls on an assert function, in behavior rules (the methods associated with :ApplyRules messages). All instrumented components are specializations of an *instrumented-box* (as well as other classes). After each invocation of :ApplyRules for such components, the :ApplyRules method for a generic instrumented-box is applied. This causes invocation of the :trigger method for each *component-probe* associated with that component. Since this flow of measurements is accomplished by means invisible to the the writer of behavior methods for a component, the concerns surrounding component design are effectively partitioned from component instrumentation. The remainder of this section details these "invisible" means used to accomplish measurement flow during a simulation run as the measurements are staged from components through component probes to instrument panels.

4.1 Component Probes

The first filtering of events is done by component probes. Some events cause no further measurement activity since, as it turns out, not all events merit action on the part of the instrumentation system. The parameters of the event and the ports and state variables of the instrumented component dealing with the event are available to the component probe as are the state variables of the probe itself. Each piece of the selected information is tagged with an identifying keyword and passed along as the parameters of the :trigger method along with a keyword identifying the type of component probe, a number representing the current event time, and a pointer to the component with which the information is to be associated in the display. This pointer might be to some component related to the one actually handling the event, for example, the component enclosing it.

Component probes may be composed of predefined probe operation modules to do standard calculations (for example, moving averages) and then to forward the results to selected panels. In order to automate the composition of probes to accomplish such operations, each of these operations is chained together by invoking the method for that probe that is associated with

E-8
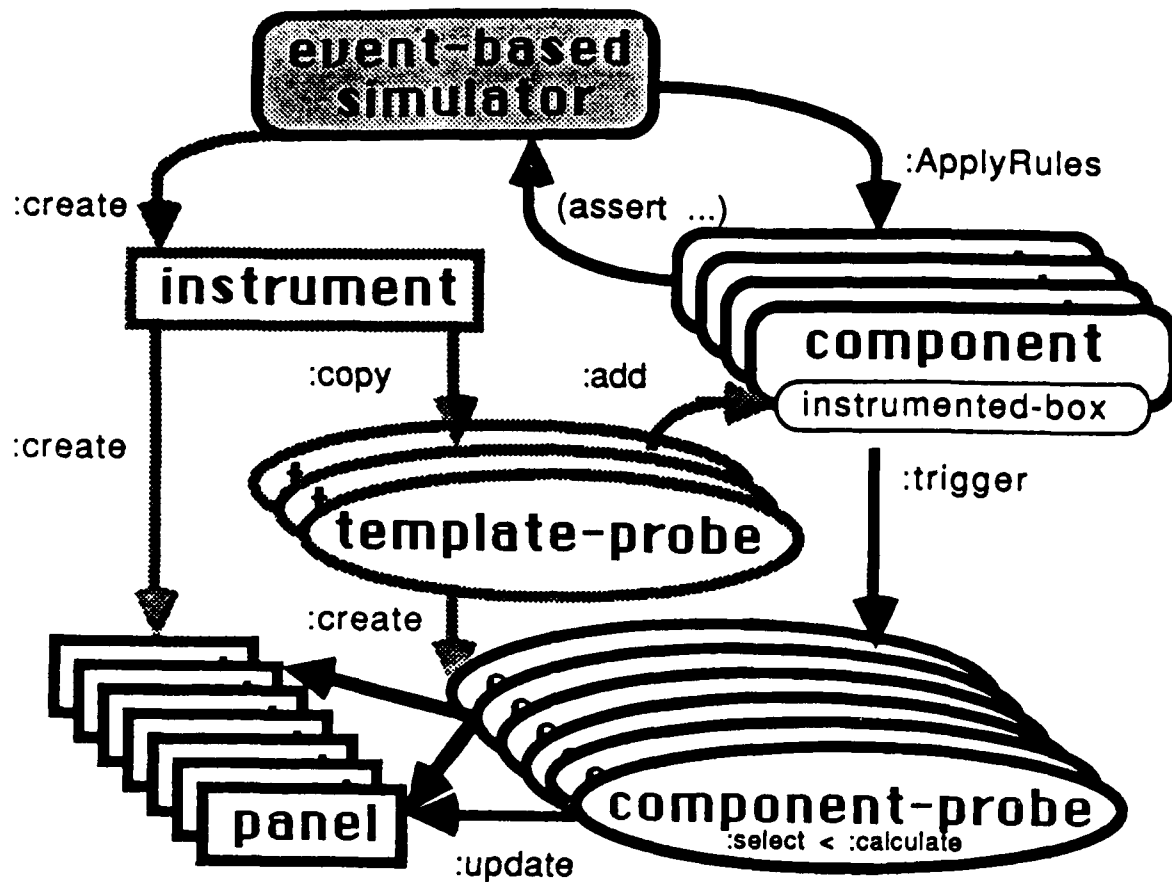
Figure 5: Instrument System Organization

the system-defined message name of the generic next operation. Thus, the :trigger method calls the :calculate method of the probe which, in turn, calls its :select method which, finally, calls the :update method of the selected panels associated with the probe. Probes are composed by naming them as specializations of appropriate probe operation modules (for example a :calculate module for moving averages) as desired. The default, if no specializations are stipulated, is to pass through information without change to all the panels associated with a probe.

Information flow between components and panels is accomplished by the component probes associated with each instrumented componen¹. The creation of such component probes and their association with appropriate components (by execution of :add methods) accomplishes the instrumentation of a circuit. This is done when an instrument is created. During simulation initialization, the components of the circuit (and their sub-components) to be instrumented are (recursively) examined by each *template probe* defined for the instrument to see if they are to be monitored. If so, the :copy method for the given template probe is invoked to create a new instance of the appropriate component probe and add it to the probes connected to the component. Each template probe previously received the identifiers for the panels to which its clones should send information. These will be the panels identified when a component probe invokes the :update method.

4.2 Instrument Specifications

The operations performed by an instrument panel are to:

- *Find* information previously stored according to the component pointer supplied by the :update method;

- *Link* new data structures as needed (to save such information) to other such structures of the panel;

- *Save* in these data structures the results of expressions that reference indicated keyed information from the :update parameters and the prior contents of the structures;

- *Send* the results of periodic analyses on the information associated with a panel for display by the same panel or by some other; and

- *Show* processed information in the manner specified for the panel.

The defaults for the panel operations supply the most commonly required specifications implicitly, so simple operations are simply specified. These defaults can be overridden as needed and either predefined or user specified alternatives for the panel operations can be selected in their place. Arbitrarily complex (Lisp) expressions can be used to specify the transformations between the information provided by a probe and that saved and displayed by the panel.

These transformations and all the default overrides for the panel operations that are stipulated in the instrument declaration are scanned when a new instrument is created for a simulation session. They are compiled at that time into code bodies referenced by run time control blocks associated with each panel. A simulated system is instrumented by examining all of its components and attaching to each component the copies of template probes specified by the instrument definition that are appropriate for the component (by means of calls on the :copy and :add methods for the probe). This can be a many to many relationship as shown in figure 6.
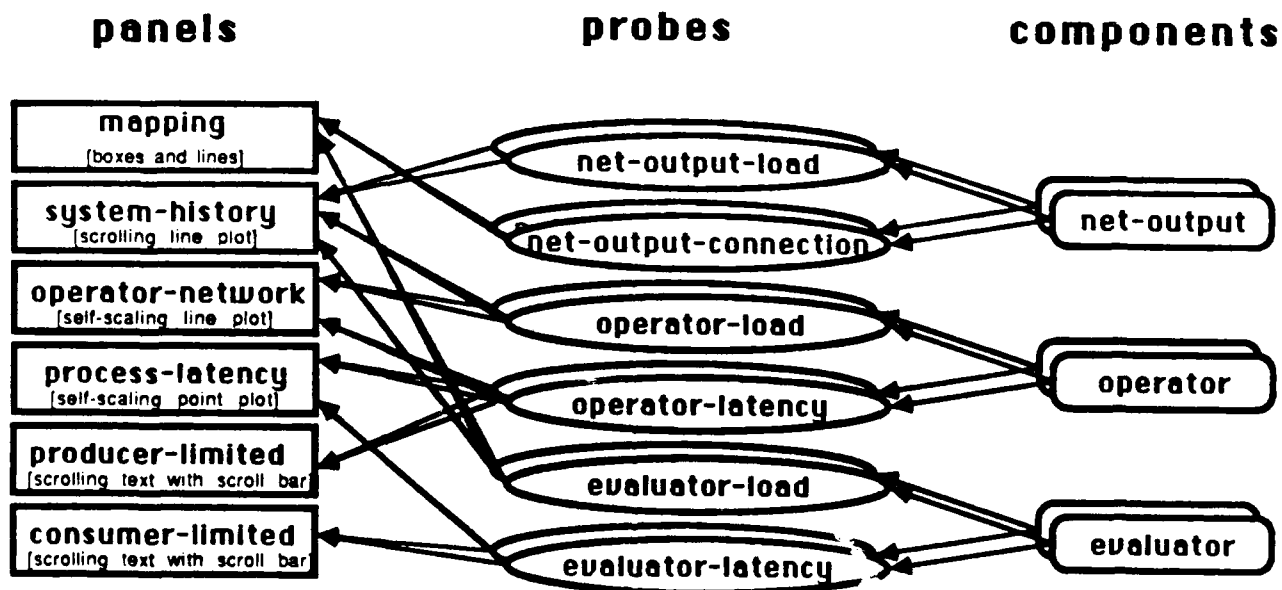
Figure 6: Instrument Probe and Panel Relationships

Component probes to measure "load" and "latency" are specified in the given example for each operator and evaluator in the circuit. The "load" and current "connection" for each net-output is also to be monitored. Some panels, for example the one showing "consumer-limited" processes, receive inputs from only one type of component probe, those measuring evaluator latency. Others, such as the one measuring "process-latency" receive inputs from more than one kind of probe (in this case, from probes measuring operator latency as well as those measuring evaluator latency). A way must thus be provided to distinguish the type of probe sending information to a panel; this is described in the next section.

Some probes send information to only one panel, for example, the net-output connection probes. Others monitor information which is needed by several panels, for example, the operator latency probe. Transformation of the raw information provided by a probe will need to be specialized to the information expected by each panel receiving it. A general way to stipulate these transformations is stipulated in the next section.

5 EXAMPLE PANELS

Some example panels are described in this section to give a feel for the instrumentation possibilities available in CARE and elaborate on how the requirements described in the previous section for probe type identification at a panel and per panel specialization of the information provided by a probe are handled.

5.1 Point Plot Panels

The first panel (shown in the left half of figure 7) is an example of a *point plot panel* used to generate a scatter plot. As an option, only points representing simulated activity over a limited past history from the most recent event time are kept for display. In this example, resource load[5] information is provided by the operator-load and evaluator-load component probes attached respectively to the operators and evaluators of the system.
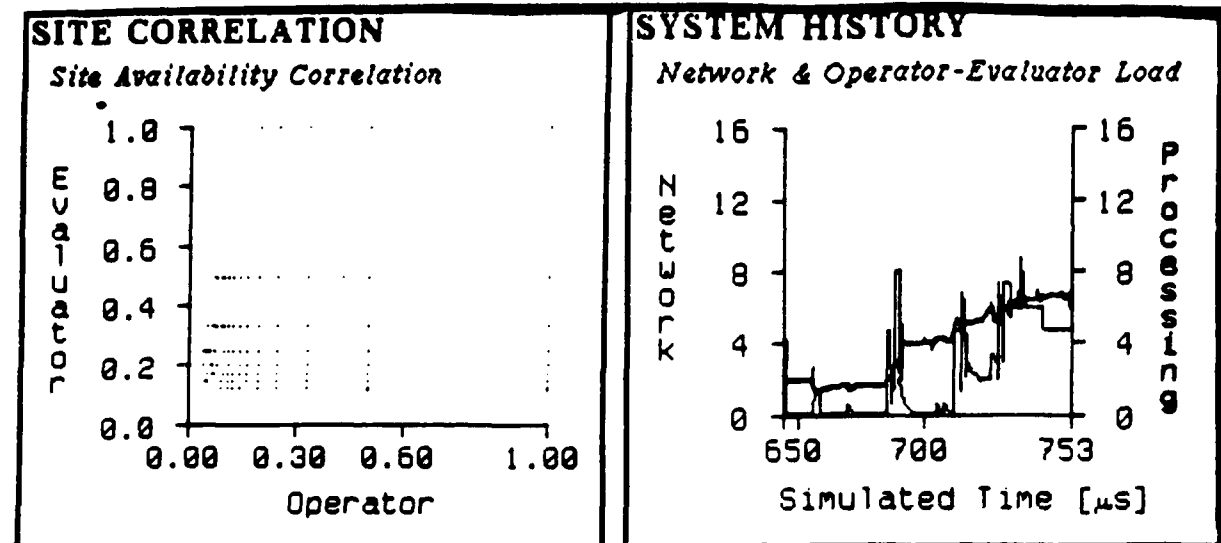
Figure 7: Point Plot and Scrolling Line Plot Panels

The balance between the "availability" of the evaluator and operator of each site, that is, the complements of their respective loads, is displayed during the simulation as events are processed that change this measure. In order to avoid capturing information at too fine a temporal granularity, previously gathered information for a given site is overwritten if it is within a given sampling interval of the new information. Information that is beyond a given history range is dropped. The scale of availabilities displayed is fixed between 0 and 1.0. The panel specification to declare all this and to also stipulate the axis labels is shown in figure 8.

---

[5] Resource load is defined as (1 - 1 / (1 + aggregate-queue-length)) where the aggregate queue-length is the sum of the lengths of all queues providing work for the resource.

```
'((("Operator") (0 1.0) (- 1 (:operator-load :busy)))        ;Bottom axis
  (("Evaluator") (0 1.0) ((- 1 (:evaluator-load :busy))))  ;Left axis
  :find (find-sample-distinct (:simulator :time) .sampling-interval)
  :show (recent-history (:simulator :time) .point-panel-history-range 0))
```

Figure 8: Site Correlation Panel Specification

5.2 Scrolling Line Plot Panels

An example of a *scrolling line plot panel* is shown in the right half of figure 7. This panel sums the loads seen by the resources in the simulated system and displays this as a strip chart, the "system history". Some of the same probe load information used by the previous panel is used in this panel as well, but with different transformations defined in the panel specification as shown in figure 9.

```
'((("Simulated Time [us]") (.history-range) (:simulator :time))  ;Bottom
  (("Network") (0 .sites) (:net-output-load :busy save-sum))     ;Left
  (("Processing") (0 .sites)                                     ;Right
     (average (:evaluator-load :busy save-sum)
              (:operator-load :busy save-sum)))
  :find (update-history (:simulator :time) .sampling-interval)
  :show (recent-history (:simulator :time) .history-range 0))
```

Figure 9: System History Panel Specification

Line plot panels may have two independently scaled vertical axes. For the system history panel shown, the sum of network loads as indicated by the net-output components of the system is plotted against the left axis and the sum of the processing loads provided by the current average of the sums of the operator and evaluator loads is plotted against the right axis. Event time is plotted on the horizontal axis. The update-history function uses the component pointer to find the information previously saved for that component and records the current event time as the (:simulator :time) so that it may be used to display information correctly on the horizontal axis. The current sums of the evaluator loads and the operator loads measured by the system are stored in a record for the given event time (or a prior event time within the specified sampling interval) by the calls to the save-sum function specified as part of the *save* operation.

5.3 Self Scaling Line Plot Panels

Figure 10 illustrates both the self scaling of displays and the use of a display analysis operation. For this self scaling line plot panel, two pieces of data are collected for each operator in the system: the load on the operator, shown on the right axis, and the latency of the information it has most recently received. This last item is provided by the operator latency probe in two parts: (1) the interval between the creation of the information and its receipt by the net-input feeding the operator and (2) the interval between such receipt and the operator taking action on it. There are thus two curves plotted on the left axis. The specification stipulates a list for the left axis display. The elements of this list are the "net delay" and the sum of this measure and the "operator delay" monitored by the operator latency probe. Since both delays are non-negative, their sum must be at least as large as either one taken alone: the two curves may be superimposed but can not cross. The difference between the two curves is the incremental delay added by the operator.

The panel specification for the operator-network panel is shown in figure 11. In addition to transformations shown previously, an analysis function is stipulated for the *send* operation of the panel. The information saved from each of the probes sending :update messages to the panel is to be sorted from the greatest to the least values of the associated sum of delays described above. This information is to be saved as the operator latency rank and used as such to determine the position on the horizontal axis that the delay and load information will be displayed.

```
OPERATOR - NETWORK
       Latency & Operator Load
```
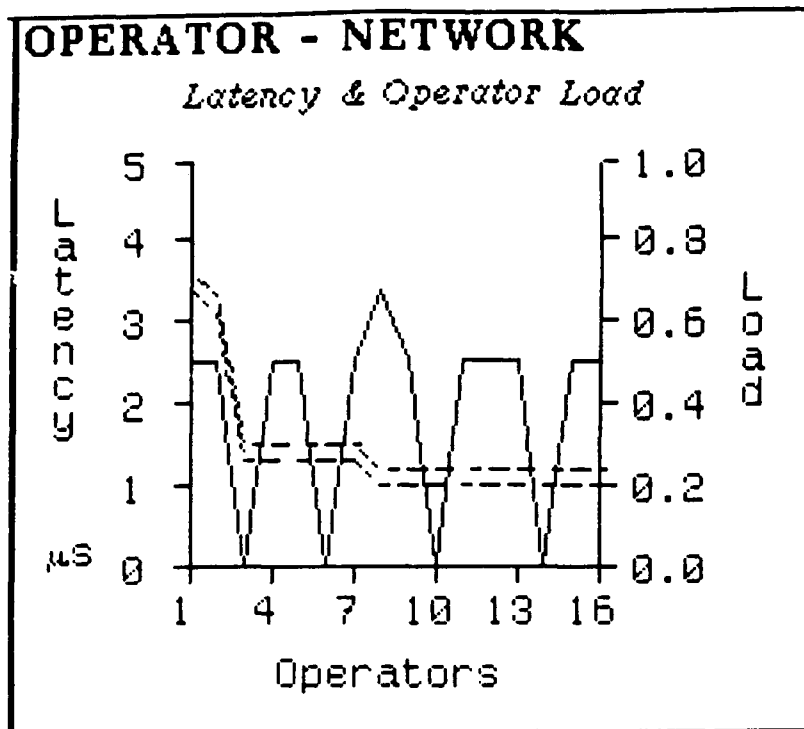

Figure 10: Self Scaling Line Plot Panel

```
'((("Operators") (1   ,sites) (:operator-latency :rank))
 ((("Latency" "us")) (0 nil) ;Second string: 90 degree baseline shift
   ((:operator-latency (:net-delay (+ :net-delay :operator-delay)))))
 (("Load") (0 1.0) (:operator-load :busy))
 :send (sort-arrays
          ((,#'> (:operator-latency (+ :net-delay :operator-delay))))
          ((:operator-latency :rank))))
```

Figure 11: Operator-Network Panel Specification

5.4 Boxes and Lines Panels

Perhaps the most intuitively satisfying of the types of panels available is the *boxes and lines panel*, a graphic representation of a circuit showing its components and their interconnections. An example of such a panel is shown the left part of figure 12. This class of panels uses information left behind by the structure editor when the circuit was defined. Its form is thus automatically generated. The position of the components ("boxes") and the connections between them ("lines") in the display are used to animate system operation. In the example shown, the shading (or color) of the boxes is used to indicate the availability of the *evaluators* in the simulated system as the simulation proceeds. Darkest shades indicate highest availability, that is, empty queues for utilization of the resource; lighter shades indicate lower availability, that is, longer queues. The lines between boxes indicate communication paths that are in use, that is, not ":free" at the time of the most recent *show* operation for the panel.

The panel specification for the *mapping panel*, an instance of a boxes and lines panel, is shown in figure 13. There are two specifications for the panel: one for the boxes and one for the lines. The specification for boxes in the panel stipulates that the availability of evaluators in the sites corresponding to the boxes displayed controls the shading of those boxes. The scale is defined to run from 0 to 1.0. The specification for lines in the panel uses the connection information reported for the net-output to determine line placement on the display. When the status is reported as :free, the connection information is dropped from the panel and the corresponding lines are removed.
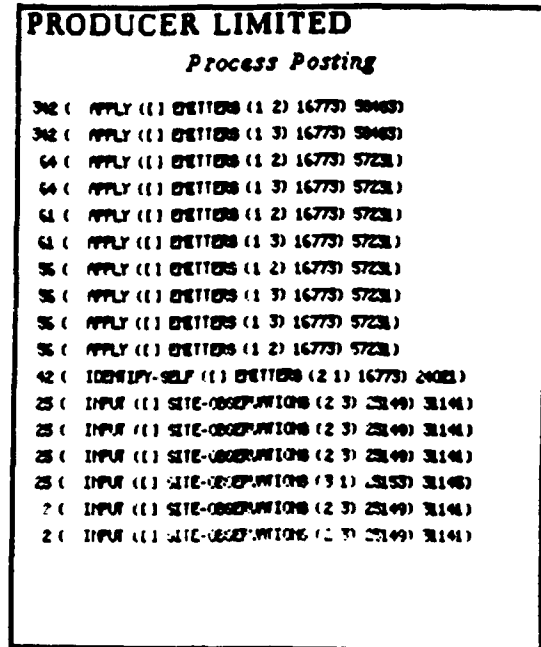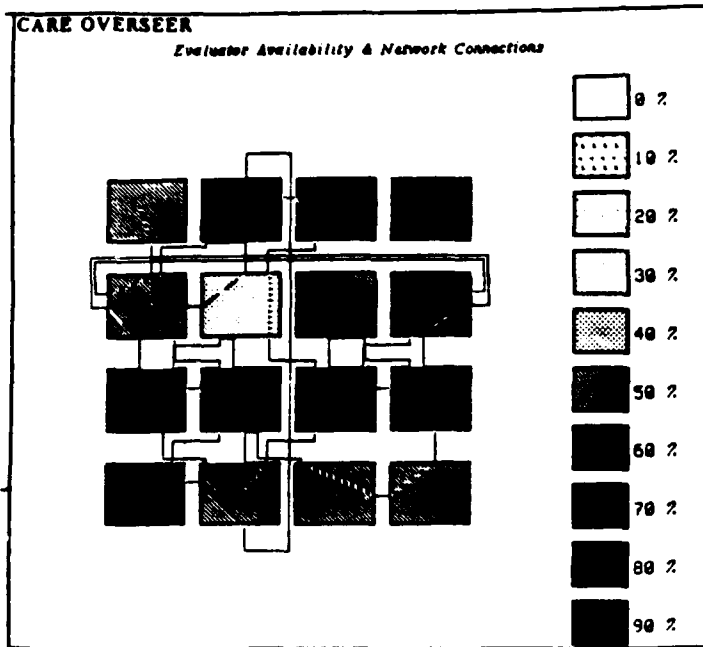
Figure 12:   Boxes and Lines Panel and Scrolling Text Panel

```
'((("Evaluator Available") (0 1.0) (- 1 (:evaluator-load :busy))))
'((("Packet Trace") nil (:net-output-connection :points))
 (("Packet Status") nil (:net-output-connection :status))
   :find (find-and-remove ,#'eq (:net-output-connection :status) :free)))
```

Figure 13:   Mapping Panel Specification

5.5 Scrolling Text Panels

Sometimes, the most appropriate way to display information is to show it as text. Based on a similar facility provided by the underlying Lisp system, the *scrolling text panel* provides a scrollable window into lines of text. In the right part of figure 12, the delay in each process execution while waiting for something to do, that is, the event time interval spent waiting for an appropriate task to appear on a certain stream of tasks, is shown together with the process that finally produced the awaited work. This information is sorted so that the text lines appear from the greatest stream waiting interval to the least.

```
'((() ("~4D ~A")
    ((fix (:stream-waiting :interval))    ;first field
     (let* ((origins (packet-origin (:stream-waiting :packet)))
            (origin (if (listp origins) (first origins) origins)))
         (remote-address-local origin))))    ;second field
    :send (sort-arrays ((,#'>) (:stream-waiting :interval))) nil))
```

Figure 14:   Producer Limited Process Panel Specification

The values and formats used for display in a scrolling text panel are defined much as in previously defined panels. Format control strings take the place of scale information. As usual, values are described by a list of forms, each one of which specifies the transformations to perform on information received from probes. The example specification in figure 14 shows the generality with which probe information can be incorporated in Lisp expressions

to produce transformation specifications. The information used to generate the value for the second field of the text display is based on the origin of the task packet that arrived on the stream the process was waiting for.

5.6 Noting Simulation Parameters

The CARE component models are parameterized through menu interaction as shown in figure 15 to allow easy variation of their performance characteristics relative to each other. Additionally, the site model parameterizes alternative routing strategies: *directed*, that is, blocking when progress can not be made toward the goal; *spiraling* around the goal if progress toward it is blocked; and *dithering*, that is, routing away from the goal even if only the last link towards it remains to be acquired. The rate at which each site accepts application data is also a parameter, the *data rate* and can be used by an application to control how hard it drives the simulated system.

```
┌─────────────────────────────────────────────────────────┐
│Simulation Parameters                                      │
├─────────────────────────────────────────────────────────┤
│Data Rate [µs]:                      25.0                  │
│Evaluation Override [µs]:            NIL                   │
│Stack Group Switch Override [µs]:    1.0                   │
│Process Block Creation Override [µs]: 4.0                  │
│Stack Group Creation Override [µs]:  20.0                  │
│Operator Word Touch Time [µs]:       0.2                   │
│Communication Cycles:                4                     │
│Routing:                  DIRECTED SPIRALING DITHERING     │
├─────────────────────────────────────────────────────────┤
│Exit □                       Quit □                        │
└─────────────────────────────────────────────────────────┘
```

Figure 15:   Parameter Menu

Many of the CARE parameters are specified as *overrides*. If not specified, the corresponding performance is taken as measured on the simulation machine. Thus, the *evaluation override*, that is, the time to perform an evaluation can be specified as non-nil in order to fix the time that each user evaluation will take. (This is useful in making runs repeatable for debugging). The time that it takes to switch context can be specified as the *stack group switch override*. Similarly, the time to create a process control block and a stack context for that process can be taken as given rather than measured by specifying respectively the *process block creation override* and the *stack group creation override*.

The time required for operator execution is modeled in terms of the number of words the operator must manipulate in handling a given message. The manipulation time per word is specified by the *operator word touch time*. Lastly, the performance of the communication subsystem is specified as *communication cycles*. This is done in terms of the minimum number of evaluator data path clock times (that is, event times) required for a 32-bit word to pass a given point in the network. Thus the parametric specification, "4 communication cycles", dictates that 8 bits may cross such a boundary each time the evaluator passes through one event time. If the communications path were narrower or the base communication clock rate were lower, a higher number would be specified.

```
NOTES:
6/25/86 00:54:40 32 DIRECTED Cycles, Acceleration 2, Creation 2000µs, Switch 250µs, Evaluation 25µs, Data 15µs
```

Figure 16:   Annotation Panel

The last example of SIMPLE panels is the annotation panel as illustrated in figure 16. This

is used to (automatically) record the date, time, and parameters of the simulation run as well as any other information the user chooses to keyboard into it.

5.7 An Instrument Screen

All these panels are put together in an instrument screen according to a set of layout constraints manipulated by the underlying window system. The finished screen might look like figure 17. The instrument screen is redrawn at a rate set by the user. By experience, it is often better to update the screen at a frequency low enough to let the user interpret each screen comfortably than at the maximum rate possible. This approach also restricts the computing resources consumed by the instrumentation system. More focused approaches to controlling instrumentation load on the system include the ability to freeze selected panels and disconnect selected probes during a simulation run.
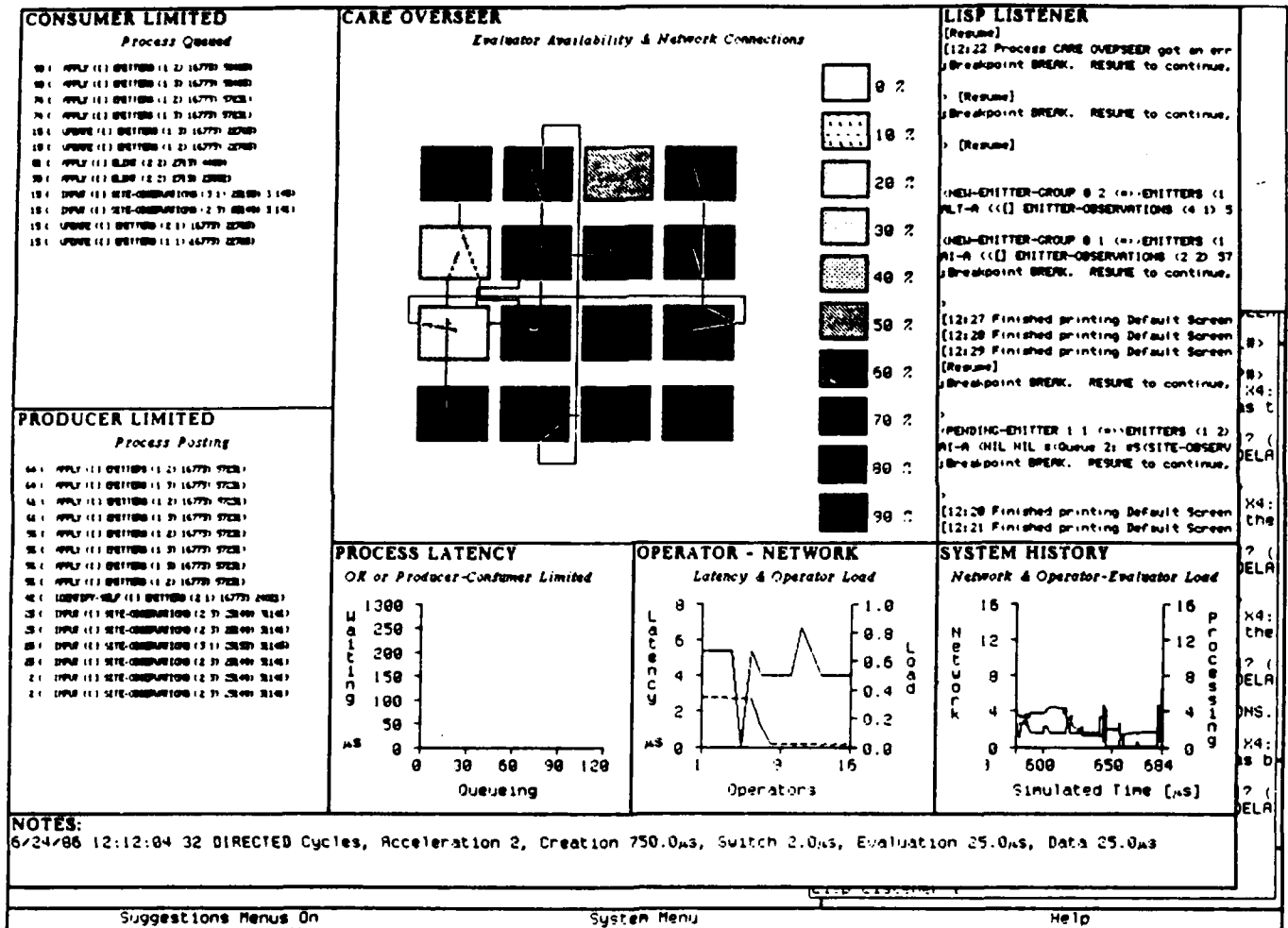
Figure 17: Overseer Instrument

6 USING PROGRAM DEVELOPMENT TOOLS

The SIMPLE/CARE simulation system is integrated into the underlying Lisp machine program development environment. The objects and data structures at both the component model and application language interface have abstraction interfaces that provide summary

state information when they are displayed in text form. These text abstractions are "mouse sensitive" in the development machine environment and so can be inspected at successively finer levels of detail as desired.

In figure 18, the net-output components of the site at grid coordinates (3 2), the particulars of the net-output on the east side of the site (that is, net-output-3), and a summary of all the sub-components of the site at (3 2) are being inspected. This same kind of view into the progress of a simulation is provided in the debugging process and may, as shown in figure 19, refer to the conceptual entities of the application that is driving the simulated system.
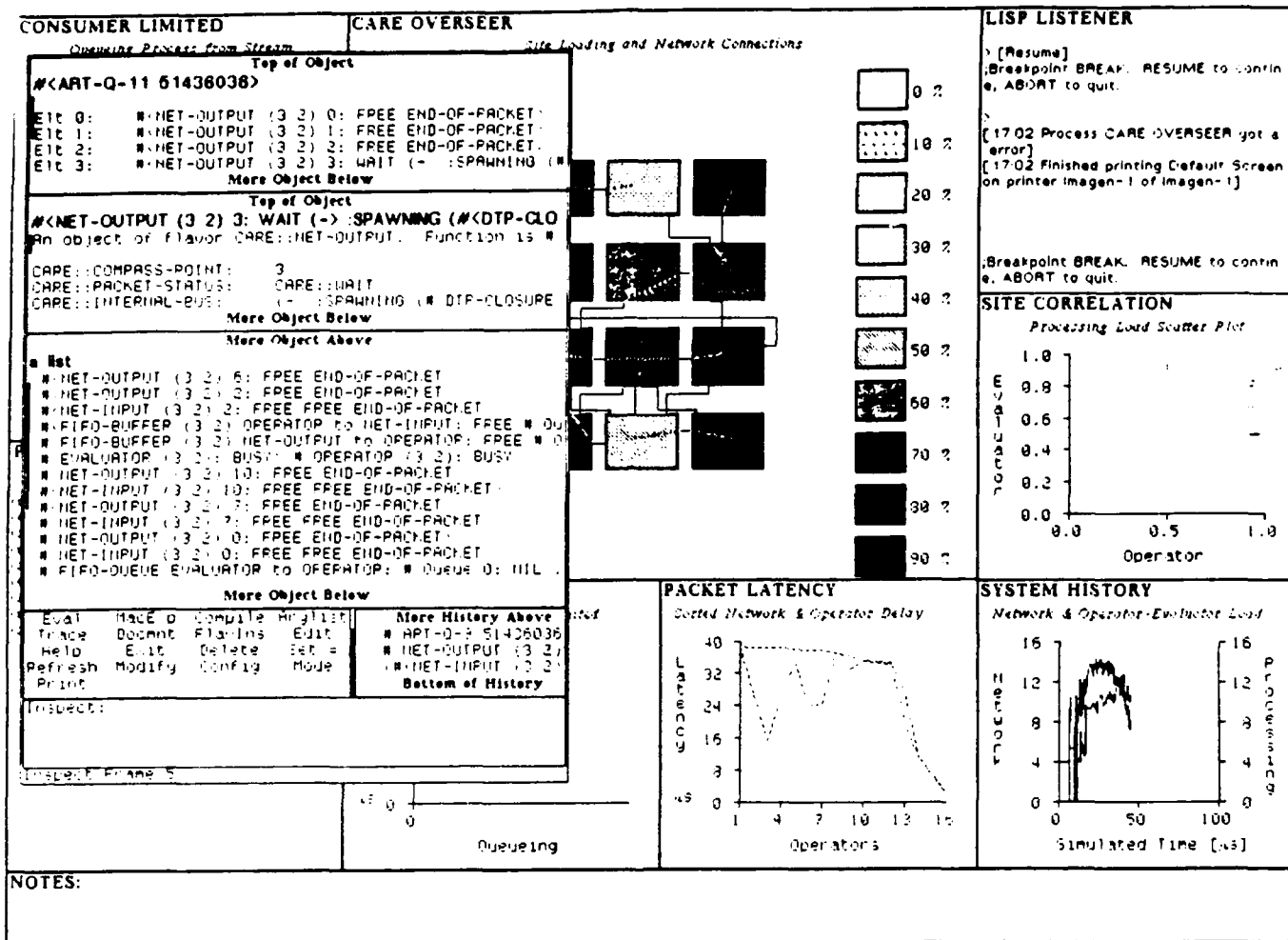
Figure 18: Inspecting Simulated Components

In the example shown in figure 19, a distributer process running on the evaluator at site (1 1) has made an improper call on the update-locale function during execution of its :start method. It might have been appropriate to investigate this situation in terms of the modeled components. That could be done, for example, using the debugger to inspect the evaluator component, its enclosing site, related net-output components, or whatever else at the component model level seemed relevant. In this case, what was done was to use a few mouse clicks to indicate interest in the source file for the distributer :start method generating the problem. It was brought up for review and control was then transferred to an editor using the underlying program development environment as shown in figure 20.

Because of the implementation system chosen for the realization of SIMPLE/CARE, at any point in the simulation, procedures either in the application or in the component models can be modified, incrementally recompiled (within a few seconds), and be made effective for all

calls on them -- even those in the interrupted stack frame. Thus simulation execution can be backed up to some previous point in the stack frame and retried (given that intermediate side effecting code, if any, is safely re-executable).

CONSUMER LIMITED | CARE OVERSEER | LISP LISTENER
Process Queued | *Evaluator Availability & Network Connections*

Top of Object

#<DISTRIBUTER -41776256>
An object of flavor DISTRIBUTER. Function is #<EQ-HASH-ARRAY (Funcallable) 3500637>

ACKNOWLEDGEMENTS: (^ (1. 1.) (=> DISTRIBUTER ACKNOWLEDGEMENTS 1573. 0 0))
REQUEST-STREAM: (^ (1. 1.) (=> DISTRIBUTER DISTRIBUTER-REQUESTS 1573. 0 0))

Bottom of Object

Top of Args for Current Frame	Top of Locals/Specials for Current Frame
Arg 0 (.OPERATION.): :START	Local 0 (COUNT): 1
Arg 1 (SERVICE): #'SIN	Local 1: #<DTP-LOCATIVE 22166536>
Arg 2 (SERVERS): 20.	Local 2: NIL
Arg 3 (FUTURE): (^ (2. 2.) (=> REQUESTOR REQUESTS-FU	Local 3 (THE-SITES): NIL
Arg 4 (LOCALE): 'NIL	Local 4 (OBJECT): NIL
Bottom of Args	Local 5 (THE-CLOCK-NOW): NIL
	More Locals Below

PRODUCER LIMITED
Process Post

Top of Stack
(EH:INVOKE-DEBUGGER #<EH:ARG-TYPE-ERROR :CONDITION-NAMES (EH:ARG-TYPE-ERROR ERROR CONDITION SYSTEM:WR
(SIGNAL-CONDITION #<EH:ARG-TYPE-ERROR :CONDITION-NAMES (EH:ARG-TYPE-ERROR ERROR CONDITION SYSTEM:WRON
(EH:FH-APPLIER-NO-RESTART SIGNAL-CONDITION (#<EH:ARG-TYPE-ERROR :CONDITION-NAMES (EH:ARG-TYPE-ERROR E
(EH:FOOTHOLD)
(UPDATE-LOCALE 'NIL)
-(#<DISTRIBUTER -41775256> :START #'SIN 20. (^ (2. 2.) (=> REQUESTOR REQUESTS-FUTURE 273. 0 0))...
((:INTERNAL FLAVOR 0.) (:START #'SIN 20. (^ (2. 2.) (=> REQUESTOR REQUESTS-FUTURE 273. 0 0))...)
(FUNCALL #<DTP-CLOSURE -36264730> (:START #'SIN 20. (^ (2. 2.) (=> REQUESTOR REQUESTS-FUTURE 273. 0 0
(CARE:USER-EVALUATE (= #<DTP-CLOSURE -36264730> #<DISTRIBUTER -41775256> 1309.) 1313.)
((:METHOD CARE:EVALUATOR :APPLYRULES) :APPLYRULES (:TRUE (0:BR-VALUE #<EVALUATOR (1. 1.): BUSY> CARE:
(#<EVALUATOR (1. 1.): BUSY> :APPLYRULES (:TRUE (0:BR-VALUE #<EVALUATOR (1. 1.): BUSY> CARE:IN-STATUS

More Stack Below

Examine	Search	Report	Resume	Bk Next	**Top of History**
Error	Arglist	Exit	Petry	Bk Exit	#<Stack-Frame UPDATE-LOCALE PC=55>
Inspect	Quit	Edit	Resume	Bk All	#<Stack-Frame (METHOD DISTRIBUTER START) PC=123>
Help	Flavins	ModInsp	Return	Step	#<DISTRIBUTER -41775256>
Dbg SG			Modify	Stay	**Bottom of History**

>:TPAP: The first argument to CLI:AREF, (:LOCALE (1. 3.)/0 (2. 1.)/0 (2. 3.)/0 ...), was of the wrong typ
e.
The function expected an array.
>
Type or mouse a function to edit (NIL aborts, T to edit nothing):
Type or mouse a message name for #<DISTRIBUTER -41775256>:

NOTES:
/23/86 10:43:10 12

Debugger Frame 3

Figure 19: Debugging A Simulation

```
(DEFMETHOD (DISTRIBUTER :START) (service servers future locale)
  "Request creation of servers and continue on to :request to wait"

  (let ((the-sites (loop for count from 1 to servers collect
                         (locale-site (update-locale locale)))))
    (let ((object (reference self)))
      (without-clock
       (format *output-stream* "~&~A [distributer] ~A"
               (send (remote-site object) :location)
               (mapcar #'(lambda (site) (send site :location)) the-sites)))

      (posting request-stream to future as :requests-stream)
      (spawning ((flavor 'server) :start service acknowledgements) on the-sites
                as service)
      (applying (:request) on object as :distributer-requesting)   ;for continuation
      object)))

(DEFMETHOD (DISTRIBUTER :REQUEST) ()
  "If there's an available server and a request, pass out request; loop"
  (loop
    for response = (accept (first-posting acknowledgements))
    for (value clients tag) = (accept (next-posting request-stream))
    do (posting value to (posting-clients response)
               for (cons acknowledgements clients) as tag)
       (next-posting acknowledgements)))   ;done with this acknowledgement

(compile-flavor-methods distributer)

(DEFFLAVOR SERVER ((service (new-stream :server-requests))) ())

(DEFMETHOD (SERVER :START) (operation acknowledgements)
  "Send back notice of availability"
  (let* ((object (reference self))
         (the-site (remote-site object))
         (the-location (send the-site :location)))
    (without-clock
     (format *output-stream* "~&~A ~A" the-location operation))
    (posting 'initialized to acknowledgements for (list service) as the-location)
    (applying
     (:request operation the-location) on object as :server-continuation)
    object))
```

Figure 20: Changing Application Code

7 CONCLUSIONS

The goals of simulation flexibility and simulation environment completenes. have been dealt with in the ways described throughout this paper. In summary, the system is flexible in that it supports:

- Arbitrary data types and lengths in simulation. The information whose flow and creation is controlled by simulated components may be of arbitrary complexity -- from numbers and keywords to procedure bodies and execution environments.

- Instantaneous effect of definition change at both the application and component modeling level (even during a simulation run).

- A broad range of instrumentation customization. Customizations may involve arbitrary expressions for probe data transformations, many to many probe to panel mappings, information from summary analyses on one panel's data included in another, and control of what state is saved and for how long.

- Separation of probe and component definitions to facilitate their independent modification.

- An application language interface that is easily extended or changed without recasting the information flow control described by the component behaviors.

While there is always room for additional capability[6], SIMPLE/CARE is a usefully complete system. It now includes:

- Supplied components for a network .nultiprocessor simulation with many of their parameters customizable by menu interactions.

- A hierarchical structure editor that currently provides automatic grid and torus composition operators. (Automated composition of richer topologies, such as hypercubes, has been provided for in the basic design).

- A rule language that supports a synchronous design style without incurring the overhead of (naive) synchronous simulation.

- Method invocation for functional simulation that is integrated into the behavioral simulation rule system and which provides for operations by and on both local and hierarchically related components.

- Method specification design aids provided by the underlying program development environment (for example, method dictionaries and quick access to method sources from the debugging system).

- An evolved set of panel templates providing sorted, scrollable text lines as well as self and fixed scaling, "two and a half" dimensioned, history sensitive displays which may be scatter plots, strip charts, line graphs, intensity maps, and signal animations.

We set off to build a multiprocessor simulation system with performance adequate for the understanding of multiprocessor systems executing significant applications. The SIMPLE/CARE simulation system has been used to study the operation of "expert systems" of respectable size [2]. Depending on instrumentation load, these studies have involved simulation runs from 20 minutes to several hours each. While faster would surely be better, performance has proven adequate to these needs.

---

[6] A *histogram panel*, for example, is just now being added to the system

8 ACKNOWLEDGEMENTS

References

1. Brown, Harold, Christopher Tong, and Gordon Foyster. "PALLADIO: An Exploratory Design Environment for Integrated Circuits." *IEEE Computer 16* (December 1983).

2. Harold D. Brown, Eric Schoen, and Bruce A. Delagi. An Experiment in Knowledge-Based Signal Understanding Using Parallel Architectures. Tech. Rept. STAN-CS-86-1136 or KSL-86-69, Stanford University, October, 1986.

3. Greg Byrd, Russell Nakano, and Bruce Delagi. A Point-to-Point Multicast Communications Protocol. Tech. Rept. KSL-87-02, Knowledge Systems Laboratory, Stanford University, January, 1987.

4. Daniel Weinreb and David Moon. *Lisp Machine Manual*. Symbolics, Cambridge, MA, 1981.

Appendix F

LAMINA: CARE APPLICATIONS INTERFACE

by

Bruce Delagi, Nakul P. Saraiya, and Gregory T. Byrd

Knowledge Systems Laboratory
Computer Science Department
Stanford University
Stanford, California 94305

Table of Contents

List of Figures

ABSTRACT

LAMINA provides extensions to Lisp for studying expressed concurrency in functional programming, object oriented, and shared variable styles of computation. The implementation of the support for all three computational styles is based on the common notion of a *stream*, a datatype which can be used to express pipelined operations by representing the promise of a (potentially infinite) sequence of values. A pipelined algorithm to provide the sorted order of sequences of set elements is presented in the functional, object oriented, and shared variable programming styles for comparison.

In addition to demonstrating that a common set of primitives based on the notion of a stream is adequate for support of all three styles mentioned, LAMINA illustrates the means by which software pipelines may be managed and the means by which dynamic structure creation, relocation, and reclamation may be localized in a multiprocessor system.

Algorithms and applications written in LAMINA may be run on the SIMPLE/CARE simulation system in order to study their execution on alternative multiprocessor architectures. This has been done for two "expert system" applications and linear speedups over the range from one to eighty processors have been measured using LAMINA.

1 Streams, Values, and References

The SIMPLE/CARE multiprocessor simulation system [4] supports an applications programming interface, LAMINA, which currently is built upon Zetalisp [14]. LAMINA has been used as the basic programming language for two "expert system" application developments [2, 10] demonstrating significant speedup with increasing numbers of processors. LAMINA includes primitive mechanisms and language interface syntax for alternative approaches to the expression and management of concurrency and allows their relative performance to be measured on a common ground.

Functional, object oriented, and shared variable programming styles are all directly supported by LAMINA. The support provided for these styles is described in sections 2, 3, and 4 respectively. Section 5 describes some general utility functions. Primitives implementing the underlying mechanisms are described in an appendix. A second appendix lists the constructs of LAMINA and provides references into the body of the paper for details. The remainder of this section consists of background material describing how the values of one computation are passed to another and how the address space of an application is spread across the processors of a system in LAMINA.[1]

1.1 Futures and Streams

Futures [5, 6] and *streams* [8, 11] provide the common ground between functional, object oriented and shared variable programming in LAMINA. They are fundamental to the LAMINA functional and object oriented programming regimes for parallel programming and, since they are the only mutable items passed as references (rather than structure values) between potentially concurrent computations in LAMINA, they are also used to build the mechanisms for shared variable computation.

Futures and streams represent promises for values. We can arrange for promises for values, that is, their futures, to be used as placeholders in a computation while the values themselves are being *eagerly* [8] produced by concurrent evaluations for consumption as available. Extending this idea, we can define a *stream* as an abstract data type which is a placeholder representing a sequence of eagerly produced but potentially unavailable values.

Some operators do not require the actual values promised by a stream or future in order to perform their work. For example, a constructor may create data structures that include streams as structure elements. The creation can be accomplished without accessing any of the promised values that the streams represent; referencing streams as placeholders is sufficient. Further, streams, as sequences of potentially unavailable but eagerly produced values, can be used to build pipelines of computation connecting the producers and consumers of such values.

Streams may be arguments to or the results of function application. In LAMINA, streams are a primitive data type developed for use in an object oriented programming style and futures are a specialization of streams that represent only a single (potentially unavailable) value as required for the functional programming style. Streams and futures are always passed as references. In the remainder of the paper, the term *stream* or *future* is equivalent (respectively) to a *reference* to a stream or a future.

1.2 Processor Address Spaces and Multilevel Allocation

In LAMINA, structures of arbitrary complexity can be supplied as a value of a stream or future either local or remote to the processor address space in which the structure was generated. Internal pointer references within copies of such structures are adjusted (for address relocation) as the copies pass between the originating processor address space and the processor address space of the stream that represents the promise for the values so supplied. External pointer

---

[1]Footnotes in the paper generally deal with details, conventions, or implementation issues that can be skipped on first reading.

references included in structures passed between spaces are restricted in LAMINA to locations in global *dynamic* or *static* address spaces as shown in figure 1. Statically allocated structures are not relocatable or reclaimable and may be regarded as cacheable and immutable. Thus, they may be globally referenced without a need for access coordination.
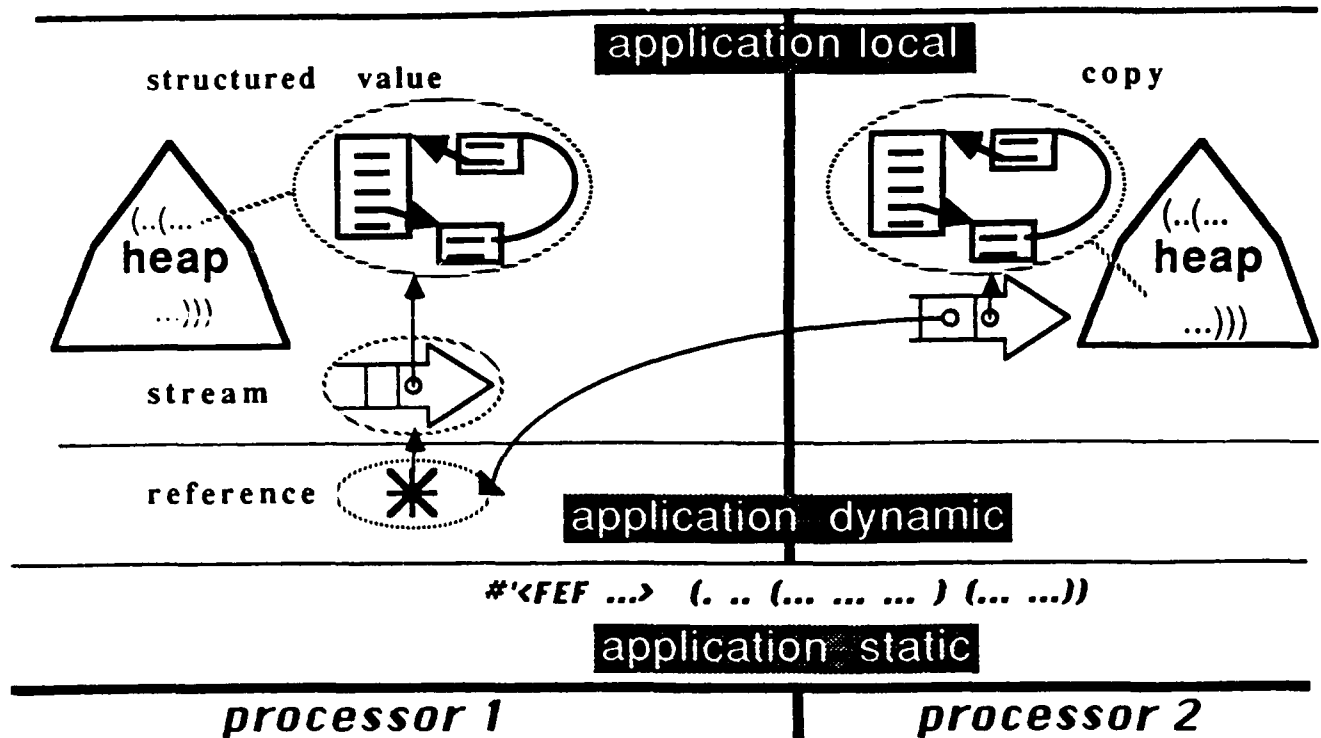
Figure 1: LOCAL, DYNAMIC, & STATIC ADDRESSES

When values are passed between processor address spaces the structure representing the value, that is, the *structure value*, is recursively copied until a data structure is produced which has the same form and internal relationships as the original value but which holds only: *static references* (to code bodies and other structures in static space), *dynamic references* (to streams or other structures) in dynamic space, *internal references* (to subcomponents of the structure value), and *self-referentials* (for example, numbers and characters).[2] Copying of a structure value might be done asynchronously with evaluation of the user application, s if changes are to be made in the structures encompassed by a structure passed between address spaces, independent copies of such structures should be formed.

An example of values and references passed between processor address spaces is shown in figure 1. One of the values of the indicated stream in the application's processor 2 *local* address space is a copy of the structure value in the application's processor 1 *local* address space. Both structure values are heap allocated from independently managed heaps in separate local spaces. Allocation, relocation, and reclamation for each given heap may be done asynchronously based on just the information in the associated processor address space. The other value shown for the indicated stream in figure 1 is a *reference* (in this case, to the original structure value) allocated in the application's *dynamic* space. Because the reference and its associated structure value are allocated within a single processor, relocation of the locally allocated structure value can be done locally and asynchronously. Relocation of the reference, however, must be globally coordinated. Statically allocated structures are not relocated or reclaimed.

---

[2] As a current implementation restriction, lexical closures [12] passed between processor address spaces may only be made over free variables whose values are references or self-referentials items and not structures that contain them.

References to streams are allocated in dynamic space and streams are accessed by reference. A stream reference, therefore, may only be relocated (for example as required by a compacting garbage collector) through globally synchronized operations affecting all computations that could access that stream. This global synchronization can be expensive and involve subtle low level implementation considerations. Expectations about the expenses involved in correct global syncronization[3] led the design of LAMINA to a multi-level allocation scheme described below.

The cheapest approach to allocation (and deallocation) of memory for dynamically created structures is *stack-based* (and local). However, the benefits of stack-based operation come at the cost of a prescribed order of deallocation. Additionally (at least for the commonly used memory management enforced stack limit schemes), stack-based operation entails a minimum storage commitment that is significantly larger than the rest of the execution environment for each highly concurrent, small granularity evaluation expected in LAMINA programs. Stack based allocation is used in LAMINA whenever references to structures with dynamic extent [13] are known to be entirely within a given sequential computation.

The next cheapest approach, for references that are local with indefinite extent [13], is heap based allocation in *local* space. Since such references are confined to a single processor address space, they may be relocated asynchronously with operations on other processors and memories or in the network connecting the components of the multiprocessor system.

Finally, as the most expensive approach, global references may be made to dynamically allocated references (which must be relocated under a global synchronization scheme). Allocation in *dynamic* space is done independently by each processor and each allocation is distinct. Operations involving dynamically allocated references are handled by the processor (or memory controller) associated with the reference. The referents for such references are mutable and may be viewed as uncacheable.

References to locally allocated structures can also be passed between processor address spaces by encapsulating them in dynamically referenced structures, that is, streams. By this indirection, pointers to selected locally allocated structures are held locally (and may readily be relocated) but a means is provided to reference them in other processor address spaces.

The multi-level allocation scheme just described creates references passed between processor address spaces (with the attendant synchronization expenses) only as necessary. The remainder of this section describes the syntax for creating and accessing such references.

1.3 Reference Creator and Accessor Functions

When a locally allocated data structure needs to be passed between potentially concurrent computations as a reference rather than as (a copy of) its value, the form (reference *item*) returns a reference for the value of the item.

The *site* of a reference, that is, the CARE processor (or memory controller) on which it was created, may be determined by executing (reference-site *reference*). The value returned by calls to this function is a *site refrence* that may be used to specify sites as required as parameters of other LAMINA functions.

Finally, references can be tested to determine whether they refer to the same item by the function reference-eq, a function that accepts two references as arguments and returns a non-nil value if they refer to the same item.

---

[3]For example, in a shared memory system with asynchronous writes to memory, a request to change the contents of a location in dynamic space so that it points to a stream in a given semispace of a compacting garbage collector may have been in transit to a memory controller when evacuation of that semispace was requested. The evacuation must be delayed somehow until all such requests either in transit or queued anywhere in the system have been processed. Shared memory systems with synchronous writes delay *all* processor operations on shared variables until the memory request can first traverse the network between processors and memories (or other caches), then be queued and serviced in the memory (or other cache) controllers, and finally traverse the network back to the processor.

2 Functional Programming

Perhaps the style of computation most readily treated as concurrent is that of functional programming. LAMINA supports concurrent programming using this style by providing means (1) to spawn computations that will provide values to futures and (2) to accept such values in a computation -- scheduling the computation when they are available. The constructs defining the LAMINA interface for functional programming are:

- (future *form*) spawns execution of a *lexical closure*, that is, a procedure body to execute a given form together with an environment (determined by the rules of lexical scoping) in which to do the execution [13]. This closure is executed (eagerly) on a randomly selected site. A future which will contain the value of the computation when it is available is immediately returned.

- (with-values *future-bindings forms*) spawns an evaluation on the local site to execute the closure corresponding to the *forms*. The evaluation is done within an environment that includes bindings for given variables to the values available for the indicated futures. The evaluation is deferred until all of the indicated futures have values that are not themselves futures. The immediate result of executing a with-values form is a future whose value will be supplied by the deferred evaluation.

Each element of a *future-bindings* list is itself a list: (*binding-pattern future-specifier*). If evaluation of a future specifier in a with-values construct produces a value other than a future, the future specifier is coerced to be a future holding that value. After all specified futures have values (which are not themselves futures), the values of each of the futures are *destructured* [13], that is, the values are treated as list structures and the elements of these list structures are used to bind corresponding variables in a binding pattern of arbitrary depth. These bindings will be included in the environment in which the spawned computation is executed. Only with-values can be used in LAMINA to reduce futures to values. Values of futures are never taken as an ancillary consequence of any other operation.

The results of the evaluation spawned by with-values are returned as a future which will receive the value of the spawned computation. The spawned evaluation that is created by a with-values construct is treated as the continuation [12] of the computation in which it is found and, as such, captures all stack allocated values required to execute that computation. Thus, each spawned computation may be viewed as running to completion; its continuation, if any, is an independent spawned computation.

Because all spawned computations run to completion (unless they are preempted by system level operations), the stack of the executing processor is (generally) left clear and any space allocated for it may be reused by the next computation on that processor. By this means, the advantages of stack-based operation are retained without incurring the space penalty discussed in section 1.2. The costs of heap allocation are incurred only as needed.

To illustrate the use of the LAMINA functional programming interface, the implementation of a (quicksorting) algorithm to associate ordering information with the numerical values of the elements of sets supplied as input is shown in figure 2. The serial and parallel implementations may be compared by contrasting the definitions of the functions order0 and order1.

The input to the ordering functions is sets of numbers to be ordered. Elements of a set are the sequential elements of a list before a separator token (which is nil). The sets (including their separator tokens) are concatenated to form the input list. The output is a list with each ordered set represented by successive elements of a list and separated from other ordered sets by nil tokens. The sets follow each other in the output in the same order in which they appeared in the input. For example, the input list (7 9 4 nil 5 3 8 nil) would result in the output (4 7 9 nil 3 5 8 nil). Thus the information concerning the ordering of the elements of a set and the identity of that set is implicit in the output.

In order0 and order1, the result of ordering nil is nil. If the input list is not nil, the

```lisp
(DEFUN ORDER0 (input-list)
  "Serial quicksort to order elements of input sets"
  (if (null input-list) nil
      (let ((pivot (car input-list)))
        (if (null pivot) '(nil . .(order0 (cdr input-list)))⁴
            (destructuring-bind (smaller larger rest)
                                (part1 pivot (cdr input-list))
              (let ((ordered-smaller (order0 smaller))
                    (ordered-larger (order0 larger))
                    (ordered-rest (order0 rest)))
                '(.@ordered-smaller .pivot .@ordered-larger
                  . .ordered-rest)))))))

(DEFUN ORDER1 (input)
  "Without pipelining: recursively spawn ordering partitioned input sets"
  (with-values ((input-list input))
    (if (null input-list) nil
        (let ((pivot (car input-list)))
          (if (null pivot)
              (with-values ((rest (order1 (cdr input-list))))
                '(nil . .rest))
              (destructuring-bind (smaller larger rest)
                                  (part1 pivot (cdr input-list))
                (with-values ((ordered-smaller (future (order1 smaller)))
                              (ordered-larger (future (order1 larger)))
                              (ordered-rest (future (order1 rest))))
                  '(.@ordered-smaller .pivot .@ordered-larger
                    . .ordered-rest))))))))

(DEFUN PART1 (pivot input-list)
  "Serial: add elements from input list sets into one collection or other"
  (let ((input (car input-list)))
    (if (null input) '(nil nil .input-list)
        (destructuring-bind (smaller-part larger-part rest)
                            (part1 pivot (cdr input-list))
          (if (> input pivot)
              '(.smaller-part (.input . .larger-part) .rest)
              '((.input . .smaller-part) .larger-part .rest))))))
```

Figure 2: FUNCTIONAL ORDERING

first element of that list is used as a pivot. If that element is nil, it is a separator token. The result then is the separator followed by the result of ordering the rest of the list. If the pivot element is not nil, it is assumed to be a number that is used by part1, a serial partitioning function which returns a list of three results: the (unordered) elements of the current set smaller than the pivot, the (unordered) elements of the current set larger or equal to the pivot, and the remaining elements of the input.

The function order1 spawns executions to apply itself to each of the three sublists returned by part1 to order them. It then waits for the results. When these are available, it appends the ordered sublist of elements that were smaller than the pivot to the list formed by the pivot, the ordered sublist of elements that were not smaller than the pivot, and the result of ordering the rest of the sets in the input.

The operation of order1 is characterized by much waiting for the results of spawned

---

⁴Due to printing limitations, the backquote character will appear as '. Inclusion of a comma in the form introduced by a backquote will disambiguate the quoting character.
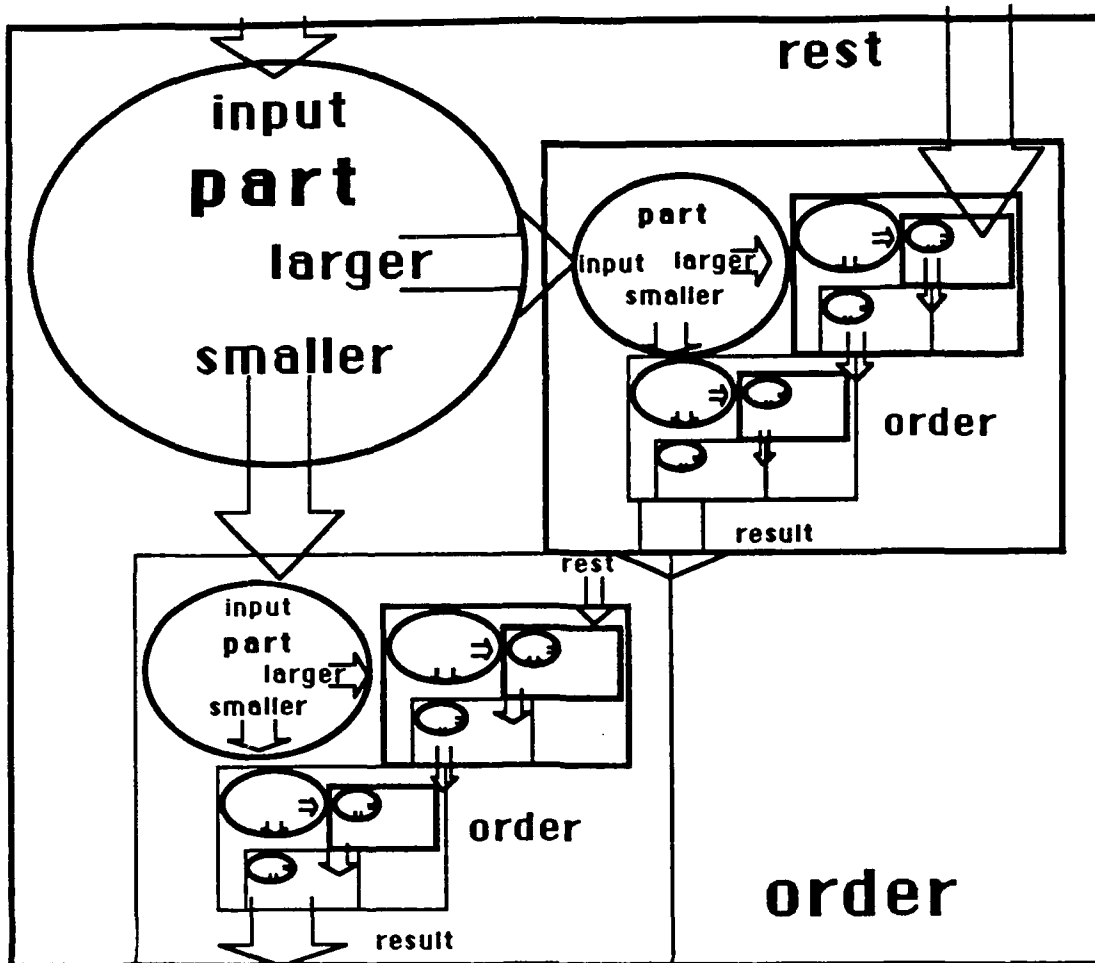
Figure 3: ORDERING PIPELINE

computations. The pattern of execution is to spawn a set of computations -- using **future** constructs -- and immediately wait for all their values to be produced -- using **with-values** constructs. This waiting represents serialization due to data dependencies and can significantly limit the concurrency of an algorithm. If, instead, computations can be handed just what they each require to get started (with promises for the rest), they can be pipelined as computation assembly lines, each station operating on a piece of the input from upstream producers and delivering a piece of the output to downstream consumers.

A schematic view of a pipelined ordering algorithm is shown in figure 3 while the code is shown in figure 4. The schematic is a recursive drawing terminating in a number of ordering computations -- one leaf for each element and separator token in the sets of elements to be ordered. Each non-leaf node of the ordering tree partitions its input by sending each input element it receives (from its upstream parent) to one of its two downstream children. The smaller child was created such that its result is used as the result that the parent was asked to produce and the rest of its input is the result of the larger child. The larger child was created so that if it is a leaf (that is, if it has nothing to order), its result will be the rest of the items given to the parent. The rest of the items seen by the largest descendent of the smaller child is the result produced by the smallest descendent of the larger child. Thus, using an approach similar to the use of *difference-lists* in logic programming [11], the results of the leaf elements are tied together to produce the result of the ordering tree.

The first input a child receives will establish the pivot for partitioning unless it is the separator token, **nil**. If it is **nil** and there is more input, the child returns **nil** as the first part of the result together with a promise for ordering the rest of its input followed by those

```
(DEFUN ORDER2 (input-future &optional rest-pair)
 "Future pipeline: rest and input pair (or its future) => ordered pair"
 (with-values (((pivot . rest-input) input-future)) ; Coerce value
  (if pivot      ; Spawn partitioning and get promises for first elements
     (with-values (((smaller-future larger-future)
                    (future (part2 pivot rest-input))))
        (let* ((ordered-larger-future ; Spawn order larger
                (future (order2 larger-future rest-pair)))
               (ordered-larger-pair
                '(,pivot . ,ordered-larger-future)))
          ;; Continue ordering smaller
          (order2 smaller-future ordered-larger-pair)))
     (if (null rest-input) rest-pair
         '(nil . ,(future (order2 rest-input rest-pair)))))))))

(DEFUN PART2 (pivot input-future)
 "Produces (<future> <pair>) or (<pair> <future>) for (<smaller> <larger>)"
 (with-values ((input-pair input-future))          ; Coerce value
  (if input-pair ; Destructure pair as (value . future)
     (destructuring-bind (input-value . rest) input-pair
       (if (null input-value) '(nil  (nil . ,rest))
          ;; Spawn continuation of this partitioning
          (let ((future-part (future (part2 pivot rest))))
            ;; and get futures for destructured value of continuation
            (let ((smaller-future
                      (with-values
                         ((value future-part)) (first value)))
                  (larger-future
                      (with-values
                         ((value future-part)) (second value))))
              ;; Return list: (<future> <pair>) or (<pair> <future>)
              (if (> input-value pivot)
                 '(,smaller-future
                    (,input-value . ,larger-future))
                 '((,input-value . ,smaller-future)
                    ,larger-future)))))))))
```

Figure 4: PIPELINED FUNCTIONAL ORDERING

values larger than anything in that input. If there is no more input, it just returns promises for the results of its larger relatives, that is, the rest-pair.

The receipt of a separator token while partitioning indicates that all the elements of a set to be ordered have been received. A terminator, nil, is passed to the smaller child and a separator followed by the rest of the unordered input (if any) is passed to the larger child.

The code for this example is written assuming that each stream can only hold one value, that is, streams are restricted to be simple futures. In the example, sequences of values are represented by pairs consisting of a value and a future for the rest of the sequence. The value of the future, when available, is a pair which itself consists of a value for the next element in the sequence and a future for the rest of the sequence. The consequence of this approach is that many short lived dynamic references are created (so that each element of the sequence has an independent reference) and then abandoned. Reclaiming the space allocated for them requires global synchronization as discussed in section 1.2.

Relaxation of the single value assumption for structures representing unavailable values -- as well as extension of LAMINA to an object-oriented programming style -- is discussed in the following section.

3 Object Oriented Programming

In LAMINA's object oriented programming interface, an object encapsulates related state variables and is referenced throughout an application by that object's Self-Stream, a stream (whose reference is in *dynamic* space) which is one of the object's state variables. Objects are allocated in *local* space as described in section 1.2. To perform operations on an object, potentially involving and modifying its state variables, a *task request posting* consisting of a *task selector* and associated parametric values for the operation is *sent to*, that is, provided as one of the values of the self-stream for that object. Each of the task request postings that provide the values for the self-stream of a object is taken in turn from that stream and serviced by that object.

Task request postings are serviced atomically in the context of an object. Executions specified by such request postings are done without visible partition with respect to other operations on that object: operations on any given object will not be interleaved. Each operation is thus defined to be *independently atomic*.

All the operations on an object done as specified by the requests are taken in turn from the object's self-stream. Each operation runs to completion. If an operation on an object is preempted (due, for example, to page faulting, schedule quanta lapse, or error condition), no other operation on that object will be started before the preempted operation is completed. However, operations on other objects may proceed normally. A stack is maintained for each preempted operation.

3.1 Sending a Task Request

Sending a task request in LAMINA is non-blocking and thus pipelined operations on objects are directly accomodated. The information required to accomplish a task is either passed with the request or is included in the state variables of the object. In an object oriented programming style, state is localized in objects and is not referenced otherwise. Arbitrarily structured values, however, may be sent in task request postings between lamina objects as (copied) values rather than as references. Additionally, as is common in object oriented programming languages, *references* may be sent in task request postings as well.

The construct for asynchronously sending a task request posting to a target self-stream of an object resembles the Zetalisp (synchronous) send construction:

(sending *self-streams task-selector value lamina-keyword* ...)

Multiple targets for a posting may be specified as a target list and LAMINA keywords (as listed in figure 5) can be used to provide additional control or debugging information. For example, the task request may be sent with a *tag* field that can be used as a descriptive auxiliary value for debugging purposes.

The value immediately returned by sending is the list of *clients* supplied following the LAMINA keyword "for" (or :for-effect if no clients are specified). As a convention, the *clients* may expect to receive consequent task requests later in the computation.

3.2 Creating a New Stream, Ordered Stream, or Sequenced Stream

The streams that pass values between objects are created by the supplied function new-stream. Streams may be tagged for debugging purposes by including a tag as the optional first argument of new-stream as in (new-stream *tag*). The default argument, nil, will cause a stream to inherit a tag identifying the execution in which the call to new-stream appears.

The new-stream function returns a reference for a stream created on the executing site. Often, the reference for a stream (for example, the self-stream of an object) is passed by a procedure as a way of telling some other procedure how the executing (or some other) procedure expects to receive values to use or tasks to accomplish.

A stream may be thought of as an ordered queue of postings. Information can be included in

TO, ON *targets*	A target stream (or site) or list of targets streams (or sites) for the indicated LAMINA operation. If no site is provided and one is needed, an unspecified site is chosen. Some LAMINA operations expect site targets rather than stream targets. These are documented as they are introduced. The choice between the alternative keywords shown is purely stylistic.
FOR *clients*	A stream or list of streams acting as the continuation of the computation that will be triggered by the LAMINA operation.
AS *tag*	Arbitrary data for debugging. Defaults to the tag of the sending execution.
BY *order-key*	A number which may be used to order information in target streams.
AFTER *delay*	Positive number indicating the number of milliseconds that the operation will be delayed before being attempted.
WITH *properties*	Arbitrary data intended for user extensions of the posting protocol.

Figure 5: LAMINA KEYWORD VALUES

postings to allow them to be ordered in streams by specifying a value following the keyword "by" in the call creating the posting. A stream ordered by increasing numeric keys can be created by the function, ordered-stream. The function takes an optional argument for a tag: (ordered-stream *tag*).

As an optimization to simplify programming and to reduce scheduling overhead (by deferring executions involving out of order task invocations), a stream can be created that only presents queued postings that have order keys less than or equal to the next expected order key. This key is greater than or equal to zero and is one more than the highest order key of any previously presented postings. Thus, in the simplest case, the presented postings will have order keys that are in the sequence of the integers beginning with zero. The function, sequenced-stream, that creates such streams also takes an optional argument for a *tag*.

Streams that have at most one value may be created by the function new-future. This function too takes an optional argument for a *tag*.

3.3 Defining Objects

LAMINA object types are built upon the base *flavor* [9], lamina, which defines the instance variable, Self-Stream. The default specification is for a first-in-first-out self-stream. Flavors intended to be mixed in to lamina, the "mixins" ordered-self-stream and sequenced-self-stream, are provided to override this default. As an example similar to the one discussed in section 2, a LAMINA object to associate ordering information with the numerical values of the elements of sets might be defined as shown in figure 6. In the example, the state variables of an ORDER3 *ordering object* are all named, the default initializations specified, and any state variables to be initialized by a creator are identified.

3.4 Triggers

Task request postings specify a task-selector, a value, and the information associated with the keywords in the posting that originated the request. The value and other information in the posting is formatted as a list: (*value clients key tag origin properties*). This list is destructured for execution according to the *trigger-pattern* specified in the trigger definition. Posting elements that are to be ignored need not be specified and an arbitrary degree of destructuring can be specified by the trigger pattern.

```
(DEFFLAVOR ORDER3 ((Controls[6] (ncons :controls))
                   (Smaller-Child) (Larger-Child) Id Result-Stream)
  (lamina)
  (:initable-instance-variables Id Result-Stream)) ; This must be specified

(DEFTRIGGER (ORDER3 :ELEMENT) (input)
  "Set pivot or partition by established pivot.  Check for completed set"
  (destructuring-bind (value set-id) input
    (let* ((control (send self :control set-id))
           (pivot (control-pivot control)))
      (if (null pivot) (setf (control-pivot control) value)
        (if (>= value pivot)
            (sending Larger-Child :element input)
            (sending Smaller-Child :element input)
            (incf (control-smaller control)))) ; Count smaller in set
      (send self :completed? control set-id))))

(DEFTRIGGER (ORDER3 :END) ((base set-id expected))
  "Note base and send :end to children if complete"
  (let ((control (send self :control set-id)))
    (setf (control-expected control) (1+ expected))
    (setf (control-base control) base)
    (send self :completed? control set-id)))

(DEFMETHOD (ORDER3 :CONTROL) (set-id)
  "Get or create control for input and make descendants if none ever made"
  (when (null Smaller-Child)
    (setq Smaller-Child (new-stream) Larger-Child (new-stream))
    (creating 'Order3 '(:Self-Stream ,Smaller-Child :Id (< ,Self-Stream)
                        :Result-Stream ,Result-Stream))
    (creating 'Order3 '(:Self-Stream ,Larger-Child :Id (>= ,Self-Stream)
                        :Result-Stream ,Result-Stream)))
  (or (get Controls set-id) (putprop Controls (make-control) set-id)))

(DEFMETHOD (ORDER3 :COMPLETED?) (control set-id)
  "Count received in set against expected and finish off set if complete"
  (let ((expected (control-expected control)))
    (when (eql expected (incf (control-count control)))
      (let ((pivot (control-pivot control))
            (base (control-base control))
            (smaller (control-smaller control)))
        (let ((pivot-order (+ base smaller))
              (larger (- expected smaller 1)))
          (sending Result-Stream :element '(,pivot ,set-id ,pivot-order))
          (let ((new-base (1+ pivot-order)))
            (if (plusp smaller)
                (sending Smaller-Child :end '(,base ,set-id ,smaller)))
            (if (plusp larger)
                (sending Larger-Child :end '(,new-base ,set-id ,larger))))
          (remprop Controls set-id))))))

(DEFSTRUCT (CONTROL :conc-name :named)
  ((pivot nil) (base nil) (expected nil) (count 0) (smaller 0)))
```

Figure 6: OBJECT ORDERING

---

[5] As a convention, capitalized names are understood to refer to the state variables of an object.

The syntactic form for trigger definition is modeled after the Zetalisp DEFMETHOD form:

```
(DEFTRIGGER (object-type trigger) trigger-pattern
    documentation-string . trigger-body)
```

Example trigger definitions for an ordering object are shown in figure 6. Iteration and assignment replace the recursion and binding used for the functional programming ordering example shown in figure 4. Sequences of values on streams are represented by long lived streams that couple producing and consuming ordering objects.

In the example, each :element message manipulated by the ordering routine indicates the value of the element to be ordered and the set in which that element appears. The output :element messages include this information together with the calculated order of the element in the indicated set. An :end message may be generated either by the root calculation requesting a set be ordered or by intermediate ordering objects serving that calculation. Each such message includes a set identifier, the number of elements the receiver should expect for that set, and the (base) order of the smallest element to be expected. The ORDER3 objects keep track of this (and other) information for each set they are dealing with in a (disembodied property) list of control records. The set of an input is used to retrieve the appropriate control record from among those in use by the object.

If there is no pivot yet received to use in partitioning the set, the ordering object saves the input value as the pivot for the set. Otherwise, the :element trigger method passes the input element to either its larger or smaller child and counts the number of elements sent to the smaller child. If all the expected inputs for a set have been received, an :element message including the value, the set, and the order of the value in the set will be sent to the result stream. An :end message will be sent to any children that have been sent elements of the set to order.

3.5 Creating LAMINA Objects

The form (creating type initializations for client-streams on site ...) stipulates the creation of a object on the indicated site (or on a randomly selected site if none is indicated). When the creation has been accomplished, the client streams will receive a posting whose value is the self-stream of the created object.

The initializations are formed as a list alternating keywords (corresponding to the state variable names for the object being created) with their initial values. These values are computed in the context of the object requesting creation. As an example, creating forms are included in the ORDER3 :control method definition shown in figure 6.

For convenience, a function, create-self-stream, is provided to create a stream which is either an ordered stream, a sequenced stream, or a FIFO stream as appropriate for the self-stream of the lamina object type specified by its argument.

An example of a trigger definition to create three intercommunicating objects is shown in figure 7. In the example, three objects each with state variables referencing the self-stream of each of its siblings are created together. State variables of each object representing an id for the triplet and the object that requested the creation are initialized as well.

3.6 Implicit Continuations

For LAMINA objects, continuations of a computation are often some explicit trigger method of some explicit object. There are cases, however, in which it is inconvenient to create an explicit name for a continuation. As a syntactic construct, execution of a continuation of a computation can be specified to occur in the context of an executing object (as defined by its set of state variables and the environment of the continuation) each time that postings have been received on some given streams. The execution spawning the continuation is finished normally and then the next operation to be done on the object is taken from its self-stream without delay. Thus LAMINA objects can be viewed as monitors [1] (because the independently

```
(DEFTRIGGER (TRIPLICATOR :ABC-TRIPLET) (id client)
 "Expect created object to send notice of its creation"
 (let ((a-stream (create-self-stream 'a))
       (b-stream (create-self-stream 'b))
       (c-stream (create-self-stream 'c)))
   (creating 'a (list :Self-Stream a-stream
                      :B b-stream :C c-stream :Id id :Parent client))
   (creating 'b (list :Self-Stream b-stream
                      :A a-stream :C c-stream :Id id :Parent client))
   (creating 'c (list :Self-Stream c-stream
                      :A a-stream :B b-stream :Id id :Parent client))))
```

Figure 7:   COUPLED OBJECT CREATION

atomic operations on objects give the required mutual exclusion) but operations on them are unnested. This is done to facilitate pipelined operation: task request postings queued for operation on an object are not deferred for a pending continuation.

The construct (with-postings *stream-bindings form*) creates an implicit continuation in the context of an object. The *stream-bindings* is a list each element of which is of the form (*binding-pattern stream*). Each of the postings on the indicated streams (including the posting clients, tag, key, origin, and properties) will be destructured and bound to a corresponding variable (identifier) according to the associated *binding-pattern*. These variables and associated values are also part of the execution environment of the continuation.

```
(DEFTRIGGER (DISTRIBUTER :MAKE-ABC-SERVERS) ((count input-stream))
 "Round robin distribution of input requests to created triplets of servers"
 (let ((a=> (creating 'a nil for (new-stream)
                      on (loop repeat count collect (random-site))))
       (b=> (creating 'b nil for (new-stream)
                      on (loop repeat count collect (random-site))))
       (c=> (creating 'c nil for (new-stream)
                      on (loop repeat count collect (random-site))))
       (servers (ncons nil)))
   (with-postings ((a a=>) (b b=>) (c c=>))
     (if servers (rplacd servers (cons (list a b c) (cdr servers)))
         (setq servers (circular-list (list a b c))))
     (with-postings ((request input-stream))
       (sending (pop servers) :request request as Self-Stream))))))
```

Figure 8:   WITH-POSTINGS

As an example of the use of with-postings, we can consider the example shown in figure 8. It uses nested with-postings constructs to create continuation closures that create and collect triples of lamina nodes and then distribute requests on an input stream to the collected triples in a round robin fashion. Note that instance variables may be accessed by the continuations.

The implicit continuation will be executed atomically with respect to any other operations on the indicated object and in the context of its state variables and the lexical environment in which the form appears. A schematic of the mechanism supporting implict continuations in objects is shown in figure 9.

Figure 9: CONTINUATION CLOSURES

The figure contains the following labels:

task stream [4]

context

context

(...

(... ... (...)

(.... ....

(... ...))))

closure

[1] *request posting*

[3] *response*

[2] *continuation link*
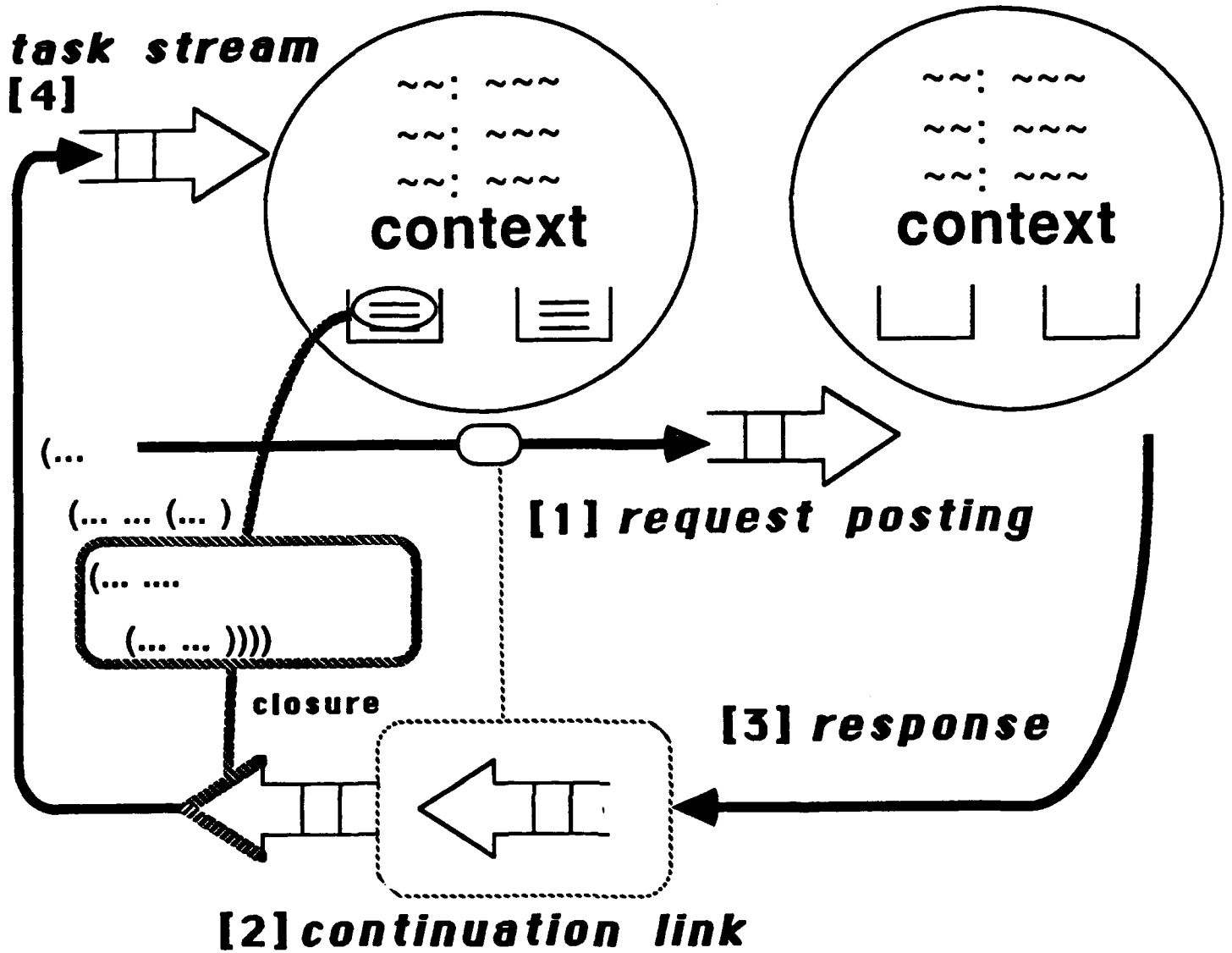
[1] References for streams on which responses are expected are sent in (task request) postings to other objects as places to supply response postings. [2] Intermediate variables (that is, the environment) and a pointer to a block of code required to execute the form wrapped in a with-postings construct are captured in a continuation closure, attached to a stream, and linked to the stream(s) on which responses are expected. [3] When all required postings become available on these streams, [4] the response postings together with the closure are sent to the self-stream of the object that generated the closure.

The closure is executed (in its turn) atomically within the context of the object and lexical environment of the form. Variable bindings are made as specified to the elements of the available response postings. Note that the execution that spawned execution of the closure and the execution so spawned are independently atomic. The state variables of the object and any structures they reference can be changed by some other operation taken from the self-stream between the two executions. The syntactic convenience is only that: invariants that need to be preserved across independent executions need to be met at the boundaries between the execution that spawned execution of the closure and the execution so spawned.

4 Shared Variables

Shared variables are dealt with in LAMINA by treating them as references whose associated value may be mutated. A shared variable reference is constructed, accessed, and mutated by the interface operations described in this section. Support for shared data pairs and arrays is also described. For all these operations, execution is deferred and no other executions are performed by the initiating processor until the indicated operation is accomplished.[6]

Shared queues (which are streams) are also provided. These queues are maintained in a processor's local memory. When a process reads from a shared queue, it is halted and descheduled; execution is resumed when the requested data arrives.

4.1 Creating and Accessing Shared Variables

A shared variable can be allocated on a specific site (containing a processor or memory controller) and given an initial value by (shared-variable *value site-reference*). This creates and returns a reference to the indicated value. The *site-reference* argument is optional; if it is omitted, a randomly selected site is chosen for the default allocation. Alternatively, the construct (in-memory *site-reference forms*) can be used to specify a default site for all allocations done while executing the enclosed *forms*. Thus, the allocation done by the form (in-memory *site-reference* (shared-variable *value*)) is the same as that done by the form (shared-variable *value site-reference*).

Once a shared variable has been allocated, the following constructs may be used to access or alter its value:

- (shared-read *shared-variable-reference*) returns the value of the reference.

- (shared-write *shared-variable-reference value*) modifies the value of the reference. The new value is returned.

- (shared-exchange *shared-variable-reference value*) performs the same function as shared-write, except that the prior value of the reference is returned.

For each of these constructs, the operation is guaranteed to be completed before execution is resumed.

4.2 Shared Data Structures

LAMINA also provides support for pairs or arrays of shared variables. A structure reference is created by an executing process, which may then initialize the structure. The site for the allocation is specified by an optional *site-reference* argument, by the innermost (dynamically) enclosing in-memory form, or is chosen at random.

A shared pair is created by (shared-cons *car-value cdr-value site-reference*). The accessors for a shared pair are shared-car and shared-cdr. Pairs are altered with the forms (shared-rplaca *shared-pair new-car*) and (shared-rplacd *shared-pair new-cdr*). Also, the form (cache-shared-pair *shared-pair-reference*) may be used to make a local, that is, non-shared, copy of a shared pair.

The (shared-array *dimensions site-reference*) form returns a reference to a shared array. The *dimensions* argument is a list of positive integers, denoting the size of each dimension of the array. There are optional :initial-element and :initial-contents keyword arguments, which may be used (respectively) to initialize all the elements of the array to the single value specified or to initialize each element of the array to the value of the

---

[6]Note that, because the simulator is executing in a uniprocessor environment, a stack group must be maintained for each deferred execution. Thus executions must be resumable (not merely restartable) to use the shared variable LAMINA interface described below. This is discussed in section I.10.

```
(DEFUN SHARED-BUFFER (size)
  (let ((<signal> (shared-queue)) (empty? t)
        (<lock> (shared-variable t))
        (<buffer> (shared-array size :initial-element nil))
        (<head> (shared-variable 0))
        (<tail> (shared-variable 0)))
    #'(lambda (operation &optional value)
        (selectq operation
          (:insert
            (with-spin-lock <lock>
              (let* ((head (shared-read <head>))
                     (tail (shared-read <tail>))
                     (new-tail (mod (1+ tail) size)))
                (when (not (= head new-tail))
                  (shared-aset value <buffer> tail)
                  (when empty?
                    (setq empty? nil) (shared-enqueue <signal> <signal>))
                  (shared-write <tail> new-tail)))))
          (:remove
            (with-spin-lock <lock>
              (let ((head (shared-read <head>))
                    (tail (shared-read <tail>)))
                (if (not (= head tail))
                    (let ((new-head (mod (1+ head) size)))
                      (shared-write <head> new-head)
                      (shared-aref <buffer> head))
                    (when (not empty?)
                      (setq empty? t) (shared-dequeue <signal>)))))))))))
```

Figure 10: SHARED BUFFER

corresponding element in a list or a list of lists. Shared arrays are initialized to nil by default.

The form (shared-aref *shared-array-reference subscript* ...) reads elements of the shared array. The number of the subscripts supplied must agree with the dimension of the array. The form (shared-aset *value shared-array-reference subscript* ...) may be used to write array elements. The cache-shared-array function returns a local (non-shared) copy of the shared array reference it is applied to, and the fill-shared-array function copies data from a non-shared array into a shared array.

4.3 Shared Queues

A shared queue construct, which is implemented as a LAMINA stream, is also provided. Because queues are streams, the creator of the queue provides atomic access to the queue and when the queue is empty, maintains a FIFO queue of processes requesting data -- the requests are serviced when data is added to the queue. Further, whenever a process attempts to remove data from the queue, the process is descheduled; execution is rescheduled when the requested data arrives.

Shared queues are created by the shared-queue function, which takes one optional argument representing the queue's tag, which may be used for debugging. Items may be added to the queue with the shared-enqueue function. The shared-dequeue function removes and returns the top item of the queue, while the shared-queue-top function merely returns it.[7] A shared-queue-p function is also provided to test whether an item is a shared queue.

---

[7]In the current implementation, only FIFO queues are provided, and (in order to maintain a consistent timing model for cross address space transmissions) only shared variable or shared queue references may be placed on a shared queue.

```
(DEFUN PART4 (<array> first last)
  "Does partition on array, and returns position of pivot — algorithm from  [7.]"
  (let ((pivot (shared-aref <array> first))
        (i first) (j (1+ last)) (left-item) (right-item))
    (loop for i = (loop for ni from (1+ i) until (= ni j)
                        do (setq left-item (shared-aref <array> ni))
                        when (>= left-item pivot) return ni
                        finally (return ni))
          for j = (loop for nj downfrom (1- j) until (< nj (-1 i))
                        do (setq right-item (shared-aref <array> nj))
                        when (<= right-item pivot) return nj
                        finally (return nj)))
          if (> j i) do (shared-aset left-item <array> j)
                        (shared-aset right-item <array> i)
          else do (shared-aset right-item <array> first)
                  (shared-aset pivot <array> j) and return j))


(DEFUN MAYBE-EXCHANGE (<array> first second)
  "Exchanges first and second items, iff first is greater."
  (let ((first-item (shared-aref <array> first))
        (second-item (shared-aref <array> second)))
    (when (> first-item second-item)
      (shared-aset second-item <array> first)
      (shared-aset first-item <array> second)))))
```

Figure 11:   SHARED VARIABLE PARTITION & EXCHANGE

Unlike other shared variable operations, accesses to shared queues do not cause the initiating processor to stall waiting for completion.   A process executing shared-enqueue continues immediately, without waiting for the data to arrive on the queue.   A process which accesses a queue, using shared-dequeue or shared-queue-top, will be halted and descheduled. Execution is rescheduled when the data arrives, but the initiating processor may perform other executions in the meantime.

4.4 Other Synchronization

A simple spin lock is provided for busy-wait synchronization in the LAMINA shared variable interface.   The form (with-spin-lock shared-variable-reference form) executes the given form after aquiring the lock specified by the indicated shared variable reference.   Subsequently, the lock is released and the value produced by the execution of the form is returned. The lock must be a reference to a shared variable that was initialized to a value other than nil.

We might use such a synchronization operator in incrementing a shared counter as:

```
(DEFUN LOCKED-INCREMENT (<var>[8] <lock> &optional (delta 1))
  (with-spin-lock <lock>
    (let* ((value (shared-read <var>)) (new-value (+ value delta)))
      (shared-write <var> new-value))))
```

We can also create locks based on the shared queue construct.   For example, we implement a mutual exclusion lock as a shared queue.   To release the lock, a process places a token reference on the queue.   A process acquires the lock by removing the token -- any other process which attempts to remove it will be blocked until the owner of the lock replaces the token.   Alternatively, reading but not removing the token (by using shared-queue-top) allows

---

[8]By convention, we denote references to shared variables and shared queues by enclosing angle brackets, as in <lock>.

```
(DEFUN ORDER4 ((<threads> <lock> requests results &optional request)
   (destructuring-bind (<array> first last) request
     (if <array>
         (let* ((pivot-position (part4 <array> first last))
                (contents (list (shared-aref <array> pivot-position)
                                pivot-position <array>)))
           (funcall                     ; Order of pivot data element is established
             results :insert (shared-array 3 :initial-contents contents))
           (let ((left-diff (- pivot-position first))
                 (right-diff (- last pivot-position)))
             (let ((order-left (> left-diff 2))
                   (order-right (> right-diff 2)))
               (cond
                 ((and order-left order-right)      ;Order right partition
                  (let* ((request
                           (list <array> first (1- pivot-position)))
                         (request-block
                           (shared-array 3 :initial-contents request)))
                    (when (null (funcall requests :insert request-block))
                      (order4 <threads> <lock> requests results request))
                    (order4 <threads> <lock> requests results
                            (list <array> (1+ pivot-position) last))))
                 (order-left         ; Exchange right and then order left
                  (when (= right-diff 2)
                    (maybe-exchange <array> (1- last) last))
                  (order4 <threads> <lock> requests results
                          (list <array> first (1- pivot-position))))
                 (order-right        ; Exchange left and then order right
                  (when (= left-diff 2)
                    (maybe-exchange <array> first (1+ first)))
                  (order4 <threads> <lock> requests results
                          (list <array> (1+ pivot-position) last)))
                 (:else              ; Order by exhange for both left and right
                  (when (= right-diff 2)
                    (maybe-exchange <array> (1- last) last))
                  (when (= left-diff 2)
                    (maybe-exchange <array> first (1+ first)))
                  ;;Declare completion of ordering request and try again
                  (locked-increment <threads> <lock> -1)
                  (order4 <threads> <lock> requests results)))))
         (let ((<request> (funcall requests :remove)))
           (if (shared-queue-p <request>)              ;If buffer was empty..
               (if (zerop (shared-read <threads>)) :    signal termination
                   (shared-enqueue <request> <request>)
                   (shared-queue-top <request>)        :    or block till signalled
                   (order4 <threads> <lock> requests results))
               (locked-increment <threads> <lock>) ;Else, pick up request
               (let ((request (listarray (cache-shared-array <request>))))
                 (order4 <threads> <lock> requests results request))))))))
```

Figure 12: SHARED VARIABLE ORDERING

more than one process to be resumed. This last approach more closely resembles the type of synchronization provided by signalling and waiting on condition variables in a monitor.

Figure 10 shows an example of using some of these synchronization schemes in generating a closure to perform operations on a shared buffer realized as a shared variable array. Processes first gain access to the shared array by spinning on a lock. Once access is granted, items are inserted or removed. An attempt to put information in a full buffer returns nil if it is unsuccessful. When an attempt is made to remove data from an empty buffer, a shared queue (rather than data) is returned -- the requesting process may then wait for something to be placed on this queue by executing shared-queue-top.

4.5 An Example

As an example of using the LAMINA shared variable interface, we present yet another implementation of ordering, this one using shared variables. The sets to be ordered are represented as shared arrays.

Each processor will execute an identical *thread of execution*. The execution of the thread is defined by the order4 function, shown in figure 12. Ordering requests are distributed to the threads through a shared buffer manipulated by a closure previously formed by calling the shared-buffer function. A request consists of a reference to a shared array and indices representing the left and right boundaries of the array (or sub-array) to be ordered. Each thread executes in a loop as follows:

- If there is an array (or sub-array) to order, the thread partitions the sub-array, using the part4 routine, shown in figure 11. The order of the set element used as the pivot is now established so the set element, its order, and the reference for the array (as a set identifier) is placed in the specified result queue.

- If both sub-arrays resulting from the partition are longer than two elements, the thread adds an ordering request to the queue for one sub-array and orders the other. If either sub-array has two or fewer elements, the ordering is trivial, so the thread does it (using the maybe-exchange function, also shown in figure 11). If neither sub-array has more than two elements, after the thread orders the sub-arrays, it signals that one less thread is currently working on any ordering requests and notes that it has no array to order.

- If the thread has no array to order, it attempts to remove a request from the queue. If successful, it signals that one more thread is trying to do ordering and orders the (sub-)array identified by the request. If the attempt is unsucessful and there are no other working threads, there will never be any more requests generated so the thread terminates. Otherwise, it tries again to remove a request from the queue. Note that the first thread to terminate places a token on the shared synchronization queue -- this wakes up the other threads, which will then terminate.

5 Utilities: Random Sites, Local Sites, Dismiss, and Boot

A few utility operations are provided by LAMINA to specify computation (and storage) sites, dismiss computations, and provide a timeout facility for applications desiring one. LAMINA also provides simulation control facilities to initiate a CARE simulation, read the current simulation time, and do a computation without increasing the simulation time.

The function random-site returns a reference for a site chosen randomly with uniform distribution over the processor sites in the simulated system. The function random-memory does the same thing over the memory controllers in the system. The function local-site returns a reference for the CARE site executing the function. The function local-memory returns a reference for a memory controller associated with the processor on which the function is executed.

In order to provide a timeout facility, the keyword after followed by a number of milliseconds in simulated time may be included in functions that take LAMINA keyword arguments. The simplest use might be to specify that a posting to a stream be sent at some future time.

A call to dismiss breaks execution. With no argument, execution is rescheduled immediately (but occurs after all previously scheduled executions are run). If an argument is specified which is a keyword, execution is terminated and will never be rescheduled. If a local stream is specified, execution is rescheduled when next that stream receives a posting -- or immediately, if that stream has a posting on it.

The current simulation time (in milliseconds) is returned by the function simulation-time.

Some computations in a simulated application need not (or should not) be timed. The macro (without-clock *form*) enclosing the forms of such computations will cause them to be accomplished "off the clock". This is generally a good idea for calls to debuggers and the like as well as for input-output operations.

Something special must be done to start up a simulation. The form

        (boot (at *time site-coordinates form*) (at ......))

will spawn computations to execute forms at the indicated sites beginning at the specified times (in milliseconds). The site coordinates are given as a list, for example, '(3 2), whose length matches the represented dimensionality of the processing unit (a surface for the case shown). The boot construct resets the simulator and thus may only be executed as the first operation of an application being simulated.

CARE user applications should be loaded into the Zetalisp care-user package where all LAMINA interface constructs and primitive functions are defined.

6 Acknowledgements

The maturation of LAMINA, to the extent this has occured, has only come to pass through the sufferance of its early users. Occasionally, a user has taken a direct hand in LAMINA's definition and implementation. The work so done has invariably improved LAMINA and made it a sounder base for concurrent programming. We are indebted to our colleagues, past and present, Eric Schoen, Harold Brown, Masufumi Minami, Russell Nakano, and Max Hailperin for putting up with our offspring and helping to direct its growth.

This work takes its roots in the achievements of Daniel Friedman, David Wise, Henry Lieberman, and Carl Hewitt. The most important concepts underlying LAMINA are theirs. The distortions of those concepts, done in error or out of a preoccupation with performance (or both) are our own.

I. LAMINA Primitives

A set of functional primitives underlies the interface syntax described in the previous sections of this paper. The set of primitives described below has evolved to provide the mechanisms to support all that syntax. It is documented here so that language implementers may more easily define additional or alternative syntax.

I.1 Posting and Target Specialization

Streams acquire values as a result of postings received by them. This is directly done by the posting operation as in (posting *value* to *target-streams* ...). A posting may be multicast [3] by supplying a list of *target-streams*.

CARE provides a facility for specializing the values transmitted in a multicast to the individual targets of the message. Anyplace a stream is used as a target of a posting, it may be replaced by a cons of that stream and the value specialization for that stream. The value specialization will be used with the value of the posting to form a list whose elements are the list elements of the specialization (or the specification itself if it is not a list) followed by the list elements of the posting value (or the posting value itself if it is not a list). This combined list will be taken as the value of the posting when it arrives at the target stream. The simplest use of this may be to multicast some data to two remote LAMINA nodes as described in section 3, asking them to perform two different operations on the data:

    (posting *data* to '((,*input-stream-1* . ,*task-selector-1*)
                         (,*input-stream-2* . ,*task-selector-2*)) ...)

Specialization is specified by a list of lists even if only one target is involved. This is required to distinguish it from a list of unspecialized targets.

I.2 Stream Posting Access Functions

The form (first-posting *stream*) returns the first posting of those present on a stream. The form (next-posting *stream*) does the same but removes the posting from the stream. The form (last-posting *stream*) returns the last posting and eliminates all others on the stream.

If the stream is empty, the three stream posting access functions, just listed, return nil. Otherwise, they return a posting as a list of the *value, clients, key, tag, origin,* and *properties* of the posting in that order. This list may be used with Lisp destructuring operators. Elements of this list may also be accessed by the posting- macros: -value, -clients, -key, -tag, -origin, and -properties. Each of these takes a posting as an argument. The number of postings available on a stream is returned by the form (postings *stream*).

If it is desired that execution be blocked until there is a posting for a specified stream, the stream posting access forms above may be wrapped in an (accept ...) construction, for example, (accept (next-posting *stream*)). When a posting is available on the indicated stream, the posting is returned to the restarted or resumed execution.

I.3 Copying Streams

A posting sent to parent streams in a tree (or graph) of streams set up by copying operations will result in that posting also appearing on all the descendant streams in the tree (or graph). Such a system of streams can be built by:

    (copying *parents* to *child-streams* for *clients* ...)

The references for the *child-streams* are sent in an operation request posting to the *parents*

where they are added to the child references of the parent. The current queue of postings held in the parent stream is copied and returned in one combined posting that is multicast to the child streams. These postings become part of each child stream. When each child receives the combined postings, it sends on to the *clients* a completion posting whose value is the parent stream from which it received the posting queue. This can be used to validate that a requested copy operation has been accomplished.

I.4 Linking Streams

Linking is an optimization of copying for those cases where it is known that postings need not be retained on intermediate streams in a system of linked streams. Linking parent streams to child streams serves to restrict the parents to act only as intermediaries in a system of linked streams. The syntax for linking is:

(linking *parents* to *child-streams* for *clients* ...)

The references for the *child-streams* are multicast in an operation request posting to the *parents*. When a parent receives the references, any postings already on parent streams are sent to the children specified by the references and eliminated from the parents. Further postings are not retained on parents after they receive a linking directive but are immediately passed on to the child streams. For efficiency in forwarding, the implementation may bypass intermediate levels in a system of linked streams.

I.5 Value Specialization

Target specialization may also be used with the linking or copying operator to specialize the value of postings transmitted from parents to children:

(linking *parents* to '((,*child-1* . ,*value-specialization-1*)) ...)

Thereafter, all postings that traverse that link from parent to child will have the appropriate value specialization prepended to their value. The resulting value is a list whose elements are the list elements of the value specialization (or the value specialization itself if it is not a list) and the list elements of the posting value (or the posting value itself if it is not a list). This is the mechanism used to support the syntax of with-postings when a continuation closure with associated response posting are to be put on a the self-stream of an object.

I.6 Relocating Streams

A linking operation does not change the way that a child stream orders postings or presents them. Relocating a stream from one site to another with that stream's means of ordering and presenting postings (together with any accumulated postings) is specified by:

(relocating *parents* to *child-streams* for *clients* ...)

This is used when there is an attempt to read from a stream that is not local to a site. The attempt causes the reference used to specify that the target stream target a new child stream, the relocation of the previously specified target. No change can be detected in the operation of reference-eq on the reference after relocation.

I.7 Group Streams

An application in LAMINA may wish to view a group of streams as a composite, a *group-stream*, carrying out some operation when all of the streams in the group have received a posting. To minimize unproductive scheduling, computations may wait on such stream composites rather than the individual streams. Group-streams are created by new-stream called with a :group keyword argument as in: (new-stream *tag* :group *member-streams*). A future, that is a stream

which may have at most one value, may be a member of many groups but otherwise a stream may be the member of only one group. If such streams of values are to be made available to several groups, a system of linked or copied streams can be created as discussed previously.

If a member stream is not local to the site of its group stream, a local member stream is created and the remote member stream is relocated there. The postings sent to the local member streams are taken from the member streams whenever a request that has been made to accept a posting from a group stream can be satisfied. Each posting available from a group stream will contain a list of postings received by its component streams as its value.

The order of posting elements in the list representing a group stream posting will correspond to the order indicated in specifying the component streams of the group stream when it was formed by calling the function new-stream as shown above.

Group streams are used to implement with-postings constructs. Continuations are only scheduled when values are available on all the streams included in the specified stream bindings.

I.8 Accessing and Exchanging Stream Values

Posting-by-posting access of the information on streams may be accomplished by requesting that a stream access function be applied to the streams at the site they exist on:

(accessing *access-function* on *target-streams* for *client-streams* ...)

The *access-function* may be any of the stream posting access functions, for example, the function next-posting described previously. A posting will be sent to the client streams when one is available on a target stream. This is the only way provided for expressing competitive access to a common stream.

An interlocked operation on streams is provided:

(exchanging *value* on *target-streams* for *client-streams* ...)

This causes last-posting to be applied to each target stream and the result sent to each client stream. The *value* replaces the last posting on the target stream. This is done atomically with applying last-posting to the stream.

I.9 Spawning a Restartable Computation

A separate, concurrent computation is created by spawning the execution of a closure as shown in the following example:

(spawning #'(lambda () *form*) on *site-reference* for *clients* ...)

The closure is formed and the *clients* returned immediately as the value of the spawning operation. The closure will sent to the indicated site and eventually executed there. The result of that execution will be returned to the specified client streams.

Spawned computations can block waiting for a value to be available on a stream. When the value is available they will be restarted and any intermediate computations done previously will be redone. This approach is taken to avoid creation of stack groups for every spawned computation. Resumable (as opposed to restartable) computations with their own stack groups can be created by LAMINA operations discussed in section I.10.

As an alternative to mounting computations with their own stack groups, the continuations of partially completed computations can be spawned on the same site as their parent. This is done by the with-values functional programming interface constructs described in section 2 and by the with-postings object-oriented programming interface constructs described in section 3.6.

I.10 Mounting Executions with Stack Groups

If an execution is blocked on trying to accept something from an empty stream, it is either restarted (as discussed above) or resumed when that stream receives a posting. In general, resuming a computation from where it left off (without spawning continuations) requires preserving indeterminate amounts of intermediate state with a stack group. Maintaining many independent stack groups is certainly an expensive operation in simulation and may also be so in a target system implementation.

However, for occasions when the full power and expense of stack group switching is warranted, LAMINA provides a construct in the same format as spawning:

(mounting *closure* on *site-references* for *clients...*)

The clients are returned immediately. The closure is sent to the specified site(s) where it will be applied and the computed result sent to the clients. Note that the boot operation discussed in section 5 spawns rather than mounts a computation. If a mounted computation is needed, it must be explicitly mounted by the computation that boot spawns.

One could implement a multiple fork and join construct (like cobegin ... coend) by mounting a number of processes with a common client stream. The creator could then wait for the appropriate number of responses on the client stream (to insure that the other processes had completed) and then continue its execution.

In applications that wish to view executions created with mounting as non-terminating, the execution will typically have an initial section that sends a reference for a newly created (task) stream to mutually agreed upon streams (by an explicit posting). The referenced task stream will then be used to supply the newly mounted execution with additional operations to perform after it completes its starting procedures.

I.11 Loading Sites and Passing Arguments to Remote Closures

An item may be sent to a remote site, a reference for it created there, and the reference sent to specified clients:

(loading *item* on *site-reference* for *client-streams* ...)

The *client-streams* are returned immediately by the form. Remote closures may be created by loading closures:

(loading #'(lambda *arglist* *form*) on *site-reference* for (new-stream) ...)

The new stream immediately returned will eventually get a value representing a reference for the closure on the specified site. A remote closure may be applied to locally evaluated arguments by passing it those arguments:

(passing *arglist* to *closure-reference* for *clients* ...)

The result of the remote application is sent to the specified clients. The loading and passing operations are combined in spawning.

II. LAMINA Primitives and Interfaces

LAMINA primitive and interface functions are listed in this appendix with a reference to the section or sections in which they are described and discussed.

II.1 References

1.3	REFERENCE *item*	*Function*
1.3	REFERENCE-SITE *reference*	*Function*
1.3	REFERENCE-EQ *reference1 reference2*	*Function*

II.2 Functional Programming Interface

2	FUTURE *form*	*Macro*
2	WITH-VALUES *future-bindings* &body *forms*	*Macro*

The *future-bindings* is a list each element of which is itself a list:
(*binding-pattern future-specifier*).

II.3 Object Oriented Programming Interface

3.1	SENDING *self-streams task-selector value* &rest *lamina-keywords*	*Function*
3.2, I.7	NEW-STREAM &optional *tag* &key group *member-streams*	*Function*
3.2	NEW-FUTURE &optional *tag*	*Function*
3.2	ORDERED-STREAM &optional *tag*	*Function*
3.2	SEQUENCED-STREAM &optional *tag*	*Function*
3.3	LAMINA, ORDERED-SELF-STREAM, and SEQUENCED-SELF-STREAM	*Flavors*
3.3	SELF-STREAM of LAMINA	*Instance Variable*
3.4	DEFTRIGGER (*object-type task-selector*) *trigger-pattern* &optional *documentation-string* &body *forms*	*Macro*

The trigger pattern destructures the list (*value clients key tag origin properties*).

3.5	CREATE-SELF-STREAM *object-type* &optional *tag*	*Function*
3.5	CREATING *object-type state-variable-settings* &rest *lamina-keywords*	*Function*

State-variable-settings is a list alternating (state-variable) keywords and values.

3.6	WITH-POSTINGS *stream-bindings* &body *forms*	*Macro*

The *stream-bindings* is a list each element of which is itself a list:
(*binding-pattern stream-specifier*).

II.4 Shared Variable Interface

4.1	SHARED-VARIABLE *site-reference value*	*Function*
4.1	IN-MEMORY *site* &body *forms*	*Macro*
4.1	SHARED-READ *shared-variable-reference*	*Function*
4.1	SHARED-WRITE *value shared-variable-reference*	*Function*
4.1	SHARED-EXCHANGE *value shared-variable-reference*	*Function*
4.2	SHARED-CONS *car-value cdr-value* &optional *site-reference*	*Function*
4.2	SHARED-CAR *shared-pair-reference*	*Function*
4.2	SHARED-CDR *shared-pair-reference*	*Function*
4.2	SHARED-RPLACA *shared-pair-reference new-car*	*Function*
4.2	SHARED-RPLACD *shared-pair-reference new-cdr*	*Function*
4.2	CACHE-SHARED-PAIR *shared-pair-reference*	*Function*
4.2	SHARED-ARRAY *dimensions* &optional *site-reference* &key :initial-element *value* :initial-contents *value-sequences*	*Function*
4.2	SHARED-AREF *shared-array-reference* &rest *subscripts*	*Function*
4.2	SHARED-ASET *value shared-array-reference* &rest *subscripts*	*Function*
4.2	CACHE-SHARED-ARRAY *shared-array-reference*	*Function*
4.2	FILL-SHARED-ARRAY *array shared-array-reference*	*Function*
4.3	SHARED-QUEUE *tag*	*Function*
4.3	SHARED-ENQUEUE *reference shared-queue-reference*	*Function*
4.3	SHARED-DEQUEUE *shared-queue-reference*	*Function*
4.3	SHARED-QUEUE-TOP *shared-queue-reference*	*Function*
4.3	SHARED-QUEUE-P *item*	*Function*
4.4	WITH-SPIN-LOCK *shared-variable-reference* &body *form*	*Macro*

II.5 Utility Operations

5	RANDOM-SITE and RANDOM-MEMORY	*Functions*
5	LOCAL-MEMORY and LOCAL-SITE	*Functions*
5	DISMISS &optional *stream-or-keyword*	*Function*
5	SIMULATION-TIME	*Function*
5	WITHOUT-CLOCK &body *forms*	*Macro*
5	BOOT &rest *at-forms*	*Macro*

An *at-form* is a list of the form: (at *time site-coordinates* &body *forms*)

II.6 Primitives

I.1	POSTING *value* &rest *lamina-keywords*	*Function*
I.2	POSTINGS *stream*	*Function*
I.2	FIRST-POSTING *local-stream*	*Function*
I.2	NEXT-POSTING *local-stream*	*Function*
I.2	LAST-POSTING *local-stream*	*Function*
I.2	POSTING-VALUE *posting*	*Function*
I.2	POSTING-CLIENTS *posting*	*Function*
I.2	POSTING-KEY *posting*	*Function*
I.2	POSTING-TAG *posting*	*Function*
I.2	POSTING-ORIGIN *posting*	*Function*
I.2	POSTING-PROPERTIES *posting*	*Function*
I.2	ACCEPT *stream-access-form*	*Macro*
I.3	COPYING *parent-streams* &rest *lamina-keywords*	*Function*
I.4, I.5	LINKING *parent-streams* &rest *lamina-keywords*	*Function*
I.6	RELOCATING *parent-streams* &rest *lamina-keywords*	*Function*
I.8	ACCESSING *access-function* &rest *lamina-keywords*	*Function*
I.8	EXCHANGING *value* &rest *lamina-keywords*	*Function*
I.9	SPAWNING *function* &rest *lamina-keywords*	*Function*
I.10	MOUNTING *function* &rest *lamina-keywords*	*Function*
I.11	LOADING *item* &rest *lamina-keywords*	*Function*
I.11	PASSING *arglist* &rest *lamina-keywords*	*Function*

References

1. Gregory R. Andrews and Fred B. Schneider. "Concepts and Notations for Concurrent Programming." *Computing Surveys 15*, 1 (March 1983), 3-43.

2. Harold D. Brown, Eric Schoen, and Bruce A. Delagi. An Experiment in Knowledge-Based Signal Understanding Using Parallel Architectures. Tech. Rept. STAN-CS-86-1136 or KSL-86-69, Stanford University, October, 1986.

3. Gregory Byrd, Russell Nakano, and Bruce Delagi. A Dynamic Cut-Through Communication Protocol with Multicast. Tech. Rept. KSL-87-44, Knowledge Systems Laboratory, Stanford University, August, 1987.

4. Bruce A. Delagi, Nakul Saraiya, Sayuri Nishimura, and Greg Byrd. An Instrumented Architectural Simulation System. Tech. Rept. STAN-CS-87-1148 or KSL-86-36, Knowledge Systems Laboratory, Computer Science Department, Stanford University, January, 1987.

5. Daniel P. Friedman and David S. Wise. An Indeterminate Constructor For Applicative Programming. 7th Annual Symposium on Principles of Programming Languages, 1980, pp. 245-250.

6. Robert H. Halstead, Jr. "Multilisp: A Language for Concurrent Symbolic Computation." *ACM Transactions on Programming Languages and Systems 7*, 4 (October 1985), .

7. Donald E. Knuth. *The Art of Computer Programming (Volume 3)*. Addison-Wesley, Reading, Massachusetts, 1973.

8. Henry Lieberman. Thinking About Lots of Things Without Getting Confused. AI Memo 626, MIT, May, 1981.

9. David A. Moon. Object Oriented Programming with Flavors. Object-Oriented Programming Systems, Languages, and Applications [OOPSLA] '86 Proceedings, September, 1986, pp. 1-8.

10. Russell Nakano and Masafumi Minami. Experiments with a Knowledge-Based System on a Multiprocessor. Tech. Rept. KSL-87-61, Knowledge Systems Laboratory, Computer Science Department, Stanford University, 1987.

11. Ehud Shapiro. "Concurrent Prolog: A Progress Report." *Computer 18* (August 1986), 44-58.

12. Steele, G.L. Jr. Lambda, the Ultimate Declarative. AI Memo 379, MIT, November, 1976.

13. Guy L. Steele. *Common Lisp: The Language*. Digital Press, Billerica, MA 01862, 1984.

14. Daniel Weinreb and David Moon. *Lisp Machine Manual*. Symbolics, Cambridge, MA, 1981.

Appendix G

Experiments with a Knowledge-Based System

by

Russell Nakano and Masafumi Minami

Knowledge Systems Laboratory
Computer Science Department
Stanford University
Stanford, California 94305

List of Figures

List of Tables

Abstract

This paper documents the results we obtained and the lessons we learned in the design, implementation, and execution of a simulated real-time application on a simulated parallel processor. Specifically, our parallel program ran 100 times faster on a 100-processor multiprocessor compared to a 1-processor multiprocessor.

The machine architecture is a distributed-memory multiprocessor. The target machine consists of 10 to 1000 processors, but because of simulator limitations, we ran simulations of machines consisting of 1 to 100 processors. Each processor is a computer with its own local memory, executing an independent instruction stream. There is no global shared memory; all processes communicate by message passing. The target programming environment, called Lamina, encourages a programming style that stresses performance gains through problem decomposition, allowing many processors to be brought to bear on a problem. The key is to distribute the processing load over replicated objects, and to increase throughput by building pipelined sequences of objects that handle stages of problem solving.

We focused on a knowledge-based application that simulates real-time understanding of radar tracks, called Airtrac. This paper describes a portion of the Airtrac application implemented in Lamina and a set of experiments that we performed. We confirmed the following hypotheses: 1) Performance of our concurrent program improves with additional processors, and thereby attains a significant level of speedup. 2) Correctness of our concurrent program can be maintained despite a high degree of problem decomposition and highly overloaded input data conditions.

1. Introduction

This paper focuses on the problems confronting the programmer of a concurrent program that runs on a distributed memory multiprocessor. The primary objective of our experiments is to obtain speedup from parallelism without compromising correctness. Specifically, our parallel program ran 100 times faster on a 100-processor multiprocessor compared to a 1-processor multiprocessor. The goal of this paper is to explain why we made certain design choices and how those choices influence our result.

A major theme in our work is the tradeoff between speedup and correctness. We attempt to obtain speedup by decomposing our problem to allow many sub-problems to be solved concurrently. This requires deciding how to partition the data structures and procedures for concurrent execution. We take care in decomposing our problem; to a first approximation, more decomposition allows more concurrency and therefore greater speedup. At the same time, decomposition increases the interactions and dependencies between the sub-problems and makes the task of obtaining a correct solution more difficult.

This paper focuses on the implementation of a knowledge-based expert system in a concurrent object-oriented programming paradigm called Lamina [Delagi 87a]. The target is a distributed-memory machine consisting of 10 to 1000 processors, but because of simulator limitations, our simulations examine 1 to 100 processors. Each processor is a computer with a local memory and an independent instruction stream.[1] There is no global shared memory of any kind.

Airtrac is a knowledge-based application that simulates real-time understanding of radar tracks. This paper describes a portion of the Airtrac application implemented in Lamina and a set of experiments that we performed. We encoded and implemented the knowledge from the domain of real-time radar track interpretation for execution on a distributed-memory message-passing multiprocessor system. Our goal was to achieve a significant level of problem-solving speedup by techniques that exploited both the characteristics of our simulated parallel machine, as well as the parallelism available in our problem domain.

The remainder of this paper is organized as follows. Section 2 introduces definitions that we use throughout the paper. Section 3 describes the model of the parallel machine that we simulate, and the model of computation from the viewpoint of the programmer. Section 4 outlines a set of principles that we follow in our programming effort in order to shed light on why we take the approach that we do. Section 5 describes the signal understanding problem that our parallel program addresses. Section 6 describes the design of our experiments, and Section 7 presents the results. Section 8 discusses a number of design issues, and Section 9 summarizes the paper.

---

[1] Each processor is roughly comparable to a 32-bit microprocessor-based system equipped with a multitasking kernel that supports interprocessor communication and restartable processes (as opposed to resumable processes). The hardware system is assumed to support high-bandwidth, low-latency interprocessor communications as described in Byrd et.al. [Byrd 87].

2. Definitions

Using the definitions of Andrews and Schneider [Andrews 83], a *sequential program* specifies sequential execution of a list of statements; its execution is called a *process*. A *concurrent program* specifies two or more sequential programs that may be executed concurrently as *parallel processes*.

We define $S_{n,m}$ *speedup* as the ratio $\frac{T_m}{T_n}$, where T_k denotes the time for a given task to be completed on a k-processor multiprocessor. Both T_m and T_n represent the *same* concurrent program running on m-processor and n-processor multiprocessors, respectively. When we compare an n-processor multiprocessor to a 1-processor multiprocessor, we obtain a measure for $S_{n/1}$ speedup, which should be distinguished from *true speedup*, defined as the ratio $\frac{T^*}{T_n}$, where T^* denotes the time for a given task to completed by the *best* implementation possible on a uniprocessor.[2] In particular, T^* excludes overhead tasks (e.g. message-passing, synchronization, etc.) that T_1 counts.

We define *correctness* to be the degree to which a concurrent program executing on a k-processor multiprocessor obtains the same solution as a conventional uniprocessor-based sequential program embodying the same knowledge as contained in the concurrent program. We call the latter solution a *reference solution*. We use a serial version of our system to generate a reference solution, to evaluate the correctness of the parallel implementation.[3]

MacLennan [MacLennan 82] distinguishes between value-oriented and object-oriented programming styles. A *value* has the following properties:

• A value is read-only.

• A value is atemporal (i.e. timeless and unchanging).

• A value exhibits referential transparency, that is, there is never the danger of one expression altering something used by another expression.

These properties make values extremely attractive for concurrent programs. Values are immutable and may be read by many processes, either directly or through "copies" of values that are equal; this facilitates the achievement of correctness as well as concurrency. A well-known example of value-oriented programming is functional programming [Henderson 80]. Other examples of value-oriented programming in the realm of parallel computing include systolic programs [Kung 82] and scalar data flow programs [Arvind 83, Dennis 85], where the data flowing from processor to processor may be viewed as values that represent abstractions of various intermediate problem-solving stages.

---

[2] A 1-processor multiprocessor executes the same parallel program that runs on a n-processor multiprocessor. In particular, it creates processes that communicate by sending messages, as opposed to sharing a common memory.

[3] Unfortunately, our reference program is not a valid producer of T^* estimates, and we cannot use it to obtain true speedup estimates. Project resource limitations prevented us from developing an optimized program to serve as a *best* serial implementation.

In contrast, MacLennan defines *objects* in computer programming to have one or more of the following properties:

- An object may be created and destroyed.

- An object has state.

- An object may be changed.

- An object may be shared.

Computer programs often simulate some physical or logical situation, where objects represent the entities in the simulated domain. For example, a record in an employee database corresponds to an employee. An entry in a symbol table corresponds to a variable in the source text of a program. Variables in most high-level programming languages represent objects. Objects provide an abstraction of the state of physical or logical entities, and reflect changes that those entities undergo during the simulation. These properties make objects particularly useful and attractive to a programmer.

Objects in a concurrent program introduce complications. In particular, many parallel processes may attempt to create, destroy, change, or share an object, thereby causing potential problems. For instance, one process may read an object, perform a computation, and change the object. Another process may concurrently perform a similar sequence of actions on the same object, leading to the possibility that operations may interleave, and render the state of the object inconsistent. Many solutions have been proposed, including semaphores, conditional critical regions. and monitors; all of these techniques strive to achieve correctness and involve some loss of concurrency.

Our programming paradigm, Lamina. supports a variation of *monitors*, defined as a collection of permanent variables (we use the term *instance variables*), used to store a resource's state, and some procedures, which implement a set of allowed operations on the resource [Andrews 83]. Although monitors provide mutual exclusion, concurrency considerations force us to abandon mutual exclusion as the sole technique to obtain correctness.

We classify techniques for obtaining speedup in problem-solving into two categories: replication and pipelining. *Replication* is defined as the decomposition of a problem or sub-problem into many independent or partially independent sub-problems that may be concurrently processed. *Pipelining* is defined as the decomposition of a problem or sub-problem into a sequence of operations that may be performed by successive stages of a processing pipeline. The output of one stage is the input to the next stage.

3. Computational model

3.1. Machine model

Our machine architecture, referred to as CARE [Delagi 87a], may be modeled as an asynchronous message-passing distributed system with reliable datagram service [Tanenbaum 81]. After sending a message, a process may continue to execute (i.e. message passing is asynchronous). Arrival order of messages may differ from the order in which they were sent (i.e. datagram as opposed to virtual circuit). The network guarantees that no message is ever lost (i.e. reliable), although it does not guarantee when a message

will arrive. Each processor within the distributed system is a computer that supports interprocessor communication and restartable processes. Each processor operates on its own instruction stream, asynchronously with respect to other processors.

In synchronous message passing, maintaining consistent state between communicating processes is simplified because the sender blocks until the message is received, giving implicit synchronization at the send and receive points. For example, the receiver may correctly make inferences about the sender's program state from the contents of the message it has received, without the possibility that the sender program continued to execute, possibly negating a condition that held at the time the original message was sent.

In asynchronous message passing, the sender continues to execute after sending a message. This has the advantage of introducing more concurrency, which holds the promise of additional speedup. Unfortunately, in its pure form, asynchronous message passing allows the sender to get arbitrarily far ahead of the receiver. This means that the contents of the message reflects the state of the sender at the time the message was sent, which may not necessarily be true at the time the message is received. This consideration makes the maintenance of consistent state across processes difficult, and is discussed more fully in Section 4.

3.2. Programmer model

Our programming paradigm, Lamina, provides language constructs that allows us to exploit the distributed memory machine architecture described earlier [Delagi 87b]. In particular, we focused our programming efforts on the concurrent object-oriented programming model that Lamina provides. As in other object-oriented programming systems, objects encapsulate state information as instance variables. Instance variables may be accessed and manipulated only through methods. Methods are invoked by message-passing.

However, despite the apparent similarity with conventional object-oriented systems, programming within Lamina has fundamental differences:

- Concurrent processes may execute during both object creation and message sending.

- The time required to create an object is visible to the programmer.

- The time required to send a message is visible to the programmer.

- Messages may be received in a different order from which they were sent.

These differences reflect the strong emphasis Lamina places on concurrency. While all object-oriented systems encounter delays in object creation and message sending, these delays are significant within the Lamina paradigm because of the other activities that may proceed concurrently *during* these periods. Subtle and not-so-subtle problems become apparent when concurrent processes communicate, whether to send a message or to create a new object. For instance, a process might detect that a particular condition holds, and respond by sending a message to another process. But because processes continue to execute during message sending, the condition may no longer hold when the message is received. This example illustrates a situation where the recipient of the message cannot correctly assume that because the sender responds to a particular condition by sending a message, that the condition still holds when the message is received.

G-4

Regarding message ordering, partly as a result of our experimentation, versions of Lamina subsequent to the one we used provide the ability for the programmer to specify that messages be handled by the receiver in the same order that they were sent [Delagi 87c]. Use of this feature imposes a performance penalty, which places a responsibility on the programmer to determine that message ordering is truly warranted. In the Airtrac application, we believed that ordering was necessary and imposed it through application level routines that examined message sequence numbers (time tags) and queued messages for which all predecessors had not already been handled.

In Lamina, an object is a process. Following the definition of a process provided earlier, we make no commitment to whether a process has a unique virtual address space associated with it. Each object has a top-level dispatch process that accepts incoming messages and invokes the appropriate message handler; otherwise, if there is no available message, the process blocks. Sending a message to an object corresponds to asynchronous message-passing at the machine level. A method executes atomically. Since each object has a single process, and only that process has access to the internal state (instance variables), mutual exclusion is assured. An object and its methods effectively constitute a non-nested monitor.

Our problem-solving approach has evolved from the blackboard model, where nodes on the blackboard form the basic data objects, and knowledge sources consisting of rules are applied to transform nodes (i.e. objects) and create new nodes [Nii 86a, Nii 86b]. Brown et. al. used concepts from the blackboard model to implement a signal-interpretation application on the CARE multiprocessor simulator [Brown 86]. Lamina evolved from the experiences from that effort. In addition, lessons learned in that earlier effort have been incorporated into our work, including the use of replication and pipelining to gain performance, and improving efficiency and correctness by enforcing a degree of consistency control over many agents computing concurrently.

4. Design principles

Lamina represents a programming philosophy that relies on the concepts of replication and pipelining to achieve speedup on parallel hardware. The key to successful application of these principles relies on finding an appropriate problem decomposition that exploits concurrent execution with minimal dependency between replicated or pipelined processing elements.

The price of concurrency and speedup is the cost of maintaining consistency among objects. When writing a sequential program, a programmer automatically gains mutual exclusion between read/write operations on data structures. This follows directly from the fact that a sequential program has only a single process; a single process has sole control over reads and writes to a variable, for instance. This convenience vanishes when the programmer writes a concurrent program. Since a concurrent program has many concurrently executing processes, coordinating the activities of the processes becomes a significant task.

In this section, we develop the concept of a dependence graph program to provide a framework in which tradeoffs between alternate problem decompositions may be examined. Choosing a decomposition that admits high concurrency gives speedup, but it may do so with the expense of higher effort in maintaining consistency. We introduce dependence graph programs to make the tradeoffs more explicit.

4.1. Speedup

Researchers have debated how much speedup is obtainable on parallel hardware, on both theoretical and empirical grounds; Kruskal has surveyed this area [Kruskal 85]. We take the empirical approach because our goal is to test ideas about parallel problem solving using multiprocessor architectures. Our thinking is guided, however, by a number of principles describing how to decompose problems to obtain speedup.

4.1.1. Pipelining

Consider a concurrent program consisting of three cooperating processes: Reader, Executor, and Printer. The Reader process obtains a line consisting of characters from an input source, sends it to the Executor process, and then repeats this loop. The Executor performs a similar function, receiving a line from the Reader, processing it in some way, and sending it to the Printer. The Printer receives lines from the Executor, and prints out the line. These processes cooperate to form a pipeline; see Figure 1. By using asynchronous message passing, we obtain concurrent operation of the processes; for instance, the Printer may be working on one line, while the Executor is working on another. This means that by assigning each process to a different processor, we can obtain speedup, despite the fact that each line must be inputted, processed, and output sequentially. By overlapping the operations we can achieve a higher throughput than is possible with a single process performing all three tasks.
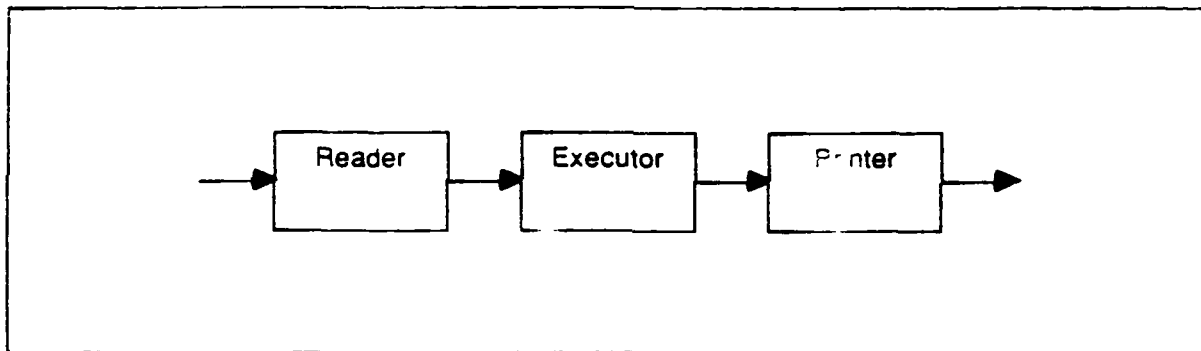
Figure 1.   Decomposing a problem to obtain pipeline speedup.

By decomposing a problem in sequential stages, we can obtain speedup from pipelining.

4.1.2. Replication

Consider a variation of Reader-Executor-Printer problem. Suppose that we are able to achieve some overlap in the operations, but we discover that the Executor stage consistently takes longer than the other stages. This causes the Printer to be continually starved for data, while the Reader completes its task quickly and spends most of its time idle. We can improve the overall throughput by replicating the function of the Executor stage by creating many Executors. See Figure 2. By increasing the number of processes performing a given function, we do not reduce the time it takes a single Executor to perform its function, but we allow many lines to be processed concurrently, improving the utilization of the Reader and Printer processes, and boosting overall throughput. This principle of replicating a stage applies equally well if the Reader or the Printer is the bottleneck.
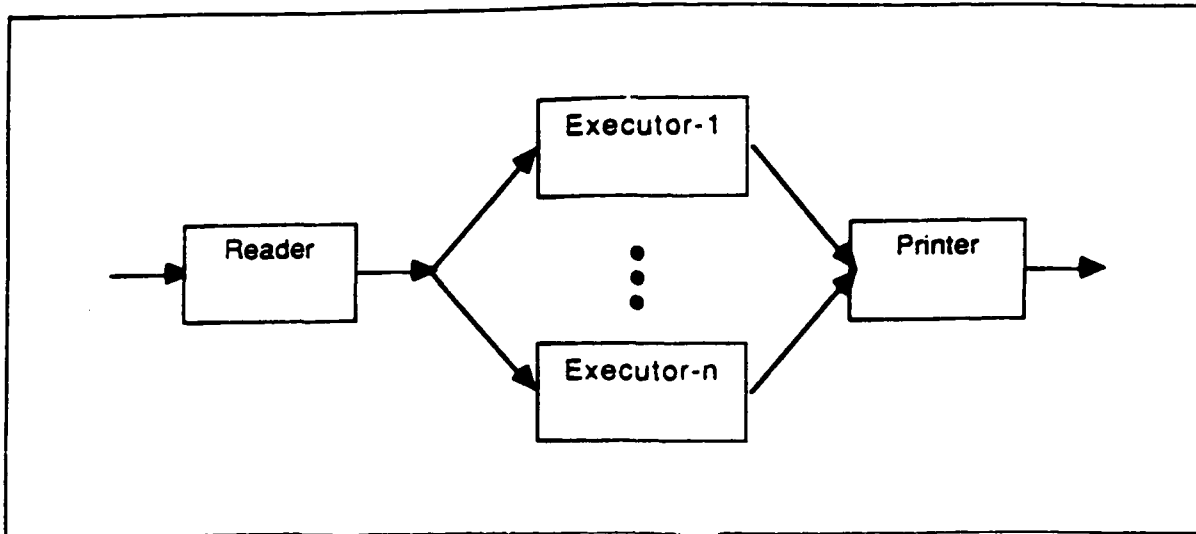
Figure 2. Decomposing a problem to obtain replication speedup.

By duplicating identical problem solving stages, we can obtain speedup from replication.

4.2. Correctness

4.2.1. Consistency

In order to achieve speedup from parallelism, we decompose a problem into smaller sub-problems, where each sub-problem is represented by an object. By doing this, we lose the luxury of mutual exclusion between the sub-problems because of interactions and dependencies that typically exist between sub-parts of a problem. For example, in the Reader-Executor-Printer problem, the simplest version is where a line may be operated upon by one process truly independently; we might want to perform ASCII to EBCDIC character conversion of each line, for instance. We organize the problem solving so that the Reader assembles fixed-length text strings, the Executor performs the conversion, and the Printer does output duties. This problem is well-suited to speedup from the simple pipeline parallelism illustrated in Figure 1. In MacLennan's value/object terminology, a "fixed-length text string" may be viewed as a value that represents the i-th line in the input text; the text string is read-only and it is atemporal. The trick is to   :w the ASCII and EBCDIC versions of a text strings as different *values* corresponding to the i-th line; the Executor's role is to take in ASCII values and transform them into EBCDIC values of the same line. As we will see, value passing has desirable properties in concurrent message-passing systems.

In a more complicated example, we might want to perform text compression by encoding words according to their frequency of appearance, where the Reader process counts the appearance of words and the Executor assigns words to a variable length output symbol set. The frequency table is a source of trouble; it is an object which the Reader writes and updates, and which the Executor reads. Unfortunately, the semantics we impose on the text compression task requires that the Reader complete its scan of the input text before the Executor can begin its encoding task. This dependency prevents us from exploiting pipeline parallelism.

As another example, we might want to compile a high-level language source program text (e.g. Pascal, Lisp, C) into assembly code. Suppose we allow the Reader to build a symbol table for functions and variables, and we let the Executor parse the

tokenized output from the Reader, while the Printer outputs assembly code from the Executor's syntax graph structures. In the scheme outlined here, the symbol table resides with the Reader, so whenever the Executor or Printer needs to access or update the symbol table, it must send a message to the Reader. Consistency becomes an important issue within this setup. For instance, suppose that the Executor determines on the basis of its parse, that the variable x has been declared global. Within a procedure, a local variable also named x is defined, which requires that expressions referring to x within this procedure use a local storage location. Suppose the end of the procedure is encountered, and since we want all subsequent occurrences to x to refer to the global location, the Executor marks the entry for x accordingly (via a message to the Reader). When the Printer sees a reference to x, it consults the symbol table (via a message to the Reader) to determine which storage location should be used; if by misfortune the Printer happens to be handling an expression within the procedure containing the local x, and the symbol table has already been updated, incorrect code will be generated. The essential point is that the symbol table is an object; as we defined earlier, it is shared by several parallel processes, and it changes. A number of fixes are possible, including distinguishing variables by the procedure they are occur within, but this example illustrates that the presence of objects in concurrent program raises a need to deal with consistency.

Consistency is the property that some invariant condition or conditions describing correct behavior of a program holds over all objects in all parallel processes. This is typically difficult to achieve in a concurrent program, since the program itself consists of a sequential list of statements for each individual process or object, while consistency applies to an ensemble of objects. The field of distributed systems focuses on difficulties arising from consistency maintenance [Cornafion 85, Weihl 85, Filman 84]. Smith [Smith 81] refers to this programming goal as the development of a problem-solving protocol.

The work of Schlichting and Schneider [Schlichting 83] is particularly relevant for our situation: they study partial correctness properties of unreliable datagram asynchronous message-passing distributed systems from an axiomatic point of view. They describe a number of sufficient conditions for partial correctness on an asynchronous distributed system:

- monotonic predicates,

- predicate transfer with acknowledgements.

An predicate is *monotonic* if once it becomes true, it remains so. For example, if the Reader process maintains a count of the lines in the variable `totalLines`, and it encounters the last line in the input text, as well having seen all previous lines, then it might send the predicate P, "`totalLines = 16`," to the Executor and to the Printer. The Printer process might use this information even before it has received all the lines, to check if sufficient resources exist to complete the job, for instance. Intuitively, it is valid to assert the total number of lines in the input text because that fact remains unchanged (assuming the input text remains fixed for the duration of the job). Formally, the Reader maintains the following invariant condition on the predicate P:

Invariant: "message not sent" or "P is true"

In contrast, an assertion that the current line is 12, as in "`currentLine = 12`," changes as each line is processed by the Reader. The monotonic criterion cannot be used to guarantee the correctness of this assertion.

A technique to achieve correctness without monotonic predicates is to use acknowledgements. The idea is to require the sender to maintain the truth condition of a predicate or assertion until an acknowledgement from the receiver returns. In the Reader-Executor-Printer example, the Reader follows the convention that once it asserts "currentLine = 12," it will refrain from further actions that would violate this fact until it receives an acknowledgement from the Executor. This protocol allows the Executor to perform internal processing, queries to the Reader, and updates to the Reader, all with the assurance that the current line will remain unchanged until the Executor acknowledges the assertion, thereby signalling that the Reader may proceed to change the assertion. Formally, the Reader and Executor maintain the following invariant condition on the predicate P:

Invariant: "message not sent" or "P is true" or "acknowledgement received"

Note that the each technique has drawbacks, despite their guarantees of correctness. For the monotonic predicate technique, the challenge is to define a problem decomposition and solution protocol for which monotonic predicates are meaningful. In particular, if a problem decomposition truly allows transfer of values between processes, then by the semantics of values as we have defined them, values are automatically monotonic. This explains in formal terms why a "data flow" problem decomposition that passes values avoids difficult problems related to consistency. For the predicate acknowledgement technique, we may address problems that do not cleanly admit monotonic predicates, but we lose concurrency in the assert-acknowledge cycle. Less concurrency tends to translate into less speedup. In the worst case, we may lose so much concurrency in the assert-acknowledge cycle that we find that we have spent our efforts in decomposing the problem into sub-problems only to discover that our concurrent program performs no faster than an equivalent sequential program!

Throughout the design process, we are motivated by a desire to obtain the highest possible performance while maintaining correctness. For tasks in the problem whose durations impact the performance measures, we take the approach of looking first for problem decompositions that allow either value-passing or monotonic predicate protocols. Where neither of these are possible, we implement predicate acknowledgement protocols. In the implementation of Airtrac-Lamina, we did not have to resort to heuristic schemes that did not guarantee correctness.

For initialization tasks, the time to perform initialization tasks (e.g. creating manager objects and distributing lookup tables) is not counted in the performance metrics, but correctness is paramount. Since initialization requires the establishment of a consistent beginning state over many objects, we use the predicate acknowledgement technique to have objects initialize their internal state based on information contained in an initialization message, and then signal their readiness to proceed by responding with an acknowledgement message.

4.2.2. Mutual exclusion

Lamina objects are encapsulations of data, together with methods that manipulate the data. They constitute monitors which provide mutual exclusion over the resources they encapsulate. These monitors are "non-nested" because when a Lamina method (i.e. message handler) in the current CARE implementation invokes another Lamina method, it does so by asynchronous message passing (where the sender continues executing after the message is sent), thereby losing the mutual exclusion required for nested monitor calls. In return, Lamina gains opportunities to increase concurrency by pipelining sequences of operations.

Within the restriction of non-nested monitor calls, the programmer may use Lamina monitors to define atomic operations. If correctness were the sole concern, the programmer could develop the entire problem solution within a single method on a single object; but this is an extreme case. The entire enterprise of designing programs for multiprocessors is motivated by a desire for speedup, and monitors provide a base level of mutual exclusion from which a correct concurrent program may be constructed.

The critical design task is to determine the data structures and methods which deserve the atomicity that monitors provide. The choice is far from obvious. For example, in the ASCII-to-EBCDIC translator example, we assumed the Executor process sequentially scanning through the string, translating one character at a time. We see that the translation of each character may be performed independently, so a finer-grained problem decomposition is to have many Executor processes, each translating a section of the text line. In the extreme, we can arrange for each character to be translated by one of many replicated Executor processes. Choosing the best decomposition is a function of the relative costs of performing the character translation versus the overhead associated with partitioning the line, sending messages, and reassembling the translated text fragments (in the correct order!). The answer depends on specific machine performance parameters and the type of task involved, which in our example is the very simple job of character translation, but might in general be a time-consuming operation. Unfortunately, the programmer often lacks the specific performance figures on which to base such decisions, and must choose a decomposition based on subjective assessments of the complexity of the task at hand, weighed against the perceived run-time overhead of decomposition, together with the run-time worries associated with consistency maintenance. On the issue of how to choose the best "grain-size" for problem solving, we can offer no specific guidance. However, since the CARE-Lamina simulator is heavily instrumented, it lets the programmer observe the relative amount of time spent in actual computation versus overhead activities.

In addition to providing mutual exclusion, Lamina also encourages the structured programming style that results from the use of objects and methods. In particular, mutual exclusion may be exploited without necessarily building large, monolithic objects and methods that might reflect poor software engineering practice. Since Lamina itself is built on Zetalisp's Flavors system [Weinreb 80], it is easy for the programmer to define object "flavors" with instance variables and associated methods to be atomically executed within a Lamina monitor. This can provide important benefits of modularity and structure to the software engineering effort.

To summarize, Lamina objects and methods may be viewed as non-nested monitor constructs that provide the programmer with a base level of mutual exclusion. The potential for additional concurrency and problem-solving speedup increases as finer decompositions of data and methods are adopted. However, these benefits must be weighed against the difficulties of maintaining consistency between objects in a concurrent program. Two techniques for maintaining consistency have been described, differing in their applicability and impact on concurrency.

4.3. Dependence graph programs

The previous sections have defined concepts relevant to the dual goals of achieving speedup and correctness. This section builds upon those concepts to provide a framework in which tradeoffs between speedup and correctness may be examined. A *dependence graph program* is an abstract representation of a solution to a given problem in which values flow between nodes in a directed graph, where each node applies a function to the values arriving on its incoming edges and sends out a value on zero or more outgoing

edges. The edges correspond to the dependencies which exist between the functions [Arvind 83]. A *pure* dependence graph program is one in which the functions on the nodes are free from side effects; in particular, a pure dependence graph program prohibits a function from saving state on any node. (Note that this definition does not preclude a system-level program on a node from handling a function $f(x, y)$ by saving the value of x if the value of x arrives before the value for y. Strictly speaking, an implementation of an f function node must save state, but this state is invisible to the programmer.) A *hybrid* dependence graph program is one in which one or more nodes save state in the form of local instance variables on the node. Functions have access to those instance variables.

Gajski et. al. [Gajski 82] summarize the principles underlying pure data flow computation:

* asynchrony

* functionality.

Asynchrony means that all operations are executed when and only when the required operands are available. *Functionality* means that all operations are functions, that is, there are no side effects.

Pure dependence graph programs have two desirable properties. First, consistency is guaranteed by design. As we have defined it, there are only values and transformations applied to those values. There are no objects to cause inconsistency problems. Second, we can theoretically achieve the maximal amount of parallelism in the solution, and if we ignore overhead costs, maximize speedup in overall performance. This follows from the asynchrony principle, which means that in the ideal case we can arrange for each computation on a node to proceed as soon as all values on the incoming edges are available.

Hybrid dependence graph programs allow side effects to instance variables on nodes, thereby making it more convenient and straightforward to perform certain operations, especially those associated with lookup and matching. This immediately introduces objects into the computational model, and raises the usual concerns about consistency and correctness.

We will use dependence graph programs to serve two purposes. First, we depict the dependencies contained within a problem. Second, we explain why we made certain design decisions in solving the Airtrac problem; in particular, we show why we impose certain consistency requirements on the problem solving protocol. A dependence graph serves as an abstract representation of a problem solution, rather than a blueprint for actual implementation. Specifically, we want to avoid the pitfall of using a dependence graph program to dictate the actual problem decomposition. Overhead delays associated with message routing/sending and process invocation degrade speedup from the theoretical ideal if the actual implementation chooses to decompose the problem down to the grain-size typically found in a dependence graph representation. Given an arithmetic expression, for instance, it may not be desirable to define the grain-size of primitive operations at the level of add, subtract, and multiply. This may lead to the undesirable situation where excessive overhead time is consumed in message packing, tagging, routing, packing, matching, unpacking, and so forth, only to support a simple add operation.

Consider the following numerical example from Gajski et. al. [Gajski 82]. The pseudo-code representation of the problem is as follows:

```
input d,e,f
    c_0 = 0
for i from 1 to 8 do
    begin
        a_i = d_i / e_i
        b_i = a_i * f_i
        c_i = b_i + c_{i-1}
    end
output a,b,c
```

One possible dependence graph program for this problem is shown in Figure 3. This is the same graph presented by Gajski et. al. They assume that division takes three processing units, multiplication takes two units, and addition takes one unit. As noted in their paper, the critical path is the computational sequence a_1, b_1, c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8; the lower bound on the execution time 13 time units.
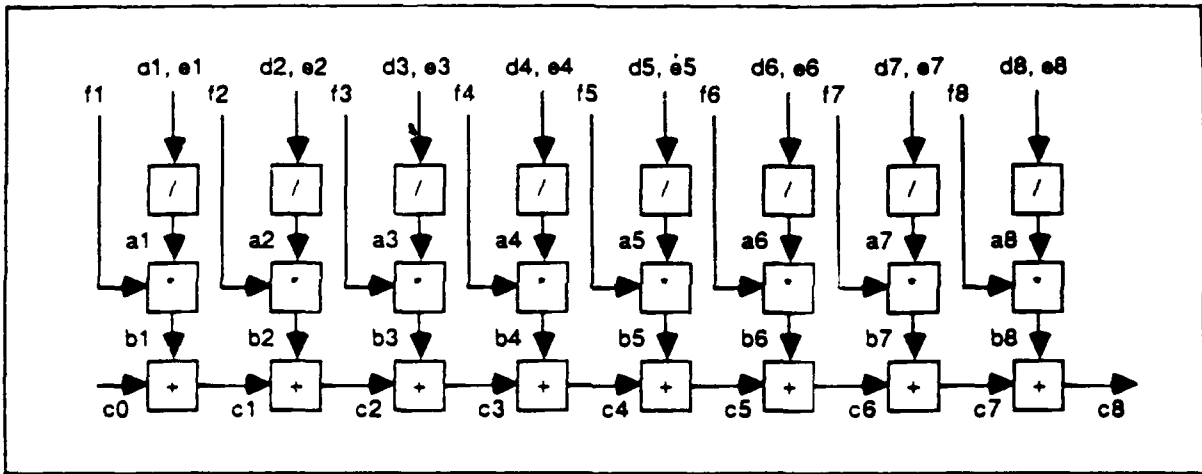
Figure 3. A dependence graph program for a simple numerical computation.

A possible concurrent program implementation would be to assign eight processes to compute the quantities $b_1,....,b_8$, and a ninth to combine the b_i and output $c_1,....,c_8$. Such an arrangement maximizes the decomposition of the problem into sub-problems that may run concurrently, while minimizing the communication overhead. For instance, there is no loss in combining the computation of $c_1,....,c_8$ into a single process because of the inherently serial nature of this particular computation.

Another concurrent program might choose a slightly different decomposition and partition the computation of $c_1,....,c_8$ into, say, three processes: c_1-c_2-c_3, c_4-c_5-c_6, and c_7-c_8. This arrangement uses 11 processes versus the 9 processes in the previous example. While this leads to no improve...ent in the lower bound of 13 time units for a single computation with d, e, and f, it shows an improvement with repeated computations with different values of the input arrays, d, e, and f. For instance, this allows one computation to be summing on the c_7-c_8 process while another is summing on the c_4-c_5-c_6 process. Depending on the complexity of the computation relative to the overhead costs, it might even be worthwhile to define one process for each of the $c_1,....,c_8$, giving 16 processes overall. This illustrates two points. First, a strictly sequential computation gives

an opportunity for pipeline concurrency if many such computations are required. Second, given a dependency graph, many possible problem decompositions are possible.

Gajski et. al. also present a different dependence graph program that is optimized to eliminate the "ripple" summation chain by a more efficient summation network. The dependence graph program for this scheme is shown in Figures 4 and 5. Figure 4 is the "top-level" definition of the program. We use the convention of using a single box, optimized summation, in Figure 4 to represent the subgraph that performs the more efficient summation. Figure 5 shows the expansion of that box as a graph. Showing a dependence graph program in this way is merely a convenience; one should envision the subgraphs in their fully expanded form in the top-level dependence program definition.

The associative property of addition is used to derive the optimized summation function. For instance, the computation of c_8 is rewritten as follows:

$$c_8$$
$$= (((((((c0 + b_1) + b_2) + b_3) + b_4) + b_5) + b_6) + b_7) + b_8)$$
$$= (c0 + ((b_1 + b_2) + (b_3 + b_4))) + ((b_5 + b_6) + (b_7 + b_8))$$

By regrouping the addition operations, this dependence graph program has more parallelism, and reduces the lower bound on execution time from 13 to 9 execution time units. It is important to realize that the second program is truly different from the first; it cannot be obtained from the first by graph transformations or syntactic manipulations that do not rely on the semantics of the functions on the nodes.
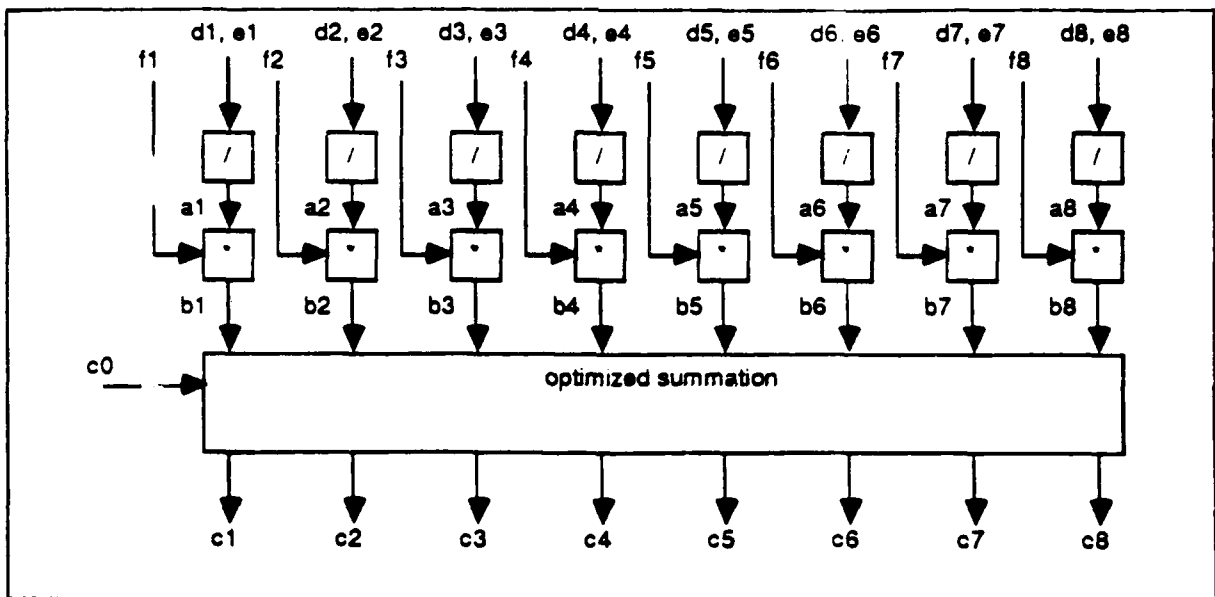
Figure 4. A dependence graph program for the simple numerical computation.

This uses optimization of the recurrence relation using the associative property of addition. This represents the "top-level" definition of the solution. The optimized summation subgraph is shown here a single box, and is shown in expanded form in Figure 5.

optimized summation is defined as...
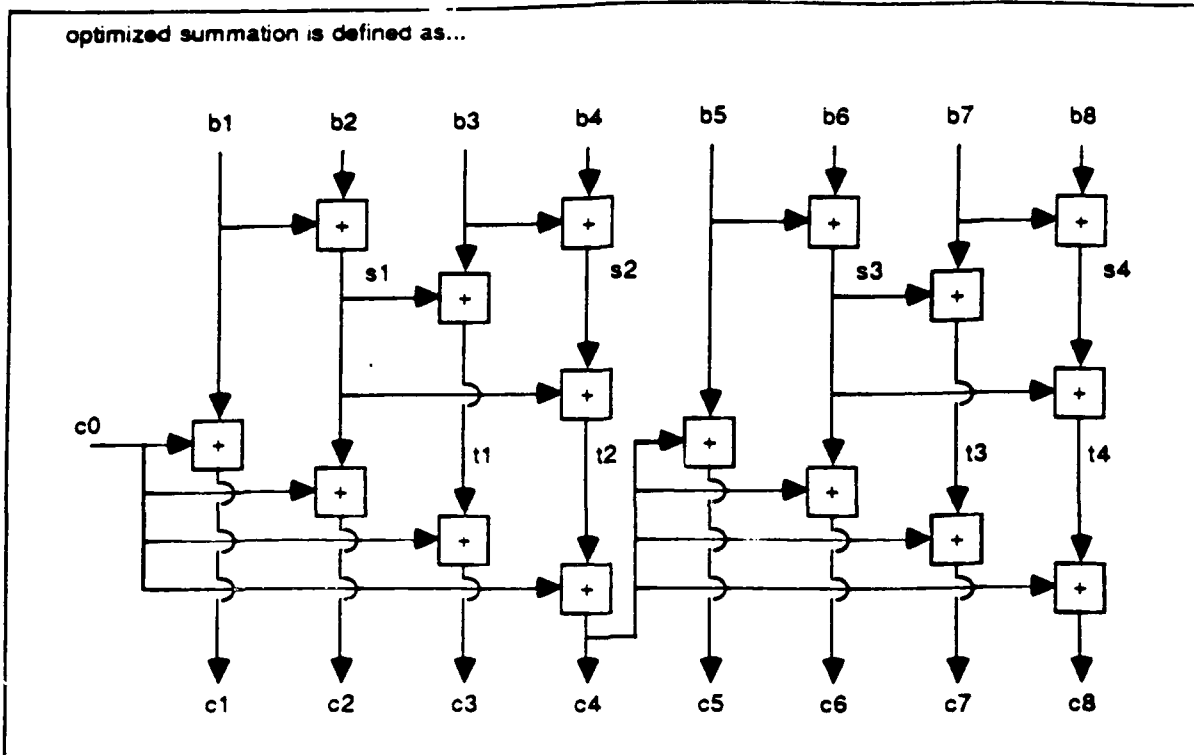
Figure 5. Definition of the "optimized summation" subgraph.

This example highlights several points. First, a given problem may have more than one valid dependence graph program. In the example presented here, the use of knowledge about the underlying semantics of the addition function allows more parallelism. Second, the dependence graph program serves as a intermediate representation from which the solution may be defined for a parallel machine. Third, the dependence graph program does not necessarily make a commitment to the form of the concurrent program. Fourth, for convenience we may describe a dependence graph program as a top-level graph, together with several subgraph definitions.

5. The Airtrac problem

In Airtrac, the problem is to accept radar track data from one or more sensors that are looking for aircraft. Figure 6 depicts a region under surveillance as it might be seen on a display screen at a particular snapshot in time. (Whereas Figure 6 shows many reported sightings, an actual radar would probably show only the most recent sighting.) Locations are designated as either good or bad, where a bad location is illegal or unauthorized, and a good location is legal. The "X" and "Y" symbols represent locations of a good and bad airport, respectively. The locations of radar and acoustic sensors are also shown. The small circles represent track reports that show the location of a moving object in the region of coverage.

Track reports are generated by underlying signal processing and tracking system, and contain the following information:

- location and velocity estimate of object (in x-y plane)

- location and velocity covariance

- the time of the sighting, called the *scantime*

- *track id* for identification purposes.

We would like to answer the following questions in real-time:

- Is an aircraft headed for a bad destination?

- Is it plausible that an aircraft is engaged in smuggling?

By "smuggling" we mean the act of transporting goods from a region or location designated as bad to another bad location. For instance, flying from an illegal airstrip and landing at another illegal airstrip constitutes smuggling.
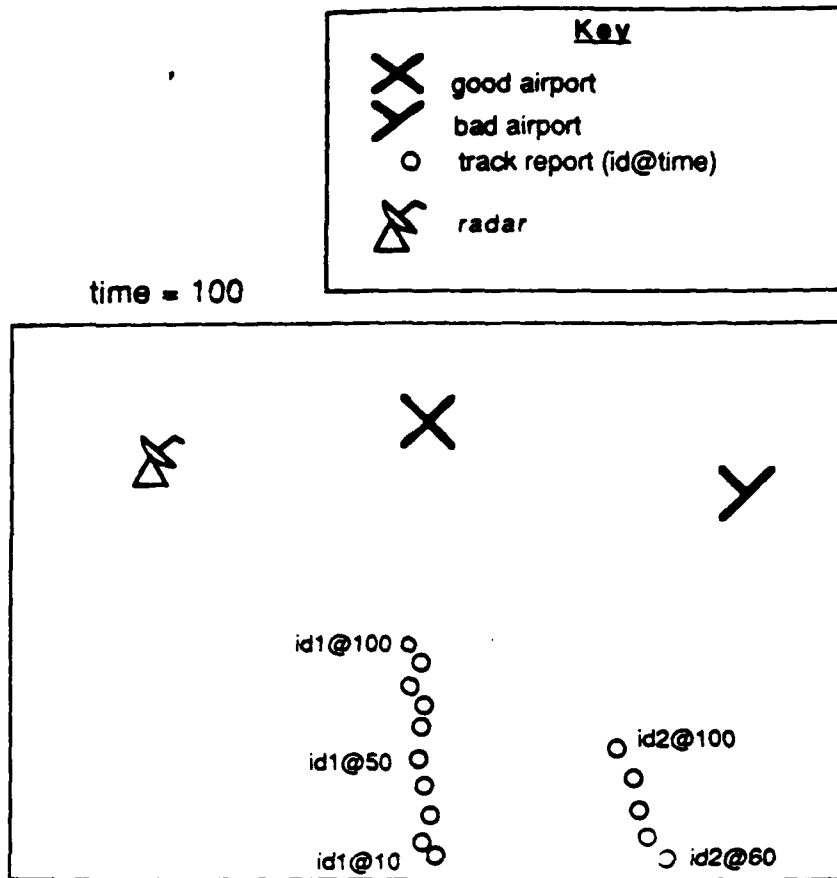
Figure 6. Input to Airtrac.

This shows the inputs that the system receives. The small circles represent estimated positions of objects from radar or acoustic sensors tagged by their identification number and observation time; the goal of the system is to use the time history of those sightings to infer whether an aircraft exists, its possible destinations, and its strategy.

This paper describes our implementation of a solution of a portion of the Airtrac problem. We refer to this portion as the *data association* module. Figure 7 depicts the desired output of the data association step: groupings of reports with the same track id into straight-line, constant-speed sections. These are called *Radar Track Segments*, and have four properties:

- If the Radar Track Segments contains three or more reports, a best-fit line is computed. If the fit is sufficiently good, the segment is declared *confirmed*.

- If a best-fit line has been computed, each subsequent report must fit the line sufficiently closely. If so, the Radar Track Segments remains confirmed. Otherwise, the report that failed to fit (call it the non-fitting report) is treated specially, and the track is declared *broken*.

- A broken track causes the non-fitting report and subsequent reports to be used to form a new Radar Track Segment.

• The last report for a given track id defines that a track is declared *inactive*.

The remaining parts of the Airtrac problem have not yet been implemented as of this writing, but are described more fully elsewhere [Minami 87, Nakano 87].

time = 100
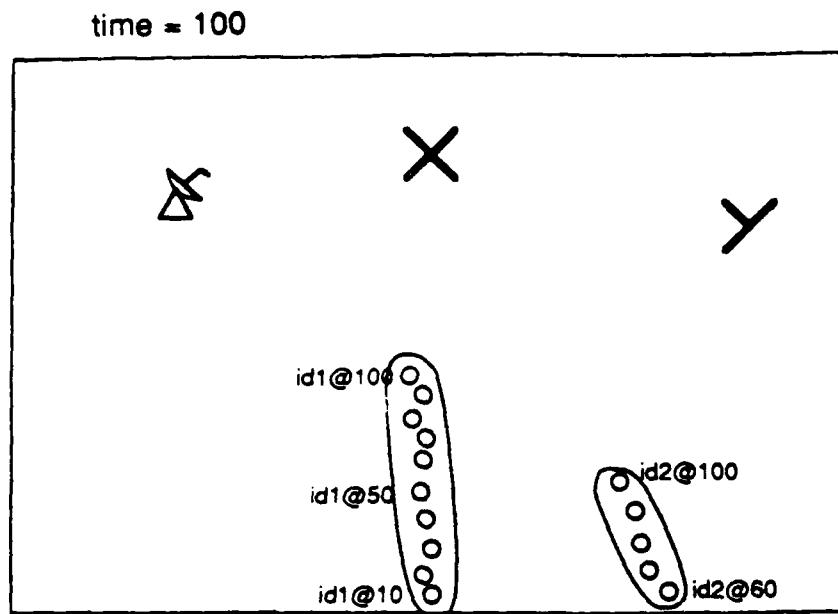
Figure 7. Grouping reports into segments in data association.

This shows the first step in problem solving, grouping the reports into straight-line sections called Radar Track Segments.

5.1. Airtrac data association as dependence graph

Figure 8 shows the Airtrac data association problem as a dependence graph program. On a periodic basis, track reports consisting of position and velocity information for a set of track ids enters the system. Two operations are performed. First, the system checks if a track id is being seen for the first time. If so, a new track-handling subgraph is created. A track-handling subgraph is shown in Figure 8 as a functional box labeled "handle track i," which expands into a graph as shown in Figure 9. Second, the system checks if any track id seen in a previous time has disappeared. If so, it generates an inactivation message for the `handle track` subgraph for the particular track id that disappeared. If the track id has been seen previously, then it is sent to the appropriate `handle track` subgraph.

We distinguish between pure functional nodes, shown as rectangles, and side-effect nodes, shown as rounded rectangles. One use of side-effect nodes is to keep track of which track ids have been seen at the previous time. For instance, by performing set difference operations against the current set of track ids, it is possible to determine the disappeared and new tracks:

```
disappearedTracks = previousTracks - currentTracks
```

G-17

`newTracks = currentTracks - previousTracks`

One way to implement this scheme is to have the `ids disappeared?` and `id previously seen?` nodes update local variables called `previousTracks` and `currentTracks`, as successive track reports arrive.
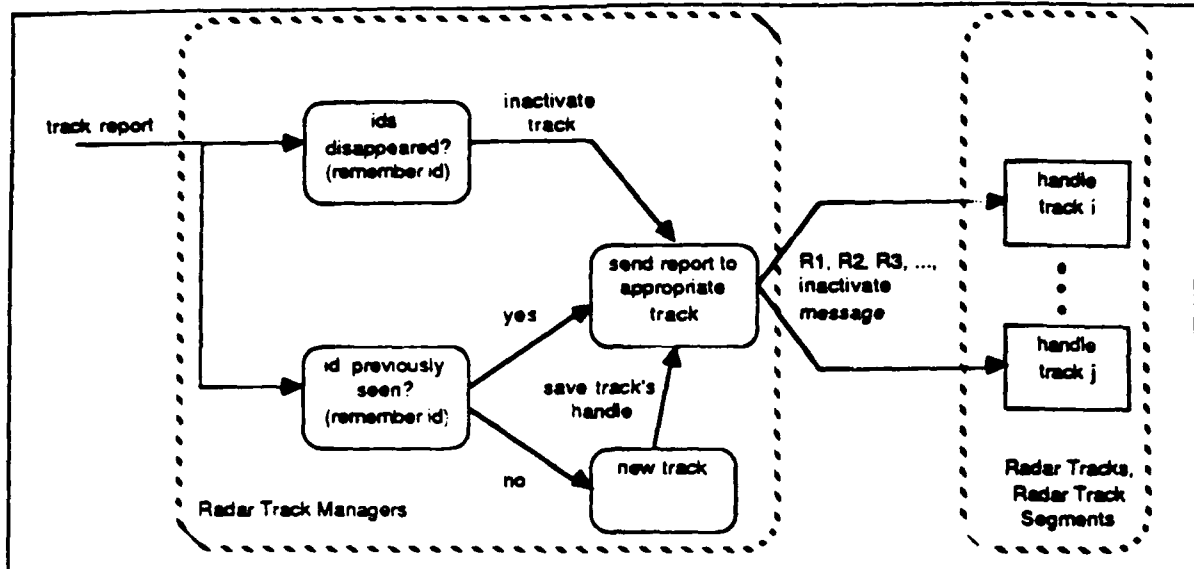
Figure 8. Dependence graph program representation of Airtrac data association.

The dashed boxes indicate the problem decomposition used .n the Lamina implementation.

Besides detecting new and disappeared tracks, side-effect nodes are used to create a new track-handling subgraph, and maintain the lookup table between track id and the message pathway to each track-handling subgraph. `New track` creates a new track handler subgraph. Whenever a new track is encountered, `send report to appropriate track` is notified, so that subsequent reports will be routed correctly. This arrangement requires that one and only one track handler exist for each track id. `Send report to appropriate track` saves the handle[4] to the track handler created by `new track`, sorts the incoming reports, and sends reports to their proper destinations.

In this abstract program, we implicitly assume that only one track report may be processed at a time by the four side-effect nodes in Figure 8. If we allow more than one track report to be processed concurrently, we may encounter inconsistent situations that allow, for instance, a track id to be seen in one track report, but the `send report to appropriate track` node does not yet have the handle to the required track handler subgraph when the next track report arrives. We define the program semantics to avoid these situations.

`Handle track` receives track reports for a particular id, as well as an inactivation message if one exists. It is further decomposed into a subgraph as shown in Figure 9. The

---

[4]A handle is analogous to a mail address in a (physical) postal system: a Lamina object may use another object's handle to send messages to that object. Since the message passing system utilizes dynamic routing and we assume that an object remains stationary once created, the handle does not need to encode any information about the particular path messages should follow.

nodes in the handle track subgraph pass a structured value between them, called track segments. A *track segment* has the following internal structure:

- report list (a list of track reports, initially empty)

- best-fit line (a vector of real numbers describing a straight-line constant-velocity path in the x-y plane)

Each node may transform the incoming value and send a different value on an outgoing edge. Add appends a report to the report list of a track segment. Linefit computes the best-fit line, and if the confirmation conditions hold, sends the track segment to confirm. Confirm declares the track segment as confirmed, and passes the list to check fit. If linefit fails to confirm, the earliest report in the list is dropped by drop, and another add, linefit box awaits the arrival of the next report to restart the cycle. The inactivate function waits until all reports have arrived before declaring the track inactive. Conceptually, we view the operations of confirm and inactivate as being monotonic assertions made to the "outside world," rather than value transformations to the track segment.
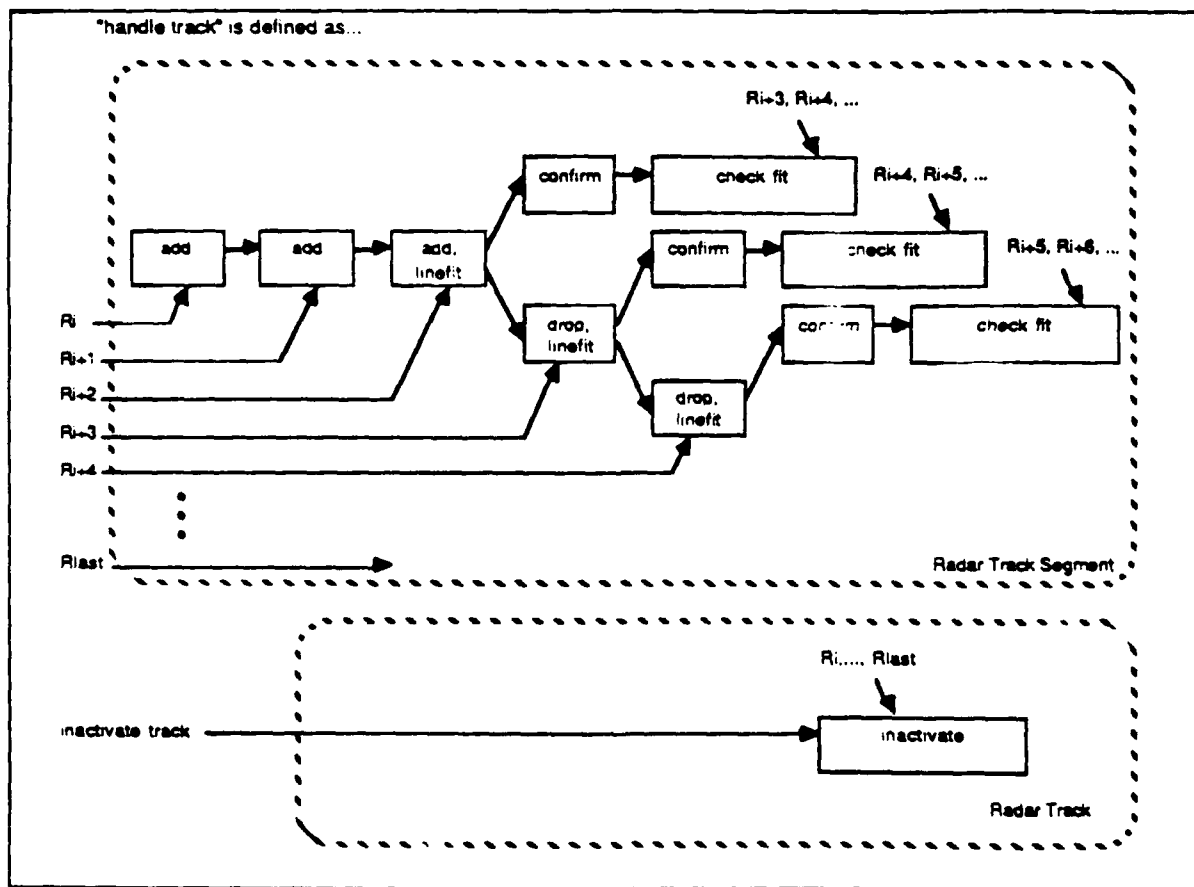
Figure 9. Decomposition of the "handle track" sub-problem.

The dashed boxes indicate the problem decomposition used in the Lamina implementation.

Check fit itself is further decomposed into more primitive operations, as shown in Figure 10. The linecheck operation is similar to the linefit function previously

described, except that it compares a new report against the best-fit line computed during the linefit operation: if the new report maintains the fit, the report list is sent to the OK box, and this cycle is repeated for the next report. If the linecheck operation fails, then the track is declared broken, a new track segment is defined. This track segment is sent the report that failed the linecheck operation, in addition to all subsequent reports for this particular track id. The track handling cycle is repeated as before.
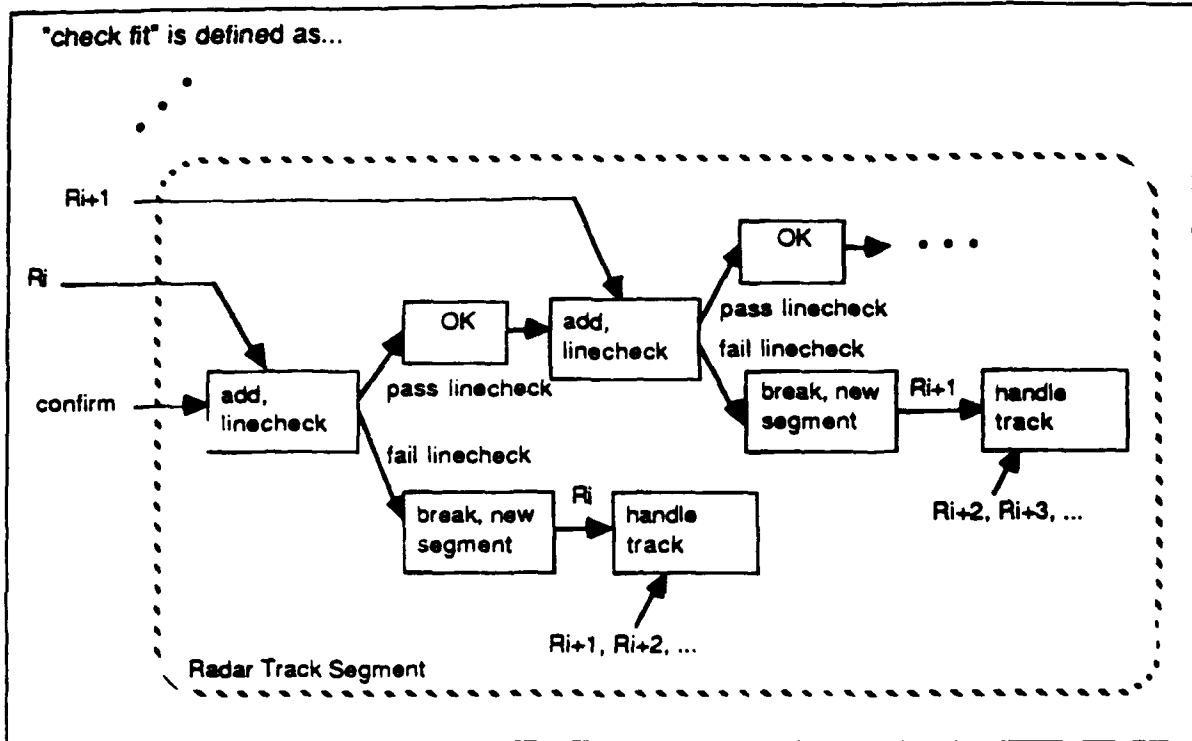
Figure 10.   Decomposition of the "check fit" sub-problem.

The dashed boxes indicate the problem decomposition used in the Lamina implementation.

A number of observations may be made about the dependence graph program described in this section. First, the sequence of the reports matters. The graph structure clearly depicts the requirement that the incorporation of the Ri-th report into the track segment by the add operation must wait until all prior reports, R1,..., Ri-1, have been processed. This is true for the linefit, linecheck, and inactivate functions. Second, this program avoids the saving of state information except in the operations that must determine whether a given track id has been previously seen, and in the sorting operation where track reports are routed to the appropriate track handler. Except for these, we find that the problem may be cast in terms of a sequence of value transformations. Third, the program admits the opportunity for a high degree of parallelism. Once the track handler for a given track id has been determined, the processing within that block is completely independent of all other tracks. Fourth, the opportunity for concurrency within the handling of a particular track is quite low, despite the outward appearance of the decompositions shown in Figures 8 and 9. Indeed, an analysis of the dependencies shows that reports must be processed in order of increasing scantime. Fifth, unlike certain portions of the dependence graph that have a structure that is known *a priori*, the track

handler portions of the graph have no prior knowledge of the track ids that will be encountered during processing, implying that new tracks need to be handled dynamically.

5.2. Lamina implementation

In this section, we express the solution to the data association problem as a set of Lamina objects, together with a set of methods on those objects which embody the abstract solution specification presented in the previous section.

Figure 11 shows how we decompose the Airtrac problem for solution by a Lamina concurrent program. We define six classes of objects: Main Manager, Input Simulator, Input Handler, Radar Track Manager, Radar Track, and Radar Track Segment. Some objects, referred to as *static objects*, are created at initialization time, and include the following object classes: Main Manager, Input Simulator, Input Handler, and Radar Track Manager objects. Others are referred to as *dynamic objects*, are created at run-time in response to the particular input data set, and include the following object classes: Radar Track and Radar Track Segment.
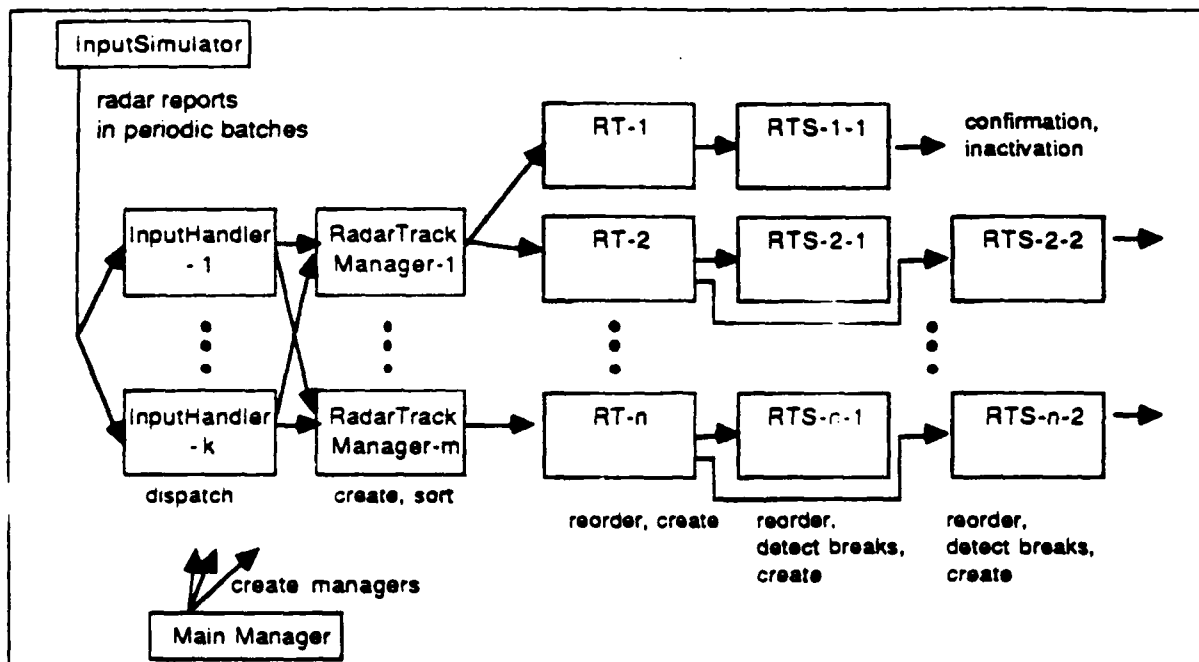
Figure 11.   Object structure in the data association module.

Each object is implemented as a Lamina object, which in Figure 11 corresponds to a separate box. The problem decomposition seeks to achieve concurrent processing of independent sub-problems. The Lamina message-sending system provides the sole means of message and value passing between objects. Wherever possible, we pass values between objects to minimize consistency problems, and to minimize the need for protocols that require acknowledgements. For example, we decompose our problem solving so that we require acknowledgements only during initialization where the Main Manager sets up the communication pathways between static objects.

With respect to the dependence graph program, the Lamina implementation takes a straightforward approach. All of the side-effect functions contained in Figure 8, together with some operations to support replication, reside in the Input Handler and Radar Track

Manager object classes. Objects in these two classes are static; we create a predetermined number of them at initialization time to handle the peak load of reports through the system. Replication is supported by partitioning the task of recognizing new and disappeared track ids among Radar Track Managers according to a simple modulo calculation on the track id. Given the partitioning scheme, each Radar Track Manager operates completely independently from the others. Thus, although it needs to maintain a set of objects (e.g. the current tracks, previous tracks), the objects are encapsulated in a Lamina object. Access to and updating of these objects is atomic, providing the mutual exclusion required to assure correctness as specified by the dependence graph program.

Functions in Figures 9 and 10 reside mostly in objects of the Radar Track Segment class, with the inactivation function being performed by objects of the Radar Track class. Objects of these two classes are dynamic: we create objects at run-time in response to the specific track ids that are encountered. For any particular track id, one Radar Track object together with one or more Radar Track Segment objects are created. A new Radar Track Segment is created each time the track is declared broken, which may occur more than once for each track id. Unlike the dependence graph program where we postulate a track segment as a value successively transformed as it passes through the graph, the Lamina implementation defines a Radar Track Segment object with instance variables to represent the evolving state of the track segment. We implement all the major functions on track segments as Lamina methods on Radar Track Segment objects. Again, Lamina objects provide mutual exclusion to assure correctness.

Although nothing in the problem formulation described here indicates why we create multiple Radar Track Segments for a given track, we do so in anticipation of adding functionality in future versions of Airtrac-Lamina. From examination of Figure 10, we see that given any sequence of reports Ri, and any pattern of broken tracks, we obtain no additional concurrency by creating a new Radar Track Segment when a track is declared broken. This is because in the dependency graph program presented here, no activity occurs on one Radar Track Segment after it has created another Radar Track Segment. However, we anticipate that in subsequent versions of Airtrac-Lamina, a Radar Track Segment will continue to perform actions even after a track is declared broken, such as to respond to queries about itself, or to participate in operations that search over existing Radar Track Segments.

Logically, the semantics of the dependency graph program and the Lamina program are equivalent, as they must be. The difference is that the former requires a graph of indefinite size, where its size corresponds to the number of reports comprising the track. The latter requires a quantity of Radar Track Segment objects equal to one plus the number of times the track is declared broken. Although we can easily conceptualize a graph of indefinite size in a dependency graph program, we cannot create such an entity in practice. Because object creation in Lamina takes time, we try to minimize the number of objects that are created dynamically, especially since their creation time impacts the critical path time. A poor solution is to dynamically create the objects corresponding to an indefinite-sized graph as we need them. A better solution is to create a finite network of objects at initialization time, with an implicit "folding" of the infinite graph onto the finite network, thereby avoiding any object-creation cost at run-time. Our Lamina program, in fact, uses a hybrid of these two approaches, folding an indefinite "handle track" graph onto each Radar Track Segment object, and creating a new Radar Track Segment object dynamically when a a track is declared broken. By this mechanism, we model transformations of values between graph nodes by changes to instance variables on a Lamina object. The effect on performance is beneficial. Relative to the first solution, we incur less overhead in message sending between objects because we have fewer objects. Relative to the second solution, we create objects that correspond to track ids that appear in the input data stream as they are

needed, which has the effect of bringing more processors to bear on the problem as more tracks become visible.

Both the Radar Track and Radar Track Segment collect reports in increasing scantime sequence. They do so because of the ordering dictated by the dependence graph program, and because the Lamina implementation at the time the experiments were performed did not provide automatic message ordering. Moreover, we know that simply collecting reports in order of receipt leads to severe correctness degradation. For instance, if the scantimes are not contiguous, the scheme by which a Radar Track Segment computes the line-fit leads to nonsensical results because best-fit lines will be computed based on non-consecutive position estimates, leading to erroneous predictions of aircraft movement. To circumvent these problems, we use application-level routines to examine the scantime associated with a report, and queue reports for which all predecessors have not already been handled. These routines effectively insulate the rest of the application from message receipt disorder, and allow the Lamina program to successfully use the knowledge embodied in the dependency graph program.

To indicate the size of the problem, a typical scenario that we experimented with contained approximately 800 radar track reports comprising about 70 radar tracks. At its peak, there is data for approximately 30 radar tracks arriving simultaneously, which roughly corresponds to 30 aircraft flying in the area of coverage.

The correspondence between the Lamina objects in the implementation presented here and the primitive operations embodied in the dependence graph program is shown in the Table 1. The functions described in the dependence graphs are implemented on Radar Track Manager, Radar Track, and Radar Track Segment objects. The Main Manager and Input Simulator perform tasks not mentioned in the dependence graph program. Their tasks may be viewed as overhead: the Main Manager performs initialization, and Input Simulator simulates the input data port. The Input Handler's job is to dispatch incoming reports to the correct Radar Track Manager, thereby supporting the replication of the Radar Track Manager function across several objects. In this way the task of the Input Handler may be viewed as a functional extension of the Radar Track Manager tasks.

Table 1. Correspondence of Lamina objects with functions in the dependence graph program

Lamina object	Corresponding dependence graph program operation
Main Manager	-none- (Create the manager objects in the system at initialization time.)
Input Simulator	-none- (Simulate the input data port that would exist in a real system. This function is an artifact of the simulation.)
Input Handler	-none- (Allows replication of the Radar Track Manager objects; this may be viewed as a functional extension of the Radar Track Manager.)
Radar Track Manager	ids disappeared?, id previously seen?, new track, send report to appropriate track
Radar Track	add, inactivate
Radar Track Segment	add, linefit, confirr. drop, inactivate, linecheck, OK, break, new segment

Table 1 also shows that we decompose the problem to a lesser extent than might be suggested by the dependence graph program, but the overall level of decomposition is still high. We "fold" the dependence graph onto a smaller number of Lamina objects, but we nonetheless obtain a high degree of concurrency from the independent handling of separate tracks. Additional concurrency comes from the pipelining of operations between the following sequence of objects: Input Handler, Radar Track Manager, Radar Track, and Radar Track Segment.

6. Experiment design

Given our experimental test setup, there are a large number of parameter settings, including the number of processors, the choice of the input scenario to use, the rate at which the input data is fed into the system, the number of manager objects to utilize; for a reasonable choice of variations, trying to run all combinations is futile. Instead, based on the hypotheses we attempted to confirm or disconfirm, we made explicit decisions about which experiments to try. We chose to explore the following hypotheses:

- Performance of our concurrent program improves with additional processors, thereby attaining significant levels of speedup.

- Correctness of our concurrent program can be maintained despite a high degree of problem decomposition and highly overloaded input data conditions.

- The amount of speedup we can achieve from additional processors is a function of the amount of parallelism inherent in the input data set.

Long wall-clock times associated with each experiment and limited resources forced us to be very selective about which experiments to run. We were physically unable to explore the full combinatorial parameter space. Instead, we varied a single experimental parameter at a time, holding the remaining parameters fixed at a base setting. This strategy relied on an intelligent choice of the base settings of the experimental parameters.

We divided our data gathering effort into two phases. First, we took measurements to choose the base set of parameters. Our objective was to run our concurrent program on a system with a large number of processors (e.g. 64), picking an input scenario that feeds data sufficiently quickly into the system to obtain full but not overloaded processing pipelines. We used a realistic scenario that has parallelism in the number of simultaneous aircraft so that nearly all the processors may be utilized. Finally, we chose the numbers of manager objects so the managers themselves do not limit the processing flow. The goal was to prevent the masking of phenomena necessary to confirm or disconfirm our hypotheses. For example, if we failed to set the input data rate high enough, we would not fully utilize the processors, making it impossible that additional processors display speedup. Similarly, if we failed to use enough manager objects, the overall program performance would be strictly limited by the overtaxed manager objects, again negating the effect of additional processors.

Based on measurements in phase one, we chose the following settings for the base set of parameter settings:

- 64 processors,

- Many-aircraft scenario (described more fully below),

- Four input handler objects,

- Four radar track manager objects,

- Input data rate of 200 scans per second.

These settings give system performance that suggests that processing pipelines are full, but not overloaded, where nearly all of the processing resources are utilized (although not at 100 percent efficiency), and the manager objects are not themselves limiting overall performance.

The input data rate governs how quickly track reports are put into the system. As reference, the Airtrac problem domain prescribes an input data rate of 0.1 scan per second (one scan every 10 seconds), where a scan represents a collection of track reports periodically generated by the tracking hardware. For the purpose of imposing a desired processing load on our simulated multiprocessor, our simulator allows us to vary the input data rate. With a data rate of 200 scans per second, we feed data into our simulated multiprocessor 2000 times faster than prescribed by the domain to obtain a processing load where parallelism shows benefits. Equivalently, we can imagine reducing the performance of each processor and message passing hardware in the multiprocessor by a factor of 2000

to achieve the same effect, or with any combination of input data rate increase and hardware speed reduction that results in a net factor of 2000.

In the second phase, we vary a single parameter while holding the other parameters fixed. We perform the following set of three experiments:

- Vary the number of processors from 1 to 100.

- Vary the input scenario to use the one-aircraft scenario.

- Vary the number of manager objects.

Figure 12 shows how the many-aircraft and one-aircraft scenarios differ in the number of simultaneous active tracks. In the many-aircraft scenario, many aircraft are active simultaneously, giving good opportunity to utilize parallel computing resources. In contrast, the one-aircraft scenario reflects the extreme case where only a single aircraft flies through the coverage area at any instant, although the total number of radar track reports is similar between the two scenarios. Although broken tracks in the one-aircraft scenario may give rise to multiple track ids for the single aircraft, the resulting radar tracks are non-overlapping in time.

Active Tracks vs. Scan
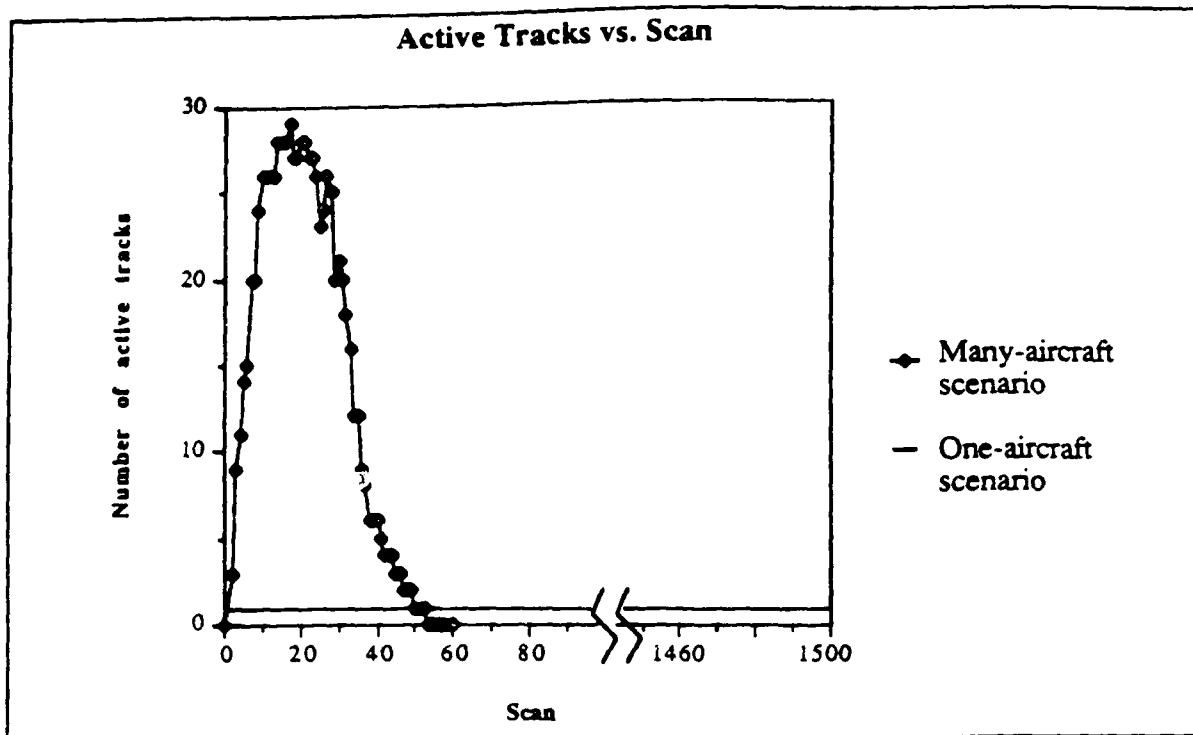
Figure 12.   Comparison of the number of active tracks in the many-aircraft and one-aircraft scenarios.

This shows the number of active tracks versus the scan. The scan number corresponds to scenario time in increments of 0.1 seconds.

7.   Results

7.1.   Speedup

Our performance measure is latency. *Latency* is defined as the duration of time from the point at which the system receives a datum which allows it to make a particular conclusion, to the point at which the concurrent program makes the conclusion. We use latency as our performance measure instead of total running time measures, such as "total time to process all track reports," because we believe that the latter would give undue weight to the reports near the end of the input sequence, rather than weigh performance on all track reports equally.

We focus on two types of latencies: confirmation latency and inactivation latency. *Confirmation latency* measures the duration from the time that the third consecutive report is received for a given track id, to the time that the system has fitted a line through the points, determined that the fit is valid, and it asserts the confirmation. *Inactivation latency* measures the duration from the time that the system receives a track report for the time following the last report for a given track id, to the time when the system detects that the track is no longer active, and asserts the inactivation. Since a given input scenario contains many track reports with many distinct track ids, our results report the mean together with plus and minus one standard deviation.

Figures 13 and 14 show the effects on confirmation and inactivation latencies, respectively, from varying the number of processors from 1 to 100. Boxes in the graphs

indicate the mean. Error bars indicate one standard deviation. The dashed line indicates the locus of linear speedup relative to the single processor case; its locus is equivalent to an $S_{n/1}$ speedup level of n for n processors.

Confirmation Latency vs Number of Processors
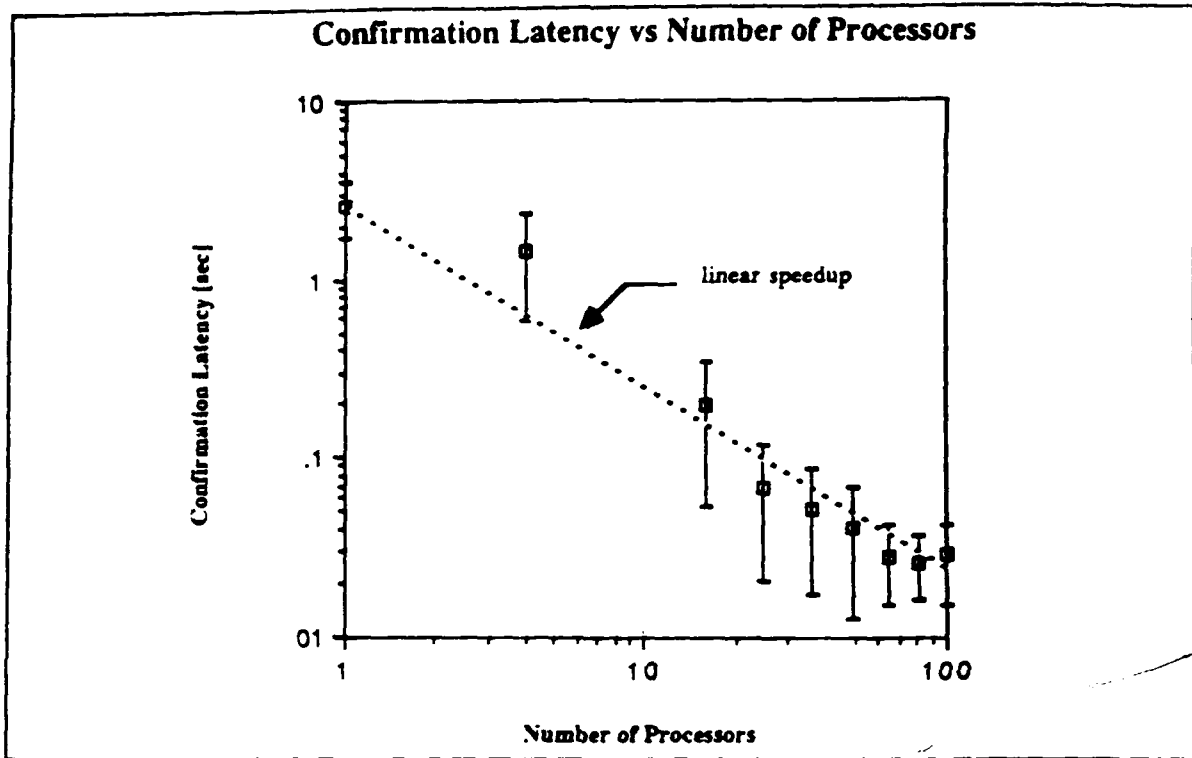
Figure 13. Confirmation latency as a function of the number of processors.

This measures the duration from the time that the third consecutive report is received for a given track id. to the time that the system has fitted a line through the points, and determined that the fit is valid.

The results for both the confirmation and inactivation show a nearly linear decrease in the mean latencies, corresponding to $S_{100/1}$ speedup by a factor of 90 for the confirmation latency, and to $S_{100/1}$ speedup by a factor of 200 for the inactivation latency. The sizes of the error bars make it difficult to pinpoint a leveling off in speedup, if there is any, over the 1 to 100 processor range.
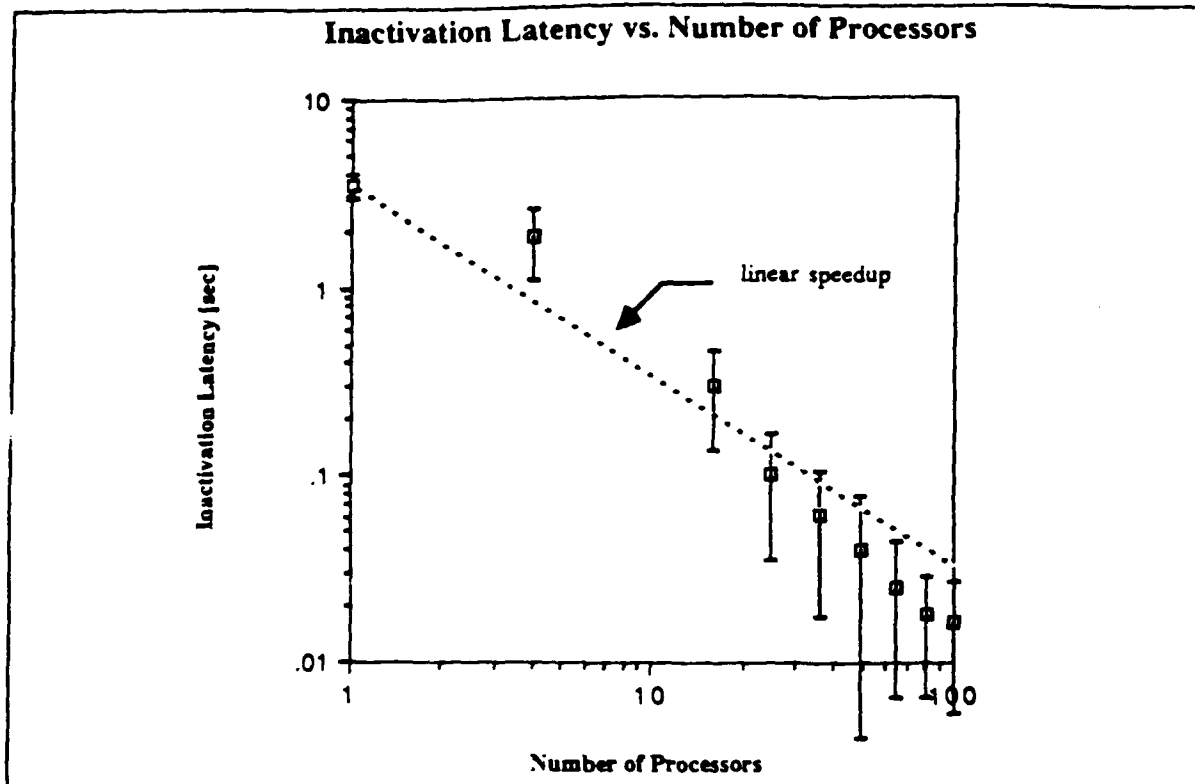
Figure 14.   Inactivation latency as a function of the number of processors.

This measures the duration from the time that the system receives a track report for the time following the last report for a given track id, to the time when the system detects that the track is no longer active, and asserts that conclusion.

7.2. Effects of replication

By replicating manager nodes, we measure the impact of the number of manager objects on performance, as measured by the confirmation latency. In one experiment we fix the number of Radar Track Managers at 4 while we vary the number of Input Handlers. In the other experiment we fix the number of Input Handlers at 4 while we vary the number of Radar Track Managers.

Figures 15 and 16 show the results. We plot the confirmation latency versus the number of managers, instead of against the number of processors as done in Figures 13 and 14.

Effect of Radar Track Managers on Confirmation Latency
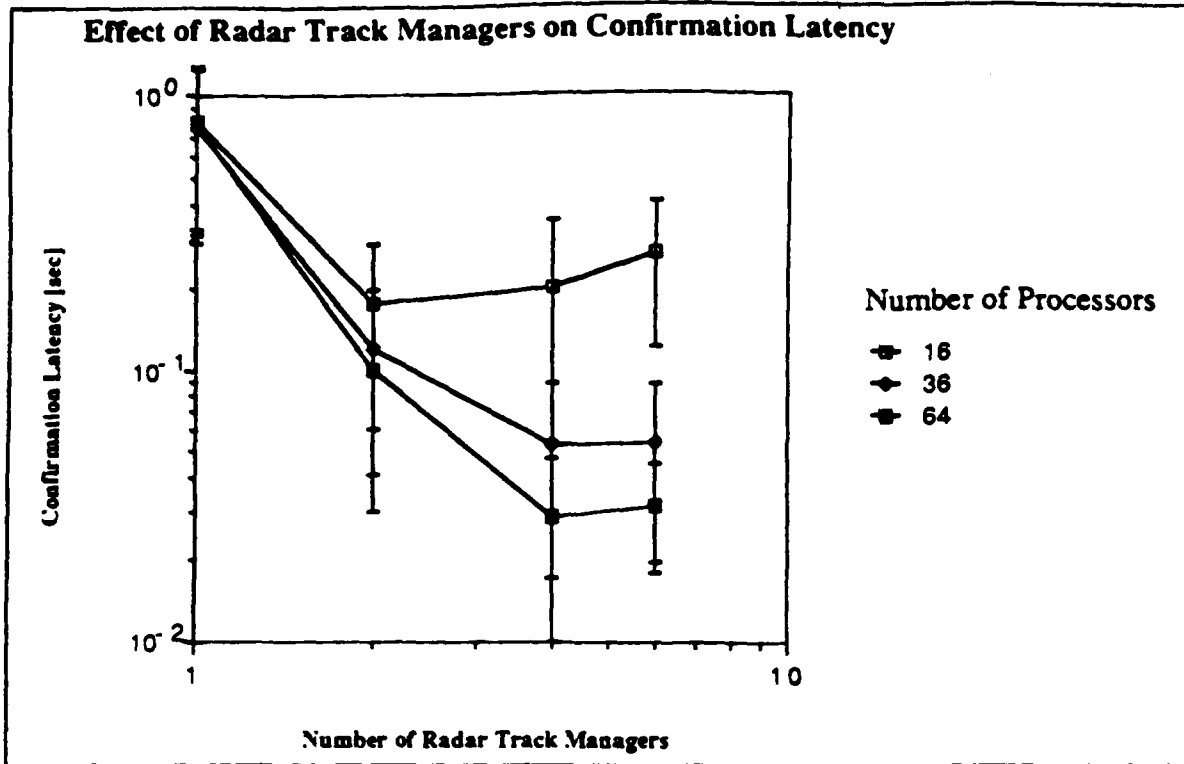
Figure 15.   Confirmation latency as a function of the number of radar track managers.

We see that replicating Radar Track Manager objects improves performance; this is because increasing the number of processors does not improve performance in the single Radar Track Manager case, but does in the 4 and 6 Radar Track Managers cases (see Figure 16). Put another way, if we had not used as many as 4 Radar Track Manager objects, then our system performance would have been hampered, and might even have precluded the high degree of speedup displayed in the previous section. Comparing Figures 15 and 16, we also observe that using more Radar Track Managers helps reduce confirmation latency more significantly than using more Input Handlers.

An interesting phenomenon occurs in the 16-processor case. Although the conclusion is not definitive given the size of the error bars, increasing the number of both types of managers from 2 to 4 and 6 increases the mean latency. The likely cause is the current object-to-processor allocation scheme: because each manager object is allocated to a distinct processor, increasing the number of manager objects decreases the number of processors available for other types of objects. Given our allocation scheme (described more fully in Section 8.2), using more managers in the 16-processor case may actually impede speedup.

Effect of Input Handlers on Confirmation Latency
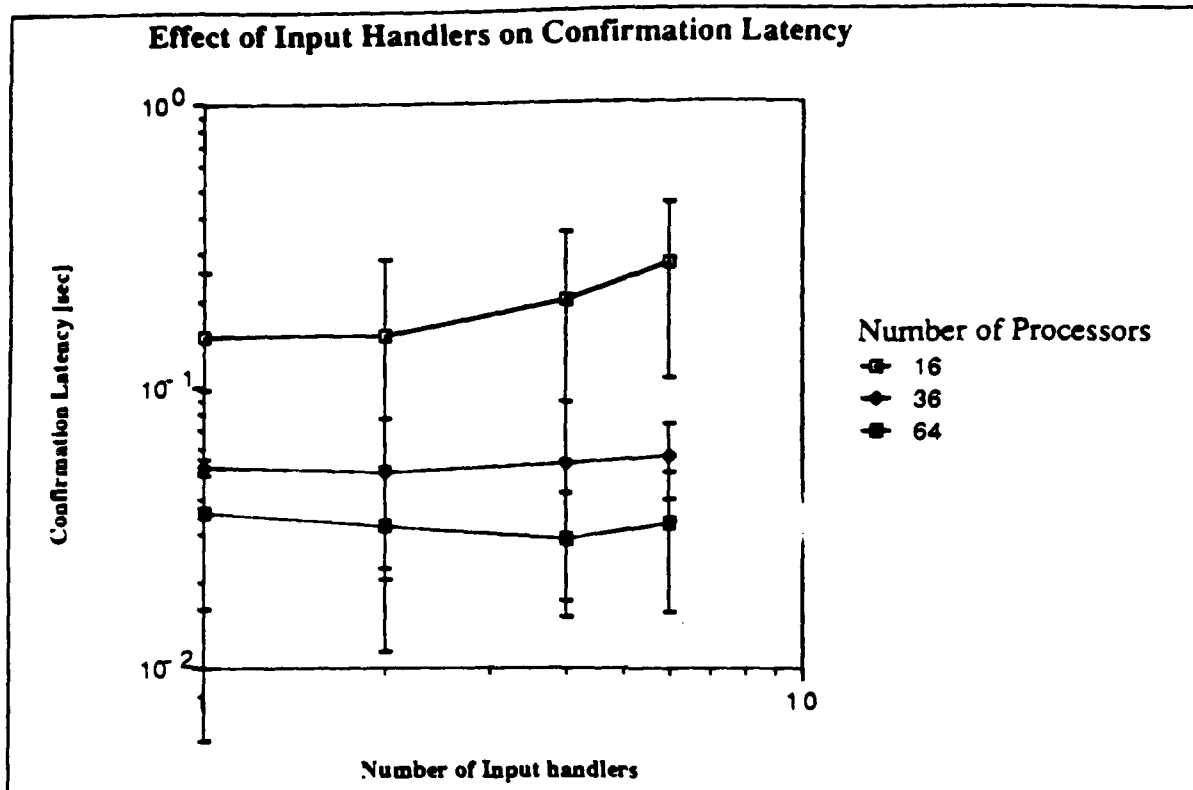
Figure 16.   Confirmation latency as a function of the number of input handlers.

The optimal number of manager objects appears to sometimes depend on the number of processors. For Radar Track Managers, 2 or 4 managers is best for the 16-processors array, and 4 or 6 managers is best for the 36 and 64-processor arrays. For Input Handlers, the number of managers does not appear to make much difference, which suggests that Input Handlers are less of a throughput bottleneck than Radar Track Managers. This suggests that in practice it will be necessary to consider the intensity of the managers' tasks relative to the total task in order to make a program work most efficiently. Overall these experiments confirm that replicating objects appropriately can improve performance.

7.3. Less than perfect correctness

Our Lamina program occasionally fails to confirm a track that our reference solution properly confirms. This arises because the concurrent program does not always detect the first occurrence of a report for a given track in the presence of disordered messages. We notice the following failure mechanism. Suppose we have a track consisting of scantimes 100, 110, 120, ..., 150. Suppose that the rate of data arrival is high, causing message order to be scrambled, and that reports for scantimes 110, 120. and 130 are received *before* the report for 100. As implemented, the Radar Track object notices that it has sufficient number of reports (in this case three), and it proceeds to compute a straight line through the reports. When a report for scantime 140 or higher is received, it is tested against the computed line to determine whether a line-check failure has occurred. Unfortunately, when the report for scantime 100 eventually arrives, it is discarded. It is discarded because the track has already been confirmed, and confirmed tracks only grow in the forward direction.

Figure 9 reveals why this error causes discrepancies between the Lamina program and the reference serial program: the handle track operation in the Lamina program is given a different set of reports compared to the reference program, leading to a different best-fit line being computed. To be certified as correct, we require that the reports contained in a confirmed Radar Track Segment must be identical between the Lamina solution and the reference solution.

The lesson here is that message disordering does occur, and that it does disrupt computations that rely on strict ordering of track reports. In our experiments, the incorrectness occurs infrequently. See Figure 17. We believe that with minimal impact on latency, this source of incorrectness can be eliminated without significant change to the experimental results.
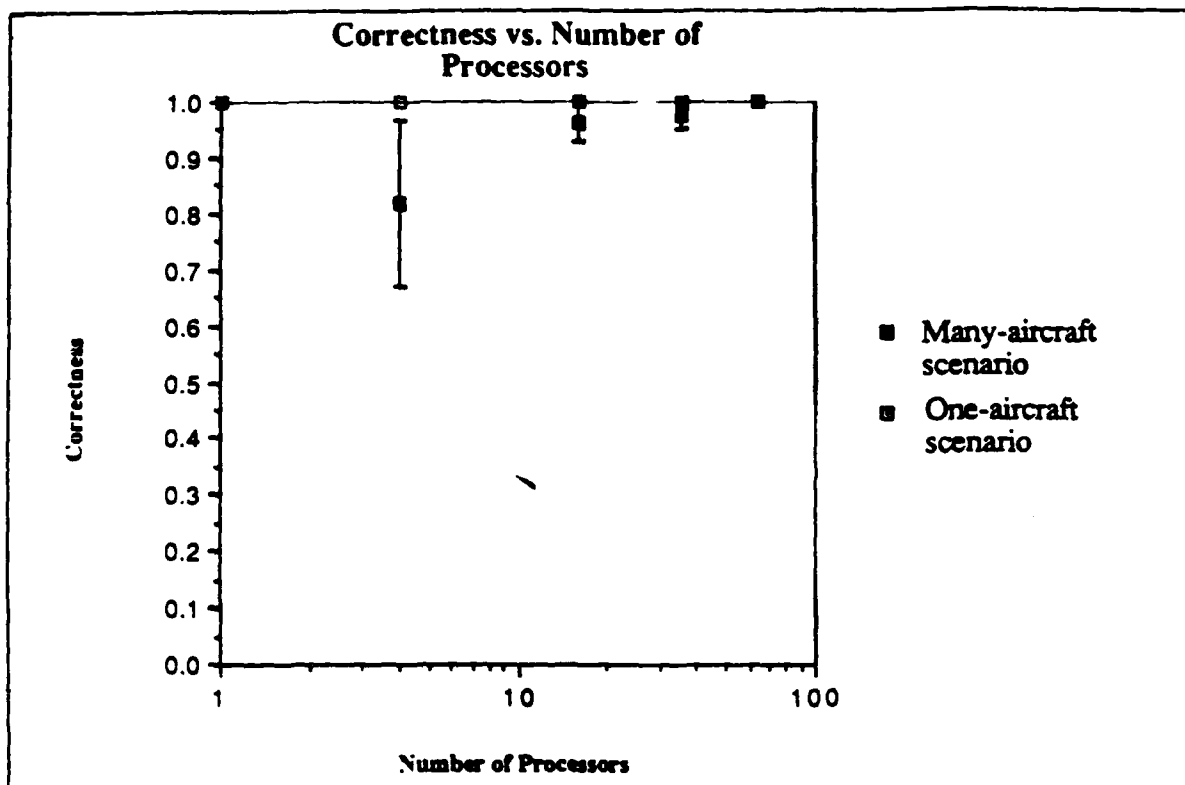
Figure 17.   Correctness plotted as a function of the number of processors for the one-aircraft and many-aircraft scenarios.

7.4. Varying the input data set

The results from using the one-aircraft scenario highlight the difficulties in measuring performance of a real-time system where inputs arrive over an interval instead of in a batch. Before experimentation began, we hypothesized that the amount of achievable speedup from additional processors is a function of the amount of parallelism inherent in the input data set. The results relative to this hypothesis are inconclusive. Figure 18 plots the confirmation latency against the number of processors for two input scenarios, the many-aircraft scenario (30 tracks per scan) and the one-aircraft scenario (1 track per scan).

Confirmation Latency vs. Number of Processors for Different Scenarios

- Many-aircraft scenario
- One-aircraft scenario
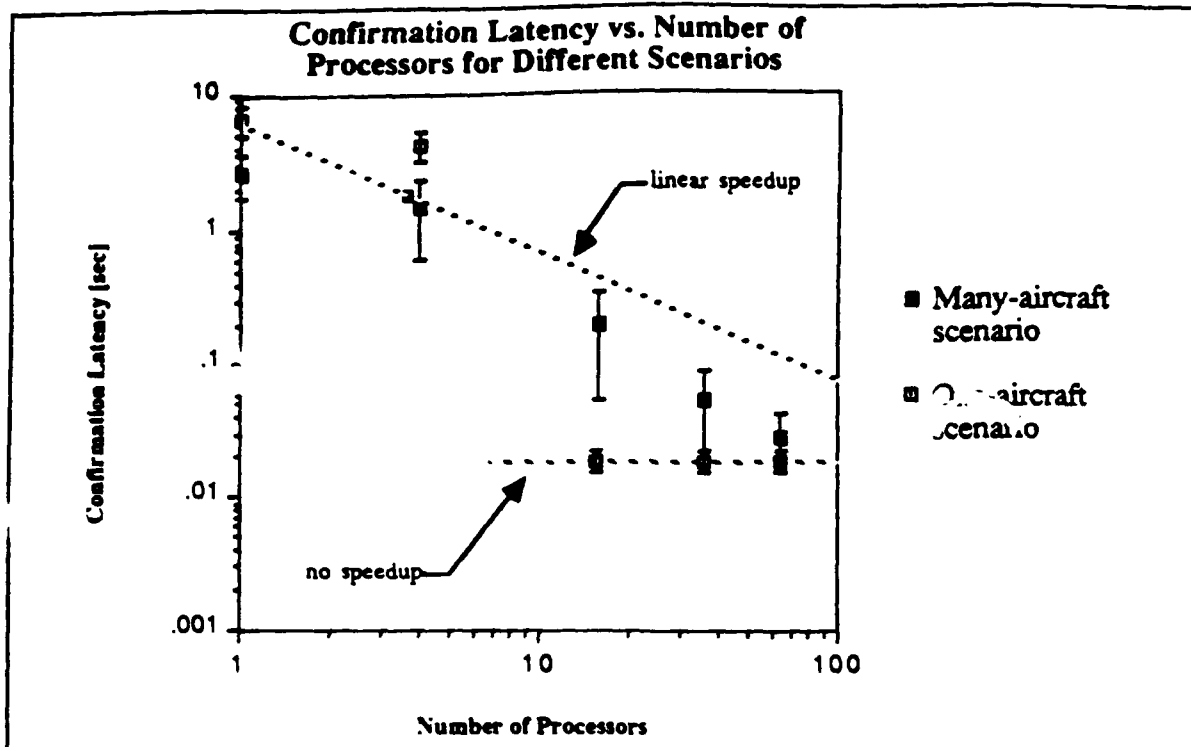
Figure 18. Confirmation latency as a function of the number of processors varies with the input scenario.

The one-aircraft scenario displays two distinct operating modes one in which processor availability and waiting time determines the latency, and another in which data can be processed with little waiting.

The one-aircraft scenario displays interesting behavior: see Figure 18. While the confirmation latency decreases from the 1-processor to 4-processor case, just as in the many-aircraft scenario, there is distinctly different behavior for 16, 36 and 64 processor cases, where the average latency is constant over this range. The key to understanding this phenomenon is to realize that inputs to the system arrive periodically. The many-aircraft scenario generates approximately 800 reports comprising 70 radar tracks over a 200 millisecond duration. In contrast, the one-aircraft scenario generates approximately 1300 reports comprising 70 radar tracks over an 8 second duration. Thus, although the volume of reports is roughly equivalent (800 versus 1300), the duration over which they enter the system differs by a factor of 40 (0.2 seconds versus 8 seconds). In terms of radar tracks per second, which is a good measure of the object-creation workload, the many-aircraft scenario produces data at a rate of 350 tracks per second, while the one-aircraft scenario produces data at a rate of 8.8 tracks per second. This disparity causes the many-aircraft scenario to keep the system busy, while the one-aircraft scenario meters a comparable inflow of data over a much longer period, during which the system may become quiescent while it awaits additional inputs.

The one-aircraft scenario displays two distinct operating modes: one in which processor availability and waiting time determines the latency, and another in which data can be processed with little waiting. For the 1-processor and 4-processor cases, the system cannot process the input workload as fast as it enters, causing work to back up. This explains why the average confirmation latency for the 70 or so radar tracks is nearly as long as the scenario itself: most of the latency is consumed in tasks waiting to be executed. In

contrast, for the 16-processor, 36-processor and 64-processor cases, there are sufficient computing resources available to allow work to be handled as fast as it enters the system. This explains why the average latency bottoms out at 18 milliseconds, and also tends to explain the small variance.

Recalling that this particular experiment sought to test the hypothesis that the amount of achievable speedup from additional processors is a function of the amount of parallelism inherent in the input data set, we see that these experimental results cannot confirm or disconfirm this hypothesis. The problem lies in the design of the one-aircraft input scenario. The reports should have been arranged to occur over the same 20 millisecond duration as in the many-aircraft scenario, instead of over an 8 second duration. Had that been done, the two scenarios would present to the system comparable workloads in terms of reports per second, but would differ internally in the degree to which sub-parts of the problem can be solved concurrently.

The distinction ⊦ veen the one-aircraft and many-aircraft scenarios can be described in Figure 19. .is graph is an abstract representation of Figure 12 presented earlier, and plots the input workload as a function of time. The many-aircraft scenario presents a high input workload over a very short duration, while the one-aircraft scenario presents the same total workload spread out over a much longer interval. If we imagine the dashed lines to represent the workload threshold for which an n-processor system is able to keep up without causing waiting times to increase, we see that the many-aircraft scenario exceeded the ability of the system to keep up even at the 100-processor level, but the one-aircraft scenario caused the system to transition from not-able-to-keep-up to able-to-keep-up somewhere between 4 and 16 processors. A more appropriate one-aircraft scenario, then, is one that has the same input workload profile as the current many-aircraft scenario. Such a scenario would allow an experiment to be performed that fixes the input workload profile, which our experiment inadvertently varied, thereby contaminating its results.
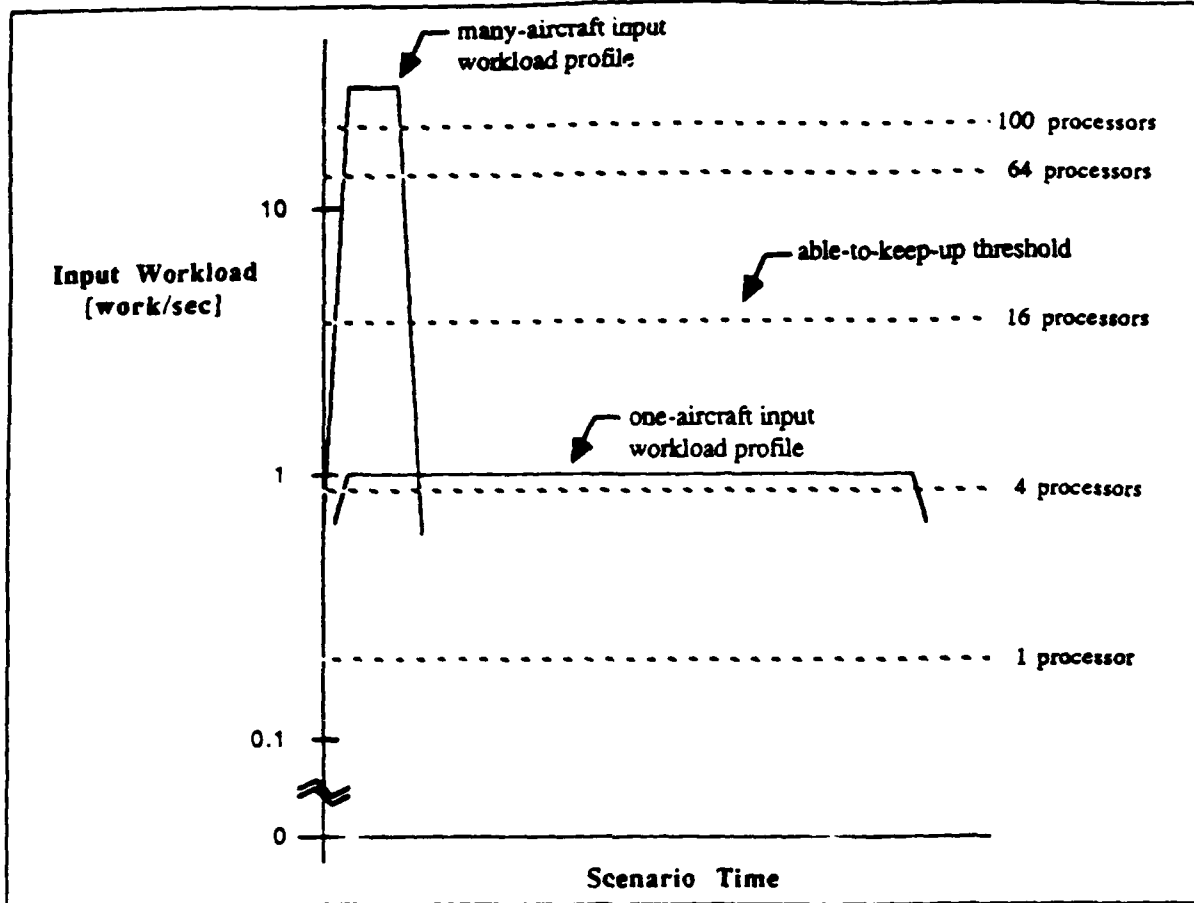
Figure 19.   Input workload versus time profiles shown for two possible input scenarios.

The workload threshold above which the work becomes increasingly backlogged varies according to the number of processors.

8.   Discussion

This section discusses how we achieved our experimental results using the concepts developed in Section 4. Specifically, we focus on the relationships between problem decomposition, speedup, and achievement of correctness.

8.1. Decomposition and correctness

In this section we analyze the problem solving knowledge embodied in the data association module. We use the dependence graph program to represent inherent dependencies in the problem. This is contrasted with the Lamina implementation to shed light on the rationale behind our design decisions. The goal is to identify the general principles that govern the transition from a dependence graph program to a runnable Lamina implementation.

8.1.1. Assigning functions to objects

We obtained speedup from both independent handling of tracks, and possibly from pipelining within a track, without the necessity to decompose the problem into the small functional pieces suggested in Figures 9 and 10. One might be tempted to believe that a direct translation of the nodes and edges of the dependence graphs into Lamina objects and methods might yield the maximal speedup, but careful study of the dependencies in Figures 9 and 10 reveals that there is very little concurrency to be gained.

In Figure 9, the entire graph is dependent on the arrival of report Ri. For instance, before a track is declared broken, the top-level "handle track" graph requires the arrival of reports R1, R2,....Rlast. The leftmost add node needs R1, and the remainder of the graph is dependent on this node. The add node to the right of this one is dependent on the arrival of R2, and the remaining right-hand subgraph is dependent on this node. This pattern holds for the entire graph, implying that computation may only proceed as far as consecutive reports beginning with R1 have arrived. Thus, little concurrency may be gained from the "handle track" operation; in particular, no pipelining is possible because the entire graph receives only one set of reports, R1,....Rlast. Figure 10 is similarly dependent on sequential processing of reports. We conclude that lumping all of the functions of Figures 9 and 10 into a small number of objects does not incur a great expense in concurrency. Given the overhead costs associated with message sending and process invocation, we speculate that one or two objects might yield the best possible design. In fact, our design uses k+2 objects, where k is the number of times a track is declared broken; k is typically fewer than three, giving us fewer than five objects for each "handle track" graph.

The dependence graph program provides several useful insights regarding a good problem decomposition. First, it justifies a decomposition that treats the "handle track" function as primitive function, rather than a finer-grained decomposition. Second, it clearly shows the independence between tracks, suggesting a relatively painless problem decomposition along these lines. Third, it shows the need to maintain consistent state about which tracks have been seen, and those which have not, suggesting a decomposition according to track id number, which is the approach that our Lamina program takes.

8.1.2. Why message order matters

A significant part of the Lamina concurrent program implements techniques to allow a Lamina object receiving messages from a single sender to handle them *as if* they were received in the order in which they were originally sent, without gaps the in the message sequence. By doing this, we incur a performance cost because the receiver waits for arrival of the next appropriate message, rather than immediately handling whatever has been received.

The dependence graphs help to justify such costs because the dependencies imply ordering. Indeed, in preliminary work in a different framework. one author discovered that when no explicit ordering constraints were imposed during Airtrac data association processing, and neither additional heuristics nor knowledge was used, incorrect conclusions resulted in cases when the input data rate was high. The incorrect conclusions arose from performing the line-fit computation on other reports different from the first three consecutive reports. As such, the incorrectness reflected an interaction between message disordering arising in CARE and the particular Airtrac knowledge, rather than the specific problem solving framework. We believe, for instance, that similar incorrect conclusions would arise in a Lamina program that did not explicitly reorder reports.

We emphasize that although the particular problem that we studied showed strong correctness benefits from imposing a strict ordering of reports, this should not be interpreted as a claim that all problems need or require message ordering. As the dependence graphs make strikingly clear, the very knowledge that we implement dictates ordering. Another problem may not require ordering, but require a strict message tagging protocol, for instance. As a general approach, we believe that the programmer should represent the given problem in dependence graph form, preferably explicitly, to expose the required set of dependencies, and let the overall pattern of dependencies suggest the kinds of decompositions and consistency requirements that might prove best.

8.1.3. Reports as values rather than objects

In the dependence graph program we represent reports as values sent from node to node. Similarly, in the Lamina implementation, we use a design where reports are values sent from object to object. This works well because reports never change, enabling us to treat reports as values. The cost of allowing an object to obtain the value of a report is a fairly inexpensive one-way message, where value-passing is viewed as a monotonic transfer of a predicate. This approach works because we know ahead of time which objects need to read the value of a report, namely the objects that constitute the processing pipeline.

Consider a second design where reports are represented as objects. In this scheme, instead of a report being a value passing through a processing pipeline, we arrange for read operations to be applied to an object. Conceptually these are identical problems, the only difference being the frame of reference. In the first case, the datum moves through processing stages requiring its value. In the case being considered here, the datum is stationary, and it responds to requests to read its value. This is attractive when it is not known in advance which objects will need to read its value. The penalty is an additional message required to request the object's value, and the associated message receipt system overhead.

A third design represents reports as objects, but replaces the read message in the previous design with a request to perform a computation, and uses the object's reply message to convey the result of the computation. By arranging a set of reports in a linear pipeline, we can allow the first report to send the results of its computation to the second report, and so forth. This design is the dual of the first design because in this design we send a sequence of computation messages through a pipeline of report objects, whereas in the first design we send a sequence of report value messages through a pipeline of computing objects. The designs differ in the grain-size of the problem decomposition; since our problem has a small number of computations and a large number of reports, the first design yields a small number of computing objects with many reports passing through, whereas the third design yields a large number of objects with a small number of computation messages passing through.

In our design, namely the first design discussed, we choose to represent reports as values sent to successive objects in a processing pipeline because our problem decomposition tells us in advance the objects in a pipeline. Using this design minimizes the number of messages required to accomplish our task, and uses a larger grain-size compared to its dual.

8.1.4. Initialization

Our approach to initialization embodies the correctness conditions of Schlichting and Schneider. Formally, we combine the use of monotonic predicates and predicate transfer with acknowledgement.

During initialization of our application, we create many objects, typically managers. At run-time, these objects communicate among themselves, which requires that we collect handles during creation, and distribute them after all creation is complete. Specifically, the Main Manager collects handles during the creation phase; in essence, each created object sends a monotonic predicate to the Main Manager asserting the value of its handle. The invariant condition may be expressed as follows:

Invariant (asserting own handle): "handle not sent" or "my handle is X"

The Main Manager detects the fact that all creation is complete when each of the predetermined number of objects respond; at this point, it distributes a table containing all the handles to each object. It waits until an acknowledgement is received from each object before initiating subsequent problem solving activity. This is important because if the Main Manager begins too soon, some object might not have the handle to another object that it needs to communicate with. In essence, the table of handles is asserted by a predicate transfer with acknowledgement. The invariant condition is described as follows:

Invariant (distributing table of handles):

"table not sent"

or "problem solving not initiated"

or "all acknowledgements received"
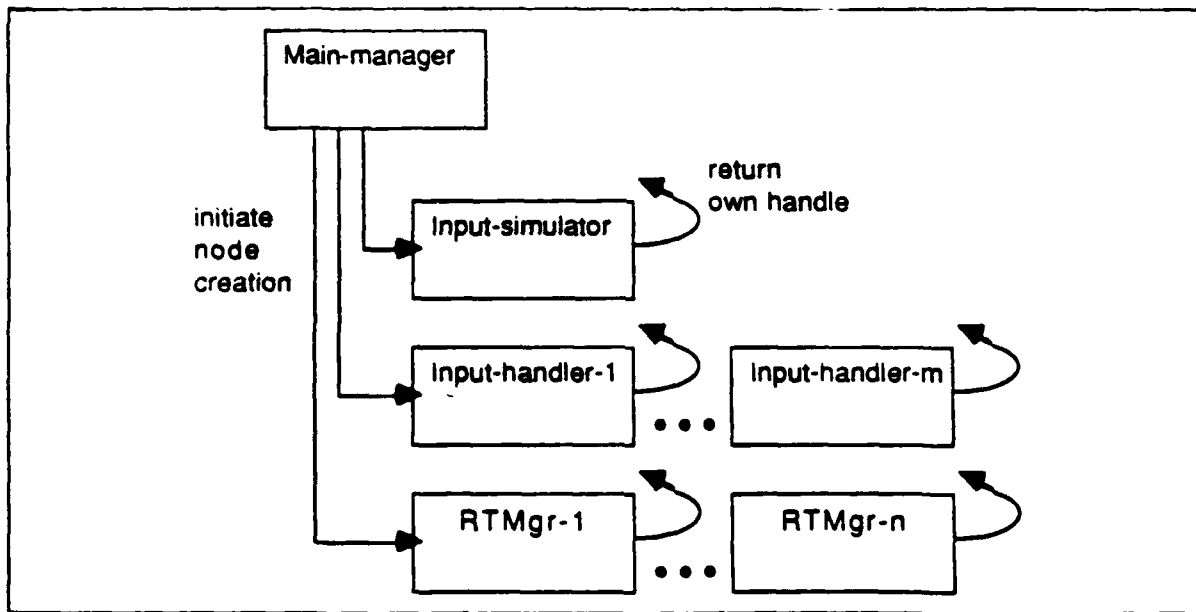
Figure 20.   Creating static objects during initialization.

Correctness is crucial during initialization because a missing or incorrect handle, or a missing or improperly created object causes problems at run-time. These problems can compound themselves, causing performance or correctness degradation to propagate. By

using an initialization protocol that is guaranteed to be correct, these problems may be avoided.

8.2. Other issues

8.2.1.   Load balance

We define *load balance* as how evenly the actual computational load is distributed over the processors in an array over time. Processing load is balanced when each processor has a mix of processes resident on it that makes all the processors equally busy. If a balanced processing cannot be achieved, the overall performance of a multiprocessor may not reflect the actual number of processors available to perform work due to poor load balance. In our experimentation, we discovered the critical importance of a good load balance algorithm.

We encountered two kinds of problems. The first problem deals with where to place a newly created object. Since we want to allocate objects to processors so as to evenly distribute the load, and because we want to avoid the message overhead associated with a centralized object/processor assignment facility, we focused on the class of algorithms that make object-to-processor assignments based on *local* information available to the processor creating the object. The second problem deals with how objects share limited processor resources. It turns out, for instance, that extremely computation-intensive objects can severely impair the performance of all other objects that share its processor.

At one point in our experimentation, for instance, we observed a disappointing value of unity for the $S_{64/16}$ speedup factor, where we instead expected a factor of 4. Moreover, we noticed an extremely uneven mapping of processes to processors: the approximately 200 objects created during the course of problem solving ended up crowded on only 14 of the 64 available processors! The culprit was the algorithm that decided which neighboring processor should be chosen to place a new object. The algorithm worked as follows. Beginning with the first object created by the system, a process-local data structure, called a *locale*, is created that essentially records how many objects are already located at every other processor in the processing array. When a new process is spawned, the locale data structure is consulted to choose a processor that has the fewest existing processes. This scheme works well when a single object creates all other objects in the system; unfortunately in Airtrac many objects may create new objects.

Given the locale for any given process, when the process spawns a new process, we arranged for the new process to inherit the locale of its parent. The idea is that we want the new process to "know" as much as its parent did about where objects are already placed in the array. This scheme fails because of the tree-like pattern of creations. Beginning with the initial manager object at the root of the tree, any given object has inherited a locale through all of its ancestors between itself and the root. Therefore the locale on a given object will only *know* about other objects that were created by the ancestors of the object *before* the locale was passed down to the next generation. Put another way, the locale on a given object will not reflect creations that were performed on non-ancestor objects, or creations that were performed on ancestor objects after the locale was passed down. This leads to extremely poor load balance.

The same problem occurs even if we define a single locale for each processor that is shared over all processes residing on that processor. Unfortunately, that locale will only know about other objects that were created by objects residing on that processor. That is,

the locale on a given processor will not reflect creations that were performed by objects that reside on *other* processors.

In contrast, ideal load balance occurs when each object knows about all creations that have taken place in the past over the entire processing array. This ideal is extremely difficult to achieve. First, we want to avoid using a single globally-shared data structure. Second, finite message sending time makes it impossible for many objects performing simultaneous object creation to access and update a globally-shared structure in a perfectly consistent manner.

We changed to a "random" load balance scheme which randomly selected a processor in the processing array on which to create a new object [Hailperin 87]. Running the base case on a 64 processor array with approximately 200 objects, we managed to use nearly all the available processors. Processor utilization improved dramatically.

Random processor allocation gave us good performance. In fact, we can argue from theoretical grounds that a random scheme is desirable. First, we deliberately constrain the technique to avoid using global information that would need to be shared. This immediately rules out any cooperative schemes that rely on sharing of information. Second, any scheme that attempts to use local information available from a given number of close neighbors and performs allocations locally faces the risk that some small neighborhood in the processing array might be heavily used, leaving entire sections of the array underutilized. We are left therefore, with the class of schemes that avoids use of shared information but allows any processor to select any other processor in the entire array. Given these constraints, a random scheme fits the criteria quite nicely and in fact performed reasonably well.

Further experimentation revealed more problems. Manager objects have a particularly high processing load because a very small number of objects (typically 5 to 9) handles the entire flow of data. When a non-manager objects happens to reside on the same processor as a manager object, its performance suffers. For example, a Radar Track object is responsible for creating a Radar Track Segment object, and the time taken for the create operation affects the confirmation performance. Unfortunately, any Radar Track object that happens to be situated on the same processor as a manager object (e.g. Input Handler, Radar Track Manager) gets very little processor time, and thereby contributes significant creation times to the overall latency measure.

Whereas in the random scheme the probability that a given processor will be chosen for a new object is $\frac{1}{n}$ for n processors, our modified random scheme does the following:

- If there are fewer static objects (e.g. managers) than processors, then place static objects randomly, which can be thought of as sampling a random variable *without replacement*. Place dynamically created objects uniformly on the processors that have no static objects, this time sampling *with replacement*.

- If there are as many or more static objects than processors, then place roughly equal numbers of static objects on each processor in the array. Place dynamically created objects uniformly over the entire array, sampling *with replacement*.

This scheme keeps the high processing load associated with manager objects from degrading the performance of non-manager objects. This scheme performs well for our

cases. We typically had from 5 to 9 static objects, approximately 150 dynamic objects, and from 1 to 100 processors in the array.

There are other considerations that might lead to further improvement in load balance performance that we did not pursue. These are listed below:

- Account for the fact that not all static objects need a dedicated processor. (In our scheme, we gave each static object an entire processor to itself whenever possible.)

- Account for the fact that a processor that hosts one or more static objects may still be a desirable location for a dynamically created object, although less so than a processor without any static objects. (In our scheme, we assumed that any processor with a static object should be avoided if possible.)

- Relocate objects dynamically based on load information gathered at run-time.

8.2.2.   Conclusion retraction

This section explores some of the thinking behind our approach toward consistency, which is to make conclusions (e.g. confirmation, inactivation) only when they were true. This is an extremely conservative stance, and possibly incurs a loss in concurrency and speedup. An alternative approach which might allow more concurrency is to make conclusions that are not provably correct: the programmer would allow such conclusions to be asserted, retracted and reasserted freely until a commitment regarding that conclusion is made. Jefferson has explored this compuational paradigm, known as *virtual time* [Jefferson 85]. The invariant condition describing the truth value of a conclusion P under such a scheme is shown below:

Invariant: "no commitment made" or "P is true"

In essence, this invariant condition says that the program may *assert* that P is true, but there is no guarantee that P is true unless it is accompanied by a *commitment* to that fact. The benefits of such an approach is that assertions may precede their corresponding commitments by some time interval. This interval may be used 1) by the user of the system in some fashion, or 2) by the program itself to engage in further exploratory computation that may be beneficial, perhaps in reducing computation later. In Airtrac-Lamina, we did not investigate the benefits from exploratory computation.

For the user of the system, he or she must decide how and when to act upon uncommitted assertions rendered by the system. On one hand, the user could view assertions as true statements even before a commitment is made, with the anticipation that a retraction may be forthcoming. On the other hand, the user could vie .v an assertion as true only when accompanied by a commitment; this latter approach places emphasis on the commitment, since only the commitment assures the truth of the conclusion.

We decided against using the scheme outlined here. As a technique to allow concurrent programs to engage in exploratory computations, there might be some merit if the power of such computations can be exploited. As a logical statement to the user of the system, such an uncommitted conclusion is meaningless, since it may later be retracted. As a probabilistic statement to the user of the system, a conclusion without commitment might indicate some likelihood that the conclusion is true. However, we believe that a better way to handle probabilistic knowledge is to state it directly in the problem rather than in the consistency conditions that characterize the solution technique. This unclear separation

between domain knowledge and concurrent programming techniques steered us away from the approach of making assertions with the possibility of subsequent retraction.

9. Summary

Lamina programming is shaped by the target machine architecture. Lamina is designed to run on a distributed-memory multiprocessor consisting of 10 to 1000 processors. Each processor is a computer with its own local memory and instruction stream. There is no global shared memory; all processes communicate by message passing. This target machine environment encourages a programming style that stresses performance gains through problem decomposition, which allows many processors to be brought to bear on a problem. The key is to distribute the processing load over replicated objects, and to increase throughput by building pipelined sequences of objects that handle stages of problem solving.

For the programmer, Lamina provides a concurrent object-oriented programming model. Programming within Lamina has fundamental differences with respect to conventional systems:

- Concurrent processes may execute during both object creation and message sending.

- The time required to create an object is visible to the programmer.

- The time required to send a message is visible to the programmer.

- Messages may be received in a different order from which they were sent.

The many processes which must cooperate to accomplish the overall problem-solving goal may execute simultaneously. The programmer-visible time delays are significant within the Lamina paradigm because of the activities that may go on *during* these periods, and they exert a strong influence on the programming style.

This paper developed a set of concepts that allows us to understand and analyze the lessons that we learned in the design, implementation, and execution of a simulated real-time application. We confirmed the following experimental hypotheses:

- Performance of our concurrent program improves with additional processors, we attain significant levels of speedup.

- Correctness of our concurrent program can be maintained despite a high degree of problem decomposition and highly overloaded input data conditions.

An inappropriate design of our one-aircraft scenario precluded us from confirming or disconfirming the following experimental hypothesis:

- The amount of speedup we can achieve from additional processors is a function of the amount of parallelism inherent in the input data set.

In building a simulated real-time application in Lamina, we focused on improving performance of a data-driven problem drawn from the domain of real-time radar track understanding, where the concern is throughput. We learned how to recognize the

symptoms of throughput bottlenecks; our solution replicates objects and thereby improves throughput. We applied concepts of pipelining and replication to decompose our problem to obtain concurrency and speedup. We maintained a high level of correctness by applying concepts of consistency and mutual exclusion to analyze and implement the techniques of monotonic predicate and predicate transfer with acknowledgements. We recognized and repaired load balance problems, discovering in the process that a modified random processor selection scheme does fairly well.

The achievement of linear speedup up to 100 times that obtainable on a single processor serves as an important validation of our concepts and techniques. We hope that the concepts and techniques that we developed, as well as the lessons we learned through our experiments, will be useful to others working in the field of symbolic parallel processing.

Acknowledgements

References

[Andrews 83]   G.R. Andrews and Fred B. Schneider, Concepts and notations for concurrent programming *Computing Surveys* 15 (1) (March 1983) 3-43.

[Arvind 83]   Arvind, R.A. Iannucci, Two fundamental issues in multiprocessing: the data flow solution, Technical Report MIT/LCS/TM-241, Laboratory for Computer Science, Massachusetts Institute of Technology, September 1983.

[Brown 86].   H. Brown, E. Schoen, B. Delagi. An experiment in knowledge-based signal understanding using parallel architectures, Report No. STAN-CS-86-1136 (also numbered KSL 86-69), Department of Computer Science, Stanford University, 1986.

[Broy 85]   M. Broy ed., *Control Flow and Data Flow: Concepts of Distributed Programming* (Springer-Verlag, Berlin, 1985).

[Byrd 87].   G. Byrd, R. Nakano, B. Delagi, A dynamic, cut-through
                    communications protocol with multicast, Technical Report KSL 87-
                    44, Knowledge Systems Laboratory, Stanford University, 1987.

[Cornafion 85]   CORNAFION, *Distributed Computing Systems: Communication,
                    Cooperation, Consistency* (Elsevier Science Publishers, Amsterdam.
                    1985).

[Delagi 87a]   B. Delagi, N. Saraiya, S. Nishimura, G Byrd, An instrumented
                    architectural simulation system, Technical Report KSL 86-36,
                    Knowledge Systems Laboratory, Stanford University, January 1987.

[Delagi 87b]   B. Delagi and N. Saraiya, Lamina: CARE applications interface,
                    Technical Report KSL 86-67, Working Paper, Knowledge Systems
                    Laboratory, Stanford University, May 1987.

[Delagi 87c]   B. Delagi, private communication, July 1987.

[Dennis 85]   J.B. Dennis, Data flow computation, Manfred Broy ed., *Control
                    Flow and Data Flow: Concepts of Distributed Programming*
                    (Springer-Verlag, Berlin. 1985) 345-54.

[Filman 84]   R.E. Filman and D.P. Friedman, *Coordinated Computing: Tools and
                    Techniques for Distributed Software* (McGraw-Hill Book Co., New
                    York, 1984).

[Gajski 82]   D.D. Gajski, D.A. Padua, D.J. Kuck, R.H. Kuhn, A second opinion
                    on data flow machines and languages, *IEEE Computer* (February
                    1982) 58-69.

[Hailperin 87]   M. Hailperin, private communication, July 1987.

[Henderson 80]   P. Henderson, *Functional Programming* (Prentice-Hall International,
                    Englewood Cliffs, 1980).

[Jefferson 85]   D.R. Jefferson, Virtual time, *ACM Transactions on Programming
                    Languages and Systems* 7 (3) (July 1985) 404-25.

[Kruskal 85]   C.P. Kruskal, Performance bounds on parallel processors: an
                    optimistic view, Manfred Broy ed., *Control Flow and Data Flow:
                    Concepts of Distributed Programming* (Springer-Verlag, Berlin.
                    1985) 331-44.

[Kung 82]   H.T. Kung, Why systolic architectures?, *IEEE Computer*. (January
                    1982) 37-46.

[MacLennan 82]   B.J. MacLennan, Values and objects in programming languages,
                    ACM Sigplan Notices 17 (12) (December 1982).

[Minami 87]   M. Minami Experiments with a knowledge-based system on a
                    multiprocessor: preliminary Airtrac-Lamina quantitative results,
                    Working Paper, Technical Report KSL 87-35, Knowledge Systems
                    Laboratory, Stanford University, 1987.

[Nakano 87]   R. Nakano, Experiments with a knowledge-based system on a multiprocessor: preliminary Airtrac-Lamina qualitative results, Working Paper, Technical Report KSL 87-34, Knowledge Systems Laboratory, Stanford University, 1987.

[Nii 86a]   P. Nii, Blackboard systems: the blackboard model of problem solving and the evolution of blackboard architectures, *AI Magazine* 7 (2) (1986) 38-53.

[Nii 86b]   P. Nii, Blackboard systems part two: blackboard application systems, blackboard systems from a knowledge engineering perspective, *AI Magazine* 7 (3) (1986) 82-106.

[Schlichting 83]   R.D. Schlichting and F.B. Schneider, Using message passing for distributed programming: proof rules and disciplines, Technical Report TR 82-491, Department of Computer Science, Cornell University, May 198.

[Smith 81]   R.G. Smith, *A Framework for Distributed Problem Solving* (UMI Research Press, Ann Arbor, Michigan, 1981).

[Tanenbaum 81]   A. Tanenbaum, *Computer Networks* (Prentice Hall, Englewood Cliffs, New Jersey, 1981).

[Weihl 85]   W. Weihl and B. Liskov, Implementation of resilient atomic data types, *ACM Trans. on Programming Languages and Systems* 7 (2) (April 1985) 244-69.

[Weinreb 80]   D. Weinreb and D. Moon, Flavors: message passing in the Lisp machine, Technical Report, Memo 602. Massachusetts Institute of Technology. Artificial Intelligence Laboratory, 1980.

Problems with Problem-Solving in Parallel: The Poligon System

by

J. P. Rice

Knowledge Systems Laboratory
Computer Science Department
Stanford University
Stanford, California 94305

Table of Contents

List of Figures

Abstract

This paper describes the desire to speed up programs in the field of *Artificial Intelligence* through the use of parallel hardware architectures and why this objective is not a simple one to achieve.

Poligon, a system designed to investigate ways to mount Artificial Intelligence programs on parallel hardware, is described, experiments performed to date on this system are described and tentative results are given.

Achieving useful speed-up has proven very difficult. These difficulties are enumerated and explained.[1]

1. Introduction

The domain of supercomputing has traditionally been very large regular problems. This has been driven by two main forces;

- A large class of important problems were soluble by existing programming technology but were intractable with *"normal"* processors, e.g. PDE solution, finite element analysis or simulation.

- Early programming languages focused on *Arrays* as data structures, whose use could efficiently use the hardware available. This led to the idea of *vector* and *array* processors.

It is, therefore, by no means a coincidence that the sort of problems that tend to use existing supercomputers are those problems best suited to supercomputers.

The field is changing now, however. This is driven by two main forces;

- Developments in hardware technology now allow the development of multiprocessor systems composed of large numbers of relatively simple processors, which are potentially more cost effective than existing super-complex supercomputer uniprocessors.

- Both hardware and software technologies have progressed to a point where a number of problems which have become soluble by means of symbolic programming would now like a slice of the speed-up cake.

Symbolic computation has for a long time been accused of inefficiency. Recent developments in compiler and hardware technologies, however, have allowed the development of high performance uniprocessor workstations for the execution of symbolic programs. These have shown that there is a large class of *"Artificial Intelligence"* (*AI*) problems for which significantly greater computational resources will be needed to make these problems worth addressing. This has focused the attention of *AI* and symbolic programming research on the exploitation of parallelism.

The sort of problem currently applied to supercomputers is very crystalline [Seitz 85] in nature. This means that a relatively small "inner loop" of the computation can be vectorized in order to exploit existing supercomputer hardware [Kuck 81]. Similarly such problems can often exploit parallelism at a finer grain in a systolic manner [Kung 78].

AI problems have none of these useful characteristics [Lee 85]. This paper describes first what is meant by *"Problem-Solving"* and how this relates to parallelism (§2). It goes on to describe Poligon [Rice 86] a system implemented in order to investigate the potential for speed-up of a class of *AI* applications called *"Blackboard Systems"* through parallelism (§3).

---

[1]This paper also appears in the proceedings of the *Third International Conference on Supercomputing, May 1988*

After this some preliminary experiments and what we have learned from them and discussed (§4).

2. Parallelism and Problem-Solving

In this section we examine what is meant by "Problem-Solving", contrasting it with common supercomputing doctrine and concerns. This will show why it is that a different approach to parallelism than is taken by conventional programs is necessary in *AI* and also why it is so hard to achieve.

2.1. What is "Problem-Solving"?

Questions are never indiscreet. Answers sometimes are. - Oscar Wilde, *"An Ideal Husband"*

"Problem-Solving" was often taken to refer to the process of searching a tree or graph of alternative solutions. *"Knowledge"* is that which allows the program to eliminate searching parts of the tree. For instance, a chess playing program might have a tree made of all of the legal moves at any given point[2]. The term "Knowledge" will always be used in this sense in this paper. The application of strategic Knowledge, such as Knowledge about chess end games, to each generated node in the tree would point out to the system likely candidate paths to follow. The method of constructing all legal possibilities at any given leaf of a dynamically generated tree and then testing them to determine whether they are possibilities worth following is usually referred to as the *"Generate and Test"* method. It is an axiom of such systems that the more Knowledge there is the less blind search has to be done - the more efficiently the tree is pruned.

The focus of much *AI* research is on the use of Knowledge to reduce or obviate search. This is because such searches are expensive and combinatorial processes. The use of Knowledge in this way might not be the best solution for the future since the use of highly parallel architectures to evaluate multiple alternatives might be faster than executing this highly specialized Knowledge. What is more, this could also save the human cost of acquiring and encoding such Knowledge. The acquisition of Knowledge is generally thought to be one of the major obstacles in the way of the more general application of *AI* systems to real-world problems.

The important thing, for the purpose of this paper, about problem-solving systems and the problems that they address is that they are structurally different from *"conventional"* programs. Throughout this paper the terms *"Problem-Solving"* and *"AI system"* will be used to describe these systems. The term *"Conventional"* will be used to describe existing practice in the supercomputer world. Some of the characteristics that make such a problem different from a conventional programming problem are listed below.

- The problem itself is often ill-defined.

- There is often more than one possible solution. This means that a satisficing[3], rather than an optimal solution is usually the "right" answer. This is quite unlike most conventional programs for which there is one and only one right answer, within the margin of error of the system[4].

- The paths to a solution cannot predefined in such systems. Possible solution paths must be dynamically generated and tried.

---

[2] Clearly this tree cannot be fully instantiated with the resources available in the universe.

[3] A solution that is said to be *"good enough."*

[4] Linear optimization is a notable exception to this. Clearly many programs use heuristics and so the distinction made here is simply one of degree. *AI* problems are usually composed of a larger proportion of heuristics than conventional programs.

- The structure of such programs differ from conventional programs in three fundamental ways; in their data structures, their control flow and their control structures.

Data Structure   It is generally the case that the data upon which the system has to operate cannot be encoded simply into an array. This is because such data structures are usually highly complex and often cyclic graphs, which are created dynamically, thus precluding static allocation and optimization.

Control Flow   The solution to the problem is not regular, which is to say that the behavior of the problem-solver is typically very data-dependent. In a PDE solving program, for instance, the computational demands of the system at any point are well understood. This is because well defined and well understood algorithms are used and the computational demands of matrix inversion, for example, are reasonably easy to estimate. This is not the case in *AI* programs. Apparently trivial changes to the source data can cause huge changes to the computation performed. As an example of this one might consider the behavior of a chess program when the opponent elects to make an unexpected move. What is more, the code generated for these programs is usually very branchy [Lee 85], thus reducing the benefits of fine grained pipe-lining.

Control Structures

The Knowledge that *AI* programmers try to encode in their programs is usually functionally different from that Knowledge which is usually encoded in conventional programs. That is to say it is more likely to be a high-level specification of the intended behavior of the system, as opposed to a set of instructions for how to compute the answer. Such details are usually left to the system. For instance, the program might be compiled into a set of assertions and rules in a Prolog system [Clocksin 81]. The program itself is executed indirectly through a virtual machine which interprets these specifications as its instructions. This results in most of such programs not being amenable either to existing vectorizing algorithms or to the application of well defined algorithms[5].

The factors mentioned above result in *AI* problems not having the properties needed for them to be parallelized by conventional means. This is cause for considerable concern for those who would like to achieve orders of magnitude of speed-up for their *AI* programs.

2.2. Concerns for Supercomputers

On how to trap a lion in a desert [Petard 38]: A topological method. *We observe that a lion has at least the connectivity of the torus. We transport the desert into four-space. It is then possible [Seifert 34] to carry out such a deformation that the lion can be returned to three-space in a knotted condition. He is then helpless.*

Implementors and programmers of supercomputers have traditionally focused on the efficient use of the hardware and the matching of the hardware to the problem. Some examples of these are discussed below.

---

[5]These interpreters themselves may, however, be implemented using well understood algorithms or microcode.

2.2.1. Where does parallelism come from?

Parallelism in conventional programs is either easy to get or nearly impossible. If the program does a lot of simple operations on arrays whose dependencies and recurrences are simple and can be unraveled then massive data parallelism[6] can be exploited. It is by this means that vector machines are able to achieve their performance. It is not generally the case that there is, qualitatively speaking, more than one thing happening at any given time. Such programs are parallel in a SIMD sense [Flynn 72]. If the control flow is too complex to analyze then the compiler may not be able to unwind the parallelism out of the program[7].

AI programs are typically short on data parallelism. There are certainly problems which have significant data parallelism but not of the order that one might get in extremely regular, conventional programs. This means that an *AI* system which hopes for speed-up through parallelism must be able to exploit *Knowledge parallelism*. It must be able to execute a significant number of different chunks of the program simultaneously. This is MIMD parallelism. The **Poligon** system described in §3 is designed to exploit this sort of parallelism[8].

Most high performance processors today exploit pipe-line parallelism in the execution of instructions. Pipe-line parallelism is also exploited at a somewhat coarser grain by the new generations of multiprocessor systems such as systolic arrays. It is crucial that any system hoping to exploit parallel hardware effectively should be able to exploit pipe-line parallelism. This is, in fact, considerably harder in *AI* systems because of the irregular structure of the problem. The **Poligon** system tries wherever it can to exploit pipe-line parallelism.

2.2.2. What sort of hardware should be used?

In order to be able to exploit the parallelism in a program to the best possible degree there must be an appropriate match between the compiled program and the target hardware. This means that if a speed-up of no more that 10 to 20 is either hoped for or expected then the program should probably be executed on a shared-memory multiprocessor[9]. If more speed-up than this is needed then a hardware design that will scale better should be used – some form of distributed memory architecture[10]. This could, in practice, have a grain size varying from that of the **Cosmic Cube** [Seitz 85] to that of the **Connection Machine** [Hillis 85]. The **Poligon** system is designed to be matched to run on a multiprocessor, which should scale satisfactorally to the order of hundreds or thousands of processing elements, each element being a highly competent symbolic language processor. This is the pure value passing **CARE** machine model [Byrd 87], one of several **CARE** machine models implemented as part of the same project of which **Poligon** is a part.

2.2.3. Compilation

Vectorizing FORTRAN compilers have been the main implementation language in supercomputing circles for quite some time. There is considerable inertia in the field in this respect. Similarly *AI* programmers are in many senses locked into the use of Lisp [Steele 84] or Prolog [Clocksin 81] as their implementation languages. Problem-Solving systems have traditionally

---

[6]Parallelism due to similar operations being performable on independent items of data, for instance elementwise addition of two arrays.

[7]The Connection Machine [Hillis 85] is an example of an experiment to test the contrary hypothesis, that SIMD machines are, indeed, appropriate for *AI* applications.

[8]MIMD programs typically have a set of implementation difficulties and bugs which are not so frequently seen in SIMD programs. These are caused by having a number of radically different types of program executing, all at different speeds and trying to communicate with one another. This causes data to arrive "out of order" and race conditions. Many of the pit-falls of parallel *AI* programming mentioned in this paper are a consequence of this.

[9]Some experiments have shown rather disappointing results here, saying that this is all that can really be hoped for. [Gupta 86]

[10]Recent claims have been made that some shared memory architectures can scale well [Wilson 87].

not been very efficiently implemented, even if the underlying implementation language has been. This is because it is expensive in human terms to implement such systems efficiently and their typical life span has not justified this sort of optimization effort. This state of affairs is beginning to change. There is now a demand for highly competent programs using *AI* techniques being embedded, for instance, into military hardware. This asks not only for high performance but also for high reliability, maintainability and modifiability. Lisp and Prolog in their common implementations are not languages which can easily be parallelized in the same way that FORTRAN compilers are[11]. There is, therefore, a need to develop languages not only capable of exploiting the parallelism in forthcoming hardware but also capable of expressing the richness of these complex symbolic programs. On top of these will need to be built highly competent tools and frameworks which will be needed for a satisfactory parallel *AI* development environment. The **Poligon** system is a first-cut prototype system developed with the objective of being able to extract parallelism from programs both by the system and by encouraging a clear programming style and problem decomposition methodology, which leads to more parallel programs.

2.3. Concerns for Problem-Solvers

The concerns of the implementors of Problem-Solving systems are quite different from those of supercomputer programmers. Some of these concerns are enumerated below.

2.3.1. Solution quality

As has been mentioned above, *AI* programs are generally expected to produce a satisficing solution. This has a significant impact on the behavior of the program, since paths used to determine heuristic solutions might be very different from those used to find analytic solutions, even if analytic solutions are known.

2.3.2. Search

These heuristic programs are typically characterized by searching a great deal for patterns over a large graph[12]. This large amount of search admits both *And* and *Or* parallelism, in principle. The **Poligon** system has specific mechanisms to facilitate the efficient execution of such searches[13].

2.3.3. Coherence

The implementor of an *AI* program may not be aware of the eventual behavior of his program when he is implementing it. This is a function of the complex nature of such problems and the fact that the paths to their solutions are not predefined. It is, nevertheless, very important that the program reach a coherent solution, even if just a satisficing one. It is no good if different parts of the solution space have mutually contradictory local solutions which contribute to the overall solution. Because the Knowledge that goes into such systems is usually implemented in distinct chunks, which may know little about the operations performed by other such chunks, there is significant potential for the system getting confused as different subsystems "trample on each others' toes." This means that it is by no means a trivial issue to make sure that a coherent or convergent solution is achieved by Problem-Solving systems.

---

[11]Implementations of both of these languages have been made with *"do this bit in parallel"* constructs e.g. [Gabriel 84] and [Clark 85] and much work is now focusing on the automatic extraction of parallelism in these languages but as yet no symbolic programming equivalent of a vectorizing FORTRAN compiler has been produced. This is because it is generally not known at compile time whether any given expression is worth evaluating in parallel, given the costs of process creation and such-like.

[12]In fact this graph can be of semi-infinite size and often has to be computed on demand; cf. the game tree for a chess game.

[13]If search dominates the computation then massively parallel machines such as the Connection Machine [Hillis 85] may well prove to have the best performance.

This problem is exacerbated by the asynchronous behavior which can happen in MIMD parallel systems. The Poligon system is designed to help the programmer arrive at coherent solutions, whilst still encouraging parallelism at a fine grain.

2.3.4. Programming

Heuristic programs are typically large and their density is great[14]. This means that their code encapsulates a great deal of Knowledge. It is difficult to write such programs for a number of reasons.

- It is difficult to acquire the Knowledge that goes into them, since this is typically not encoded already in a formal algorithmic way.

- It is difficult to represent the Knowledge once it has been acquired. For instance, the programming associated with implementing a statement such as *"Control of center is very important during openings"* would be considerable.

- Good, clean implementations of such systems need to maintain the logical independence of the Knowledge in the system. This is because failure to do so can result in systems that are very brittle when Knowledge is executed in new orders or when new Knowledge is added. The interconnectedness of Knowledge is often difficult to determine when the Knowledge is formulated. Clearly having dependencies between pieces of Knowledge could have a significant impact on the amount of parallelism that could be extracted from such a program and on the program's ability to get the "right" answer.

It is, therefore, a major concern of *AI* programmers that these programs should be easy to implement, debug, modify and maintain.

3. Poligon a System for Parallel Problem-Solving

In this section we describe Poligon. Poligon is an attempt to produce a system which addresses the issues mentioned above to support the development of parallel *AI* systems. It represents, in many ways, an attempt to find an analogue for and implement a parallel form of existing *AI* systems, known as Blackboard Systems [Nii 86].

A brief description of the important aspects of Blackboard Systems will be given, then Poligon itself will be described; structurally, in the way in which it matches its problem domain, and the way in which it is matched to its target hardware.

3.1. Blackboard Systems

Blackboard Systems are instances of a particular computational or problem-solving model - the *"Blackboard"* model or metaphor. This metaphor takes as its source the idea of a collection of experts gathered around a blackboard (see Figure 3-1). Each expert has a specific domain of expertise, which relates to how a part of the problem at hand is to be solved. Each expert looks at the blackboard for representations of the problem which are of interest to his specific area of expertise. Having found such a piece of information he performs whatever operations he finds necessary and posts his conclusions on the blackboard. This new representation of part of the solution might itself be of interest to another expert and so the process continues.

It is clear from this that the sum of the Knowledge in the system must be sufficient to connect all of these areas of expertise. With less Knowledge than this the problem simply will not be soluble. With more Knowledge than this it should be possible to achieve successively

---

[14]This means that the number of executed machine instructions for each line in the source code is typically very large.
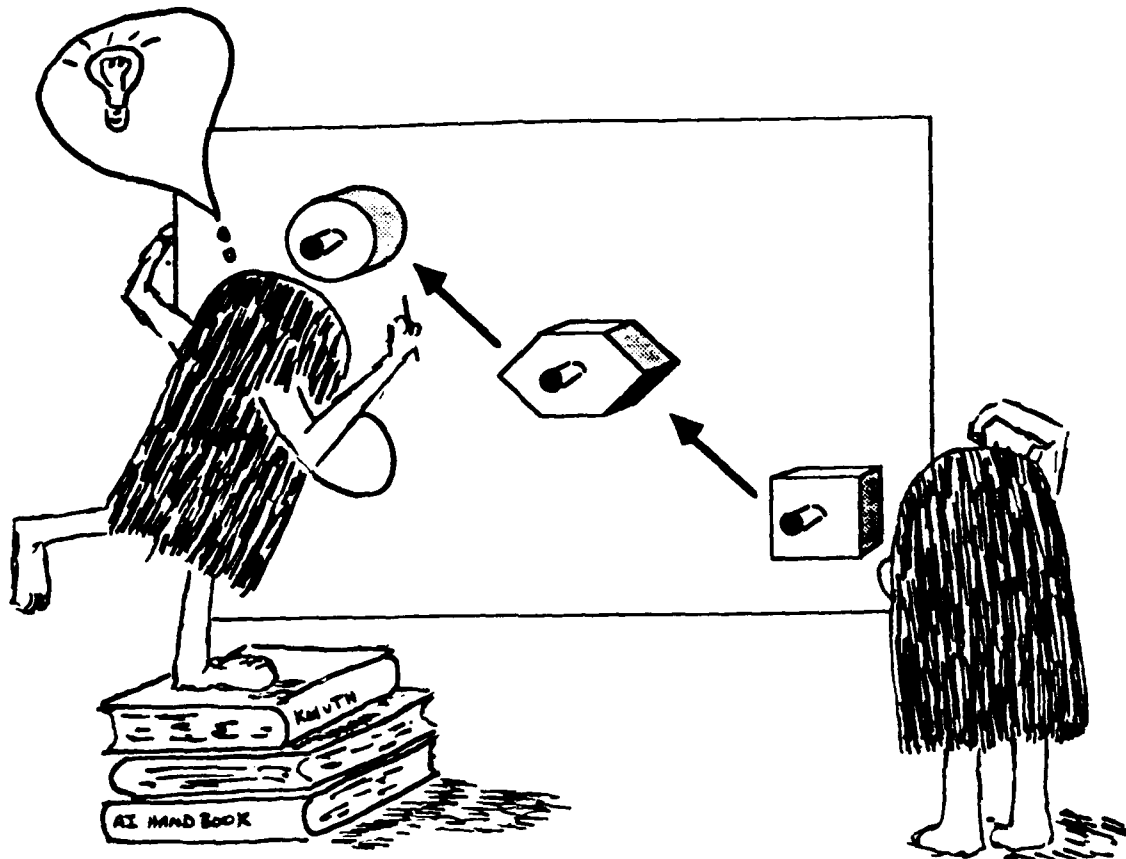
Figure 3-1: The Blackboard Metaphor.
*Eegar, uses encoded Knowledge and comes
to a startling conclusion.*

higher performance from the system; be it faster solutions or better solutions.

This simple model has considerable intellectual appeal and has been the cause of substantial research. It is often claimed that all of these "experts" should be able to operate simultaneously. The Poligon system represents an attempt to test this assertion.

Blackboard systems are typically implemented as large data structures - the *"Blackboard"* - in which are stored the elements of the possible solutions, called *"Nodes"*, which are typically linked together in some way to form a complex graph. There are normally a large number of these Nodes, representing everything from the input data through intermediate solutions to high level abstractions of the current state of the solution. Nodes have internal structure, which allows the mapping of names onto values. They are usually made up of a collection of named *"Slots"* or *"Fields"*, which contain data pertinent to the solution. The Knowledge in the system is usually implemented as a collection of Pattern/Action *"Rules"* collected into groups called *"Knowledge Sources"* (*"KSs"*) [Nii 80]. These reside in an area referred to as the *"Knowledge Base"* (see Figure 3-2).

3.1.1. Consistency and Coherence

Reaching the coherent solution, discussed in §2.3.3, in a Blackboard System is a function of achieving consistency in a number of aspects:
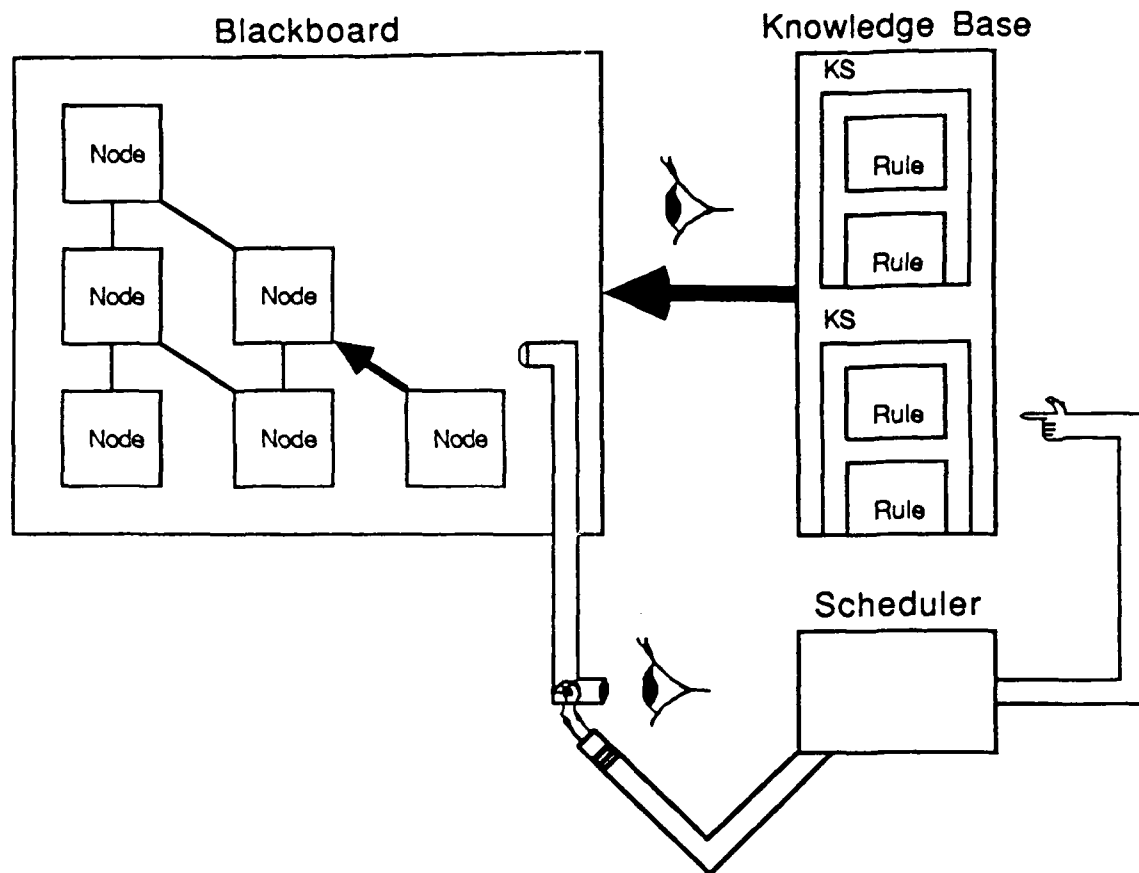
Figure 3-2: A Serial Blackboard System.
*Here, the Scheduler notices a modification event
and invokes a Knowledge Source.*

Node Level   The program should create the right number of Nodes representing the elements in the solution and they should be connected together correctly.

Slot Level   The Slots in the Nodes should contain a respectable representation of the state of that node and its relationship to others.

Rule Execution   When Rules are executed they should do so in an environment which is internally consistent. This means that any information used in the rule during its execution should be based on a consistent snapshot of reality.

3.2. A description of Poligon

Poligon is a framework for the development of Blackboard-like applications on a (simulated) multiprocessor. It consists of:

1. A compiler, which compiles a high-level description of the Blackboard's structure and the Knowledge to be applied by the system, to run on a distributed memory multiprocessor.

2. A run-time system which provides a debugging and testing environment for **Poligon**

programs as well as run-time support.

Both the compiler and the run-time system are thoroughly integrated with the program development environment of TI Lisp machines, the machine on which the execution of **Poligon** programs are simulated.

Serial Blackboard Systems are implemented with the Nodes being represented as records on the Blackboard[15]. The Knowledge is encoded in Knowledge Sources. These are typically compiled into procedures which are invoked by the Blackboard System's kernel. There is some form of scheduler for the Knowledge, which invokes one Knowledge Source after another. The Blackboard and the Knowledge Base both share the same address space, though they are functionally distinct. Knowledge Sources are "*Invoked*" (executed) as a result of changes in the Blackboard placing that change event in a queue used by the scheduler. The scheduler repeatedly picks a Knowledge Source which is interested in the type of event at the end of the queue.
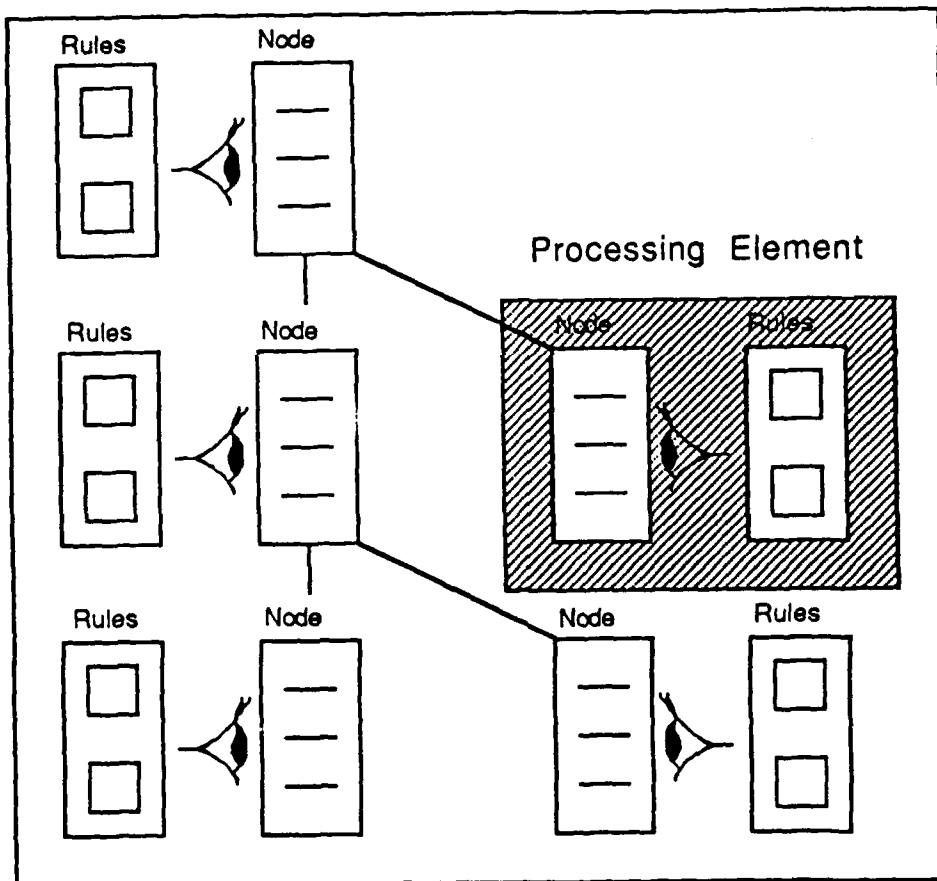
Figure 3-3: Poligon's Blackboard.
*Nodes are seen linked together being watched
by Rules, waiting for modification events.*

The design of ᵢ· ᵢgon has been motivated by the idea of trying to eliminate the bottlenecks that would be experienced if an existing, serial Blackboard System were to be parallelized by

---

[15]These records might well be Pascal-like records or instances of some Class in the native system's object-oriented package.

the inclusion of *"do this bit in parallel"* constructs[16]. The major changes from this model are listed below.

- The scheduling queue of a serial system is eliminated altogether in **Poligon**. This means that concurrent attempts to invoke Rules are not held up waiting for access to this shared data structure.

- Having a Knowledge Base, which is logically distinct from the Blackboard, is no longer necessary since there is now nothing to get between them to control the application of the knowledge. This allows all Knowledge to be attached to those Nodes that are interested in the Knowledge by the compiler (see Figure 3-3).

These changes eliminate at one stroke the bottlenecks of the shared scheduler and the Knowledge Base to Blackboard interface. These changes allowed the development of the idea of the *"Node as a processor"* metaphor for parallel Blackboard systems.

Having eliminated the scheduling mechanism, however, one needs some means of determining when a certain piece of Knowledge should be invoked. It would be hopelessly inefficient to have all of the Knowledge executed all of the time, since most of the time it would find itself inapplicable. It was decided that a simple *daemon-driven* approach would be used to avoid this problem. This results in the Knowledge being directly sensitive to changes in the Blackboard and able to act immediately upon any such changes.

Existing Blackboard Systems often express the Knowledge in their Knowledge Sources as collections of Pattern/Action Rules. These are normally executed serially, in the lexical order in which they are defined. Poligon on the other hand compiles Knowledge Sources away all together, allowing their constituent Rules to be executed in parallel.

The *"Node as a processor"* metaphor is itself a major step away from the normal means of implementing Blackboard Systems. This, however, is not enough. This would give us data parallelism, resulting from the large number of Nodes in the system being able simultaneously to execute Rules, whilst still failing to exploit the potential Knowledge parallelism. This is because each processing element is a uniprocessor, clearly capable of executing at most one Rule at a time[17]. Poligon, therefore, goes beyond this simple model to one which would more accurately be called the *"Rule invocation as a process"* model. This allows the Poligon system to distribute concurrent Rule invocations to different processing elements (see Figure 3-4.

The elimination of serializing components in a Blackboard system also eliminates those mechanisms which are normally used to preserve coherency in the solution. Clearly there is a trade-off which can be made between the amount of control and coherency preserving mechanisms and the amount of exploitable parallelism. **Poligon** is an experiment to explore one extreme of this spectrum. It remains to be seen whether the trade-off made in **Poligon** results in an overall improvement in system performance.

3.3. How Poligon matches the problem domain

Poligon is not a general purpose programming language, other than in the *Turing Complete* sense [Turing 36]. It is specialized to support one computational model and that computational model, itself, has limitations on its sphere of reasonable applicability. It has been designed with applications such as real-time signal understanding and data fusion in mind, though applications outside this domain are being investigated.

The structure of the problem domain is one that requires the representation of a large num-

---

[16]The CAGE system [Aiello 86] is an example of a considerably more conservative approach to the parallelizing of Blackboard Systems.

[17]Each element allows multiple processes but only one is executed at any time.
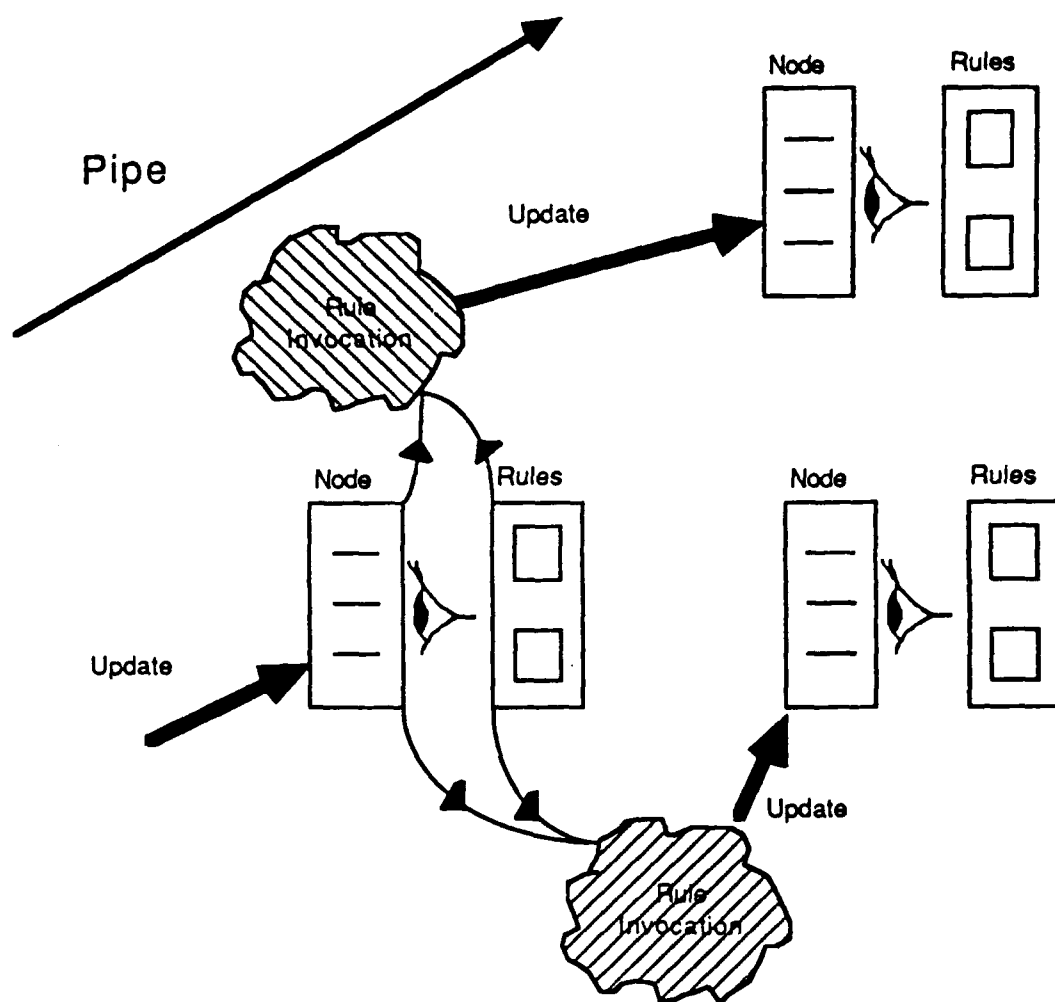
Figure 3-4: Poligon's Execution model.
*An update to a Node triggers concurrent Rule
invocations, which in turn update other Nodes.
Pipes are formed as changes to the Blackboard
flow from one Node to another.*

ber of distinct entities in the solution space. For example the vocabulary of the problem domain is full of such things as aircraft, radar emitting platforms and radar track segments. Poligon provides a rich representation language in which these objects and specializations of them can be expressed. This allows the system to take full advantage of the mutual independence of any of the objects in the solution space to exploit parallelism.

3.4. How Poligon matches its target hardware

Poligon could, of course, run on any machine in principle. In practice, however, it has been designed with a particular kind of machine model in mind and has been optimized to take advantage of it. This class of target machine, which was briefly described in §2.2.2, is exemplified by certain kinds of message-passing, distributed-memory multiprocessors. The grain

size of the executable chunks in **Poligon** programs is designed to suit this model, i.e. each chunk represents, ideally, a few function calls. This makes it coarser grained than those systems that want to execute everything that can be in parallel, for instance data flow machines [Dennis 80], but it is a lot finer grained than most other concurrent Blackboard Systems, such as [Lesser 83] in which each processing element contains a complete Blackboard System.

The target machine model, being of the distributed-memory, message-passing variety including essentially no capability to pass references, strongly discourages shared variables or mutable global data of any sort and encourages a message-passing style of programming. The **Poligon** language is one in which the programmer is given an abstract view of programming using the Blackboard Problem-Solving model. The **Poligon** language has no construct for message sending at all, nor has it any primitives by which the user has access to the underlying architecture or topology. It is assumed to be the duty of the **Poligon** system or the target machine's operating system to look after such concerns. The **Poligon** compiler compiles its programs into the message passing primitives of the underlying system. This allows the efficient use of the underlying architecture, whilst still leaving the source program uncluttered by concrete details of the target architecture.

Poligon allows only global constants, but not variables, since these can be distributed at program load-time.

3.5. What we have learned

Truth comes out of error more easily than out of confusion. - Francis Bacon

Experiments with **Poligon** are by no means complete, but we have learned quite a bit so far. Some of these lessons are enumerated below.

- It is very hard to write any program which implements either a framework, such as **Poligon** or an application such as those which have been mounted on **Poligon**. This is due largely to asynchronous side effects. A system with better formal properties would be less error prone in this respect but might well make less efficient use of the hardware. These difficulties could also be caused by an insufficiency of mechanisms to control coherency in **Poligon** see §3.1.1.

- In order to produce a reliable program it is necessary to write code which makes no assumptions about anything that any other part of the system might be doing. Failure to do so results in brittle systems.

- In order to achieve a coherent solution it was found to be necessary to develop a number of programming methodologies. These will be covered in the same form as they were introduced in §3.1.1.

  Node Level   The creation of Nodes is tricky. Because each element is likely to represent some real-world object, such as an aircraft, it is important either to provide a mechanism for resolving the conflict caused by multiple asynchronous requests to create an element that represents the same thing or to provide a mechanism for managing the creation of Nodes. **Poligon** opts for the latter approach.

  Slot Level   The programmer should cause each Node to have an idea of how to improve its own idea of the solution - to have *Goals*. In **Poligon** this is done at a fine grain, with each field of each element in the solution being able to have associated with it functions which enable it to evaluate itself. This state of affairs has been observed in a different manifestation at a larger grain size in [Corkill 83].

  It was found that a good axiom for programming these systems is *"Never throw away any data unless you are convinced that you*

have better data." This is the sort of behavior that is used in the evaluation functions mentioned above.

Rule Execution   **Poligon** attempts to maintain the smallest critical sections possible. The original implementation of **Poligon** in fact had as its only atomic actions reading a field and writing a field. It was soon found that, in order to maintain consistency during rule execution, it had to be possible to read the values from a number of fields simultaneously - taking a snapshot without the subject moving. This, coupled with critical sections for the writing of collections of values, allows confidence that the picture that one sees when taking such a snapshot of a Node is consistent, even if not necessarily the most up to date. It is important for a **Poligon** programmer to be aware that the Node of which a snapshot has been taken may well be read from and written to by other Rules asynchronously during the invocation of the Rule taking the snapshot.

4. Experiments

In this section we describe, briefly, a series of experiments being performed by the *Advanced Architectures Project* at Stanford University on the **Poligon** system and on CAGE [Aiello 86] and **Lamina** [Delagi 86], other systems developed as part of the same project. However, these experiments will be discussed only in the context of the **Poligon** system.

It would be premature to quote any hard and fast performance figures here, since we still have much to do in order to understand the results that we are getting. The main purpose of reporting these experiments is to show the lessons that have been learned both from performing the experiments and about the ways in which **Poligon** behaves.

4.1. The Problem

Each of the systems mentioned above has been used to implement an application called *"Elint"*, a problem in the domain of real-time interpretation of passive radar signal data [Brown 86].

The problem is one of receiving reports from radar systems, abstracting these into hypothetical radar emitting aircraft and tracking them as they travel through the monitored airspace. These aircraft are themselves abstracted into clusters - perhaps formations - which are themselves tracked. The nature of the radar emissions from the aircraft are interpreted in order to determine the intentions and degree of threat of each of the clusters of emitters.

The Elint application has a number of characteristics which are of significance.

- The system must be able to deal with a continuous data stream. It is not acceptable to wait until all of the data has been read in and then figure out what is going on.

- The application domain is potentially very data parallel. The ability to reason about a large number of aircraft simultaneously is very important. What is more, the aircraft themselves, as objects in the solution space, are quite loosely coupled.

- The application is Knowledge poor. This means that the experiments performed were geared primarily to evaluating the performance of these systems with respect to data parallelism, not knowledge parallelism.

4.2. The Purpose of the Experiments

Napoleon: "*I see no mention of God.*"

Laplace: "*I had no need of that hypothesis.*"

These experiments have five main objectives.

1. To investigate methods of achieving speed-up for expert systems applications by mounting them on parallel hardware architectures[18].

2. To build a number of systems using different computational and problem-solving models and compare their relative performance and thus to deduce an appropriate course for future research. It is therefore imperative that, to the greatest degree possible, each of the systems should implement the same application and should perform the same experiments.

3. To perform experiments on individual systems specialized to investigate characteristics of each computational model, which might not be shown by the experiments mentioned above and which are not shared by other systems.

4. Having done the above, it should be possible to draw some conclusions about the amount of speed-up attainable given these architectures. This should help one to conclude whether these architectures are in fact appropriate and efficient for parallel implementation.

5. The implementation of the Elint system in Poligon was intentionally not tuned. This means that it was a copy of the original serial implementation modified only in so far as it was necessary in order to make it solve problems correctly in parallel. The intent was to achieve a reasonable measure of the performance of an average system that might be written by a Poligon user, as opposed to a very highly tuned version.

4.3. A Description of the Experiments performed on Poligon

Deciding exactly which experiments to perform is difficult, since there are a very large number of variable factors in the system. Amongst these are; the implementation of the Elint system, the characteristics of the data sets used and numerous machine simulation parameters including processor and communications network performance. However, it was decided to freeze most of these and perform a number of experiments, having chosen "reasonable", justifiable values for the frozen parameters. We have, in fact, learned a lot from this process and this has helped us to design a better set of experiments, which are now being performed.

The primary variable factor for these experiments is the data set used to drive the experiment. This data set represents a simulated set of radar observations. These data sets are of finite length. The *length, number of simulated emitters* and *radar observation frequency over time*[19] are the main variable factors in the data sets.

To perform each of these experiments the simulated rate at which data arrived in the system was fixed at a value which was high enough to prevent data starvation when running the experiment on the largest reasonable processor grid. This meant that the speed-up for a grid of size N could be measured simply by dividing the time taken for the grid of size 1 by the time taken by the simulation of the N sized grid.[20]

---

[18]"Expert Systems" are *AI* systems which attempt explicitly to encode the Knowledge of human experts.

[19]Radar system reports per simulated time unit

[20]Performing experiments in this way was intended to give a base-line set of results of the same form as those derived from the CAOS system's implementation of Elint [Schoen 86] and of the Lamina implementation of *Airtrac*, another application [Nakano 87]. For the reasons mentioned in this section this might not be a good base-line for comparison.

It should be noted that these early experiments are open to some criticism as being unrealistic. They represent the speed-up for given programs under some fixed conditions. The conditions that are fixed may not be reasonable. For instance, if the program being run was merely a parallel implementation of *Quicksort* then these would be reasonable experiments. Unfortunately, because the implementations of Elint are intended to be real-time[21] systems it is not realistic to load the system in this way. The problem-solving behavior of the system is sensitive to machine load. Systems running with smaller numbers of processors will be more heavily loaded. They may, therefore, spend a lot of time queue thrashing.

For this reason it is now known that these experimental results should not be taken at face value. More satisfactory experiments have been devised, in which the experiment is run for a given number of processors with the data rate being varied until the latency of the output traces is constant over time. This means that the maximum sustainable data rate without increasing latency in the system's outputs is the preferred measure of the speed-up for these systems.

4.3.1. Experiment 1

The Fusion Plasma requires a temperature of 500 million degrees, but I forget whether that's Centigrade or Absolute. - Overheard by Arthur H. Snell, Oak Ridge National Laboratory.

This experiment was intended to be a simple cross comparison experiment, performed by all of the systems. Its data set was a simple, and quite small one, which contained observations of sufficient variety to exercise all of the system's required behavior.

The speed-up figures produced showed a peak speed-up for the system of about 4.5X for sixty-four processors, with the speed-up trailing off quite sharply. This was disappointing.

One of the problems with this experiment was that the data set was varied in the frequency of input data for the system over time. It was sparse at the beginning, heavy in the middle and sparse at the end. This resulted in the system being data starved near the beginning of the simulation and then flooded in the middle.

Although such spikes in input data are entirely characteristic of real data, this extra variable factor was thought to be too difficult to factor out, in order to arrive at a realistic speed-up figure. If the system is lightly loaded then not much speed-up is needed. For this reason all subsequent experiments have been and will be performed on data sets that have a constant frequency of input data.

The most important thing to conclude from this result is that we had much to learn about how to conduct these experiments.

4.3.2. Experiment 2

This experiment was designed to compensate for the variability found in the data set used in Experiment 1. The data set had a constant frequency for input data over time.

This experiment showed that the peak speed-up had increased to about 7X, which was reached after sixteen processors. This result was somewhat better than that from Experiment 1, supporting our hypothesis that the shape of the input data was affecting our results. Analysis of the instrumentation indicated that the limiting factor in the parallelism detected was probably a bottleneck on a particular Node representing a cluster of emitters. It also showed that even if all bottlenecks were eliminated, so that all pipes were balanced, a major limiting factor in the performance of the system was that there wasn't enough parallelism at this grain size available in the data set for this system to exploit.

---

[21]"Real-Time" is used here in the sense that the system must cope with an unbounded continuous stream of data, whilst delivering results reasonably promptly. It is not intended to refer to those real-time systems where guaranteed response times might be required.

4.3.3. Experiment 3

This experiment was intended to determine how efficiently the simulated hardware architecture was being used and thus show where effort would best be expended to speed up the system if the application could not be changed structurally. To achieve this Experiment 2 (see §4.3.2) was repeated a number of times but for each iteration the simulated speed of the processor was varied. This gave speed-up figures for processor performances which were 2, 4 and 8 times the speed of the processor simulated in Experiment 2[22]. All of the speed-up figures produced were then normalized against the case of Experiment 2. A significant reduction in the speed-up of the system would have indicated that the increasing performance of the processor was swamping the communication hardware, thus indicating that time and effort would better be spent on improving communication performance.

It was found that the normalized speed-ups matched each other very closely. This is taken to indicate that, if such a machine were to be implemented for Poligon programs, effort spent on improving the processor's performance or in optimizing the program would probably be rewarded by close to linear speed-up.

4.3.4. Discussion of Experiments: What we have learned.

Experience is the name everyone gives to their mistakes. - Oscar Wilde, *"Lady Windermere's Fan"*

As has already been mentioned the experiments on these systems are in their infancy. It is essential for the reader to note, therefore, that these results should be taken as nothing more than indication of where our research is leading us, rather than hard and fast statements about the performance of these systems.

We have, however, learned quite a bit in the execution of these experiments. The more important of these lessons are listed below.

- Getting useful speed-up out of these systems, at least given the current level of our understanding and methodologies, is very difficult. The speed-ups shown for the experiments mentioned in this section may, indeed, have been achievable by very careful coding on a uniprocessor. These difficulties are characterized mainly by the difficulty of implementing the program and debugging it and of combating serial components in the processing.

- Problem-Solving systems such as the ones mentioned in this paper are significantly more complex than those programs normally implemented to evaluate experimental parallel hardware. Our difficulty in getting results indicates that there is more to getting useful speed-up for real problems than there is to demonstrating speed-up for Quicksort programs such as [Deminet 82].

- The domain of Real-time systems is one in which the *AI* community in general and this project in particular has little experience. This has made implementation of these systems and the analysis of them difficult. The selection of a different field for research, outside that of real-time systems, would have alleviated this problem but would have removed the area of experimentation from an important area of application where it is believed that speed-up through parallelism is both necessary and feasible.

- Real-time systems present a set of problems for performance evaluation so great that it is difficult to formulate easily analyzable experiments and draw worthwhile conclusions from them. These problems are caused by; the need for continuous data, end effects when the data is bounded in extent, the difficulty of defining suitable performance measures and Heisenbergian effects i.e. changes in system load during speed-up measurement changing the speed-up itself.

- Investigation of the amount of "Knowledge Parallelism" has been limited by the

---

[22]For each of these experiments the simulated input data rate was also increased so as to factor out this change.

relatively small amount of Knowledge available in this area. New applications are being sought in which more Knowledge is available. This has concentrated the investigation on the extraction of data parallelism from these systems.

- The data sets for the experiments mentioned above are limited in the amount of data parallelism that can be extracted from them. To add to this problem the **Poligon** system is sufficiently difficult to simulate that experiments with significantly larger data sets are probably not feasible.

- The immediate conclusion that one is led to by these results is that a relatively simplistic implementation of a system can lead to speed-ups of the order of 10X. It seems to be possible to get higher speed-ups from such systems but, at least at present, only by very careful coding and very careful and thorough instrumentation of the running system so that bottlenecks can be eliminated.

- So far, it has not been possible to demonstrate overall speed-ups of more than ~8X using **Poligon**. The hypothesis that **Poligon's** implementation of Elint will be able to exploit data parallelism as larger data sets are used remains, as yet, untested, though tentative results from an implementation of Elint in **Lamina** (~23X) and Airtrac in **Lamina** [Nakano 87] (~80X) give cause for hope, indicating that with larger data sets there definitely is more parallelism to extract.

5. Conclusions

There is something fascinating about science. One gets such wholesale returns of conjecture out of such a trifling investment of fact. - Mark Twain, *"Life on the Mississippi"*

This paper has introduced the problems associated with attempts to achieve speed-up though parallelism for *Problem-Solving* systems, systems developed in the *Artificial Intelligence* field. Numerous applications for such systems would benefit greatly from being sped-up considerably. Because of their irregular structure, such systems are shown to be difficult to speed up through well established means.

The **Poligon** [Rice 86] system was described. **Poligon** is an attempt to create a system which is able to encourage the decomposition of a particular class of Problem-Solving systems, known as *Blackboard Systems*, into a form, which can be efficiently executed by it on a distributed-memory, message-passing multiprocessor.

The **Poligon** system has been implemented and an application called *"Elint"* has been implemented using it. Lessons learned in the implementation of **Poligon** and the Elint application are detailed.

Experiments are now being performed on the Elint application, both for the implementation mentioned in **Poligon** and also for systems called **Lamina** [Delagi 86] and CAGE [Aiello 86]. Some preliminary experimental results are shown. Lessons learned from these experiments are described. Some of these are as mentioned below.

- It is very difficult to implement both frameworks for concurrent Problem-Sciving and concurrent Problem-Solving systems themselves. This is due largely to the difficulty of coping with asynchronous events, caused largely by these systems being MIMD systems.

- Real-time systems are difficult systems to calibrate for the purposes of experimentation to evaluate speed-up.

- Modest speed-up has been achieved (~8X). Indications of higher performance (~23X-80X) are thought possible through the exploitation more data parallelism [Nakano 87] .

- The potential for the exploitation of Knowledge Parallelism has not yet been investigated.

- If these results are supported by further work they would indicate that large amounts of parallelism at this grain size might not be easily achieved for this type of *AI* system. Thus, if there is not a lot of Knowledge to apply, if there is not a lot of data parallelism available and if there are not many alternatives to explore in the application it may be that a software architecture optimized for a distributed-memory hardware architecture is not appropriate. This does not mean, however, that implementation techniques such as data copying and a message passing metaphor often used in distributed memory systems are not appropriate for a shared memory implementation, since they can help to avoid bottlenecks.

Report writing, like motor-car driving and love-making, is one of those activities which every Englishman thinks he can do well without instruction. The results are of course usually abominable. - Tom Margerison, reviewing *Writing Technical Reports* by Bruce M. Cooper in the *Sunday Times, 3 January 1965*

References

[Aiello 86] Nelleke Aiello.
User-Directed Control of Parallelism; The CAGE System.
Technical Report KSL-86-31, Heuristic Programming Project, C. S. Dept.,
 Stanford University, 1986.

[Brown 86] Harold Brown, Eric Schoen, Bruce A. Delagi.
An Experiment in Knowledge-Base Signal Understanding Using Parallel Ar-
 chitectures.
Technical Report STAN-CS-86-1136, Heuristic Programming Project, C. S.
 Dept., Stanford University, 1986.

[Byrd 87] Gregory T. Byrd, Bruce A. Delagi.
Considerations for Multiprocessor Topologies.
Technical Report KSL-87-07, Heuristic Programming Project, C. S. Dept.,
 Stanford University, 1987.

[Clark 85] Clark, K and Gregory, S.
PARLOG: Parallel Programming in Logic.
Technical Report Research Report DOC 84/4, Dept. of Computing, Imperial
 College of Science and Technology, 1985.

[Clocksin 81] Clocksin, W. F. and Mellish, C. S.
Programming in PROLOG. ·
Springer-Verlag, Berlin, 1981.

[Corkill 83] Daniel D. Corkill and Victor R. Lesser.
The use of Meta-Level Control for Coordination in a Distributed Problem
 Solving Network.
Proc. of IJCAI 8 , 1983.

[Delagi 86] Bruce A. Delagi, Nakul P. Saraiya, Gregory T. Byrd.
LAMINA: CARE Applications Interface.
Technical Report KSL-86-67, Heuristic Programming Project, C. S. Dept.,
 Stanford University, 1986.

[Deminet 82] Jarek Deminet.
Experience with Multiprocessor Algorithms.
IEEE Trans. on Computers C-31(4):278 - 287, April, 1982.

[Dennis 80] Jack B. Dennis.
Data Flow Supercomputers.
IEEE Computer :48 - 56, November, 1980.

[Flynn 72] M. Flynn.
Some Computer Organizations and their Effectiveness.
IEEE Trans on Computers, C-21 :948 - 960, 1972.

[Gabriel 84] Gabriel, Richard P. and McCarthy, John.
Queue-based Multi-processing Lisp.
Proceedings of the ACM Symposium on Lisp and Functional programming :25
 - 44, August, 1984.

[Gupta 86]   Anoop Gupta.
                Parallelism in Production Systems.
                PhD thesis, Department of Computer Science, Carnegie-Mellon University,
                    March, 1986.

[Hillis 85]   W. Daniel Hillis.
                The Connection Machine.
                MIT Press, Cambridge, Massachusetts, 1985.

[Kuck 81]   D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure and M. Wolfe.
                Dependence Graph and Compiler Optimizations.
                Proceedings of the 8th ACM Symposium on Principles of Programming Lan-
                    guages , January, 1981.

[Kung 78]   H. T. Kung and C. E. Leiserson.
                Systolic Arrays (for VLSI).
                In I. S. Duff and G. W. Stewart (editor), *Sparse Matrix Proceedings*, pages
                    256 - 282. Society for Industrial and Applied Mathematics, 1978.

[Lee 85]   Gyungho Lee, Clyde P. Kruskal and David J. Kuck.
                The Effectiveness of Automatic Restructuring on Nonnumerical Programs.
                IEEE Trans. on Computers , 1985.

[Lesser 83]   Victor R. Lesser and Daniel D. Corkill.
                The Distributed Vehicle Monitoring Testbed: A Tool for Investigation Dis-
                    tributed Problem Solving Networks.
                The AI Magazine Fall:15 - 33, 1983.

[Nakano 87]   Russel T. Nakano, Masafumi Minami.
                Experiments with a Knowledge-Based System on a Multiprocessor.
                Technical Report KSL-87-61, Heuristic Programming Project, C. S. Dept.,
                    Stanford University, 1987.

[Nii 80]   H. Penny Nii.
                An Introduction to Knowledge Engineering, Blackboard Model, and AGE.
                Technical Report HPP-80-20, Heuristic Programming Project, C. S. Dept.,
                    Stanford University, March, 1980.

[Nii 86]   H. Penny Nii.
                Blackboard Systems.
                AI Magazine 7:2, 1986.

[Petard 38]   H Petard.
                A contribution to the mathematical theory of big game hunting.
                American Mathematical Monthly (45):446, 1938.

[Rice 86]   J. P. Rice.
                The Poligon User's Manual.
                Technical Report KSL-86-10, Heuristic Programming Project, C. S. Dept.,
                    Stanford University, 1986.

[Schoen 86]   Schoen, Eric.
                The CAOS System.
                Technical Report STAN-CS-86-1125, Heuristic Programming Project, C. S.
                    Dept., Stanford University, 1986.

[Seifert 34]   H. Seifert and W. Threlfall.
                Lehrbuch der Topologie.

[Seitz 85]   Charles L Seitz.
                The Cosmic Cube.
                Communications of the ACM 28:22 - 33, 1985.

[Steele 84]   Guy L. Steele Jr.
                Common Lisp.
                Digital Press, 1984.

[Turing 36]   Alan M. Turing.
                On Computable Numbers, with an Application to the Entscheidungsproblem.
                Proc. of London Mathematical Society 2(42 and 43):230 - 265 and 544-546,
                    1936.

[Wilson 87]   Andrew W. Wilson Jr.
                Hierarchical Cache/Bus Architecture for Shared Memory Multiprocessors.
                Proc. of 14th Symposium on Computer Architectures :244 - 252, 1987.

Appendix I

Frameworks for Concurrent Problem Solving: A Report on Cage and Poligon

by

H. Penny Nii, Nelleke Aiello, and James Rice

Knowledge Systems Laboratory
Computer Science Department
Stanford University
Stanford, California 94305

Table of Contents

Frameworks for Concurrent Problem Solving:
A Report on Cage and Poligon

Abstract

This paper describes the ways in which blackboard systems can be made to operate in a multi-processor environment. Cage and Poligon, two concurrent problem solving systems based on the blackboard model are described. The factors which motivate and constrain the design of parallel systems in general and parallel problem-solving systems in particular are described.

1. Background

A *Concurrent Problem Solving System* is a network of autonomous, or semi-autonomous, computing agents that solve a single problem. In building concurrent problem solvers, our objectives are twofold: (1) to evolve or invent models of problem solving in a multi-agent environment *and* (2) to gain significant performance improvement by the use of multi-processor machines. Within the community of researchers in artificial intelligence, there is an interest in understanding and building programs that exhibit cooperative problem-solving behavior among many intelligent agents, independent of computational costs (see [Corkill 83, Lesser 83, Smith 81] for some examples). *But*, one of the important pragmatics of using many computers in parallel is to gain computational speed-up[1]. Often, methods useful in a serial (single) problem solver in obtaining a valid solution and coherent problem-solving behavior, usually a centralized control, are not compatible with performance gain in a multi-agent environment. Cage and Poligon attempt to find a balance -- to achieve adequate coherence with minimal global control *and* to gain performance with the use of multiple processors.

1.1. Problem Solving and Concurrency

Those problems that have been successfully solved in parallel, such as partial differential equations and finite element analysis, share common characteristics: they frequently used vectors and arrays; solutions to the problems are very regular, using well understood algorithms; and the computational demands, for example, for matrix inversion, are relatively easy to compute. In contrast, the class of applications we are addressing (and AI problems in general) are ill-structured or ill-defined. There is often more than one possible solution; paths to a

---

[1]Multiple computers are also used for other reasons besides speed-up -- redundancy, mix of specialized hardware, need for physical separation, and so on.

solution cannot be predefined and must be dynamically generated and tried; generally data cannot be encoded in a regular manner as in arrays -- the data structures are often graph structures that must be dynamically created, precluding static allocation and optimization. These differences indicate that to run problem solving programs in parallel, current techniques for parallel programs must be augmented or new ones invented. It is worth reviewing some of the key points to be addressed in building concurrent, problem-solving programs.

1.1.1. Problem Solving Issues

Problem solving has traditionally meant a process of searching a tree of alternative solutions to a problem. Within each generate-and-test cycle, alternatives are generated at a node of a tree and promising alternatives selected for further processing. Knowledge is used to prune the tree of alternatives or to select promising paths through the tree. It is an axiom that the more knowledge there is the less generation and testing has to be done. In the extreme, many knowledge-based systems have large knowledge bases containing pieces of knowledge that recognize intermediate solutions and solution paths, thereby drastically reducing, or even eliminating, search. These two types of problem-solving techniques have been labeled *search* and *recognition* [McDermott 83]. In the search technique the majority of computing time is taken up in generating and testing alternative solutions; in the recognition technique the time is taken up in *matching*, a process of finding the right piece of knowledge to apply. Most applications use a combination of search and recognition techniques. A concurrent problem solving framework must be able to accommodate both styles of problem solving.

In serial systems meta-knowledge, or control knowledge, is often used to reduce computational costs. One common approach decomposes a problem into hierarchically organized sub-problems, and a control module selects an efficient order in which to solve these sub-problems. Closely related is the introduction of contextual information, or domain knowledge, to help in the recognition process. Both approaches enhance performance -- reduce the number of alternatives to search or the amount of knowledge to match. In concurrent systems meta-knowledge and control modules become fan-in points, or hot-spots. A *hot-spot* is a physical location in the hardware where a shared resource is competed for, forcing an unintended serialization. Does this imply that problem solving systems that rely heavily on centralized control are doomed to failure in a concurrent environment? Can control be distributed? If so, to what extent? If more knowledge results in less search, can a similar trade-off be made between knowledge and control? In concurrent systems where control, especially global control, is a serializing process, can knowledge be brought to bear to alleviate the need for control?

1.1.2. Concurrency Issues

The biggest problem in concurrent processing was first described by Amdahl [Amdahl 67]. Simply stated, it is as follows: The length of time it takes to complete a run with parallel processes is the length of time it takes to run the longest process plus some overhead associated with running things in parallel. Take a problem that can be decomposed into a collection of independent sub-problems that can run concurrently, but which internally must run serially. If all of these components are run concurrently, then the run-time for the whole problem will be equal to the run-time for the longest running component, plus any overhead needed to execute the sub-problems in parallel. Thus, if the longest process takes 10% of the total run time that the parallel processes would have taken if run end-to-end (serially), then the maximum speed-up possible is a factor of 10. Even if only one percent of the processing must be done sequentially this limits the maximum speed-up to one hundred, however hard one tries and however many processors are used. This is a very depressing result, since it means that many orders of magnitude of speed-up are only available in very special circumstances.

This raises the issue of *granularity*, the size of the components to be run in parallel. Amdahl's argument indicates the need for as small a granularity as possible. For example, is a rule a good candidate grain size for computation? On the other hand, if the process creation and process switching time is expensive, we want to do as much computation as possible once a process is running, that is, favor a larger granularity. In addition, in a multi-computer architecture a balance must be achieved between the load on the communication network and on the processors. It is often the case that as process granularity decreases, the processes become more tightly coupled -- that is, there is a need for more communication between them. The communication cost is of course a function of the hardware-level architecture, including bandwidth, distance, topology, and so on. Finding an optimal grain size at the problem solving level is a multi-faceted problem.

Even if one is able to find an optimal granularity, there are forces that inhibit the processes from running arbitrarily fast in parallel. Some of the more common problems are:

- *Hot-Spots* and *Bottlenecks:* It is frequently the case that a piece of data must be shared. In any real machine multiple, simultaneous requests to access the same piece of data cause *memory contention*. The act of a number of processes competing for a shared resource -- memory or processors -- causes a degradation in performance. These processor and memory hot-spots cause bottlenecks in the processing of data; they restrict the flow of data and reduce parallelism.

- *Communications*: Multi-computer machines do not have a shared address space in which to have memory bottlenecks of the kind mentioned above. However the communications network over which the processing elements communicate still represents a shared resource which can be overloaded. It has a finite bandwidth. Similarly, multiple, asynchronous messages to a single processing element will cause

that element to become a hot-spot.

- *Process Creation*: Execution of the sub-problems mentioned above require that they run as processes. The cost of the creation and management of such processes is non-trivial. There is a process grain size at which it does not pay to run in parallel, because executing it sequentially is faster than executing it in parallel.

Having introduced some issues and constraints associated with parallelizing programs, we now introduce some other concepts that are important in writing concurrent programs, an understanding of which is useful to appreciate the discussions later in this paper fully.

- *Atomic operation*: This refers to a piece of code which is executed without interruption. In order to have consistent results (data) it is important to have well defined atomic operations. For instance, an update to a slot in a node might be defined to be atomic. Primitive atomic actions are usually defined at the system level.

- *Critical sections*: Critical sections are usually programmer-defined and refer to those parts of the program which are uninterruptible, that is, atomic. The term is usually used to describe large, complex operations that must be performed without interruption.

- *Synchronization*: This term is used to describe that event which brings asynchronous, parallel processes together synchronously. Synchronization primitives are used to enforce serialization.

- *Locks*: Locks are mechanisms for the implementation of critical sections. Under some computational models, a process that executes a critical section must acquire a lock. If another process has the lock, then it is required to wait until that lock is released.

- *Pipeline*: A pipeline is a series of distinct operations which can be executed in parallel but which are sequentially dependent; for instance, an automobile assembly line. The speed-up that can be gained from a pipeline is proportional to the number of stages, assuming that each stage takes the same amount of time, that is, if the pipe is "well balanced." Pipeline parallelism is a very important source of parallelism.

1.2. Background Motivation

In experiments conducted at CMU [Gupta 86], Gupta showed that applications written in OPS [Forgy 77] achieved speed-up in the range of eight to ten, the best case being about a factor of twenty. The experiments ran rules in parallel, with pipelining between the condition evaluation, conflict resolution, and action execution. The overhead for rule matching was reduced with the use of a parallelized Rete algorithm. (In programs written in OPS, roughly 90% of the time is spent in the match phase.) The speed-up factors seem to reflect the

amount of relevant knowledge chunks (rules) available for processing a given problem solving state; this number appears to be rather small. Although the applications were not written specifically for a parallel architecture, the results are closely tied to the nature of the OPS system itself, which uses a monolithic and homogeneous rule set and an unstructured working memory to represent problem solving states.

The premise underlying the design of Cage and Poligon is that this discouraging result could be overcome by dividing and conquering. It is hoped that by partitioning an application into loosely-coupled sub-problems (thus partitioning the rule set into many subsets of rules), and by keeping multiple states (for the different sub-problems), multiplicative speed-up, with respect to Gupta's experimental results, can be achieved. If, for example, a factor of seven speed-up could be achieved for each sub-problem, the simultaneous execution of rule sets could result in a speed-up of seven times the number of sub-problems. We are looking for methods that can provide at least a two orders-of-magnitude speed up. The challenge, of course, is to coordinate the resulting asynchronous, concurrent, problem solving processes toward a meaningful solution with minimal overheads.

1.3. The Blackboard Model and Concurrency

The foundation for most knowledge-based systems is the problem-solving framework in which an application is formulated. The problem-solving framework implements a computational model of problem solving and provides a language in which an application problem can be expressed. We begin with the blackboard model of problem solving [Nii 86], which is a problem-solving framework for partitioning problems into many loosely coupled sub-problems. Both Cage and Poligon have their roots in the blackboard model of problem solving. The blackboard framework seems, at first glance, to admit the natural exploitation of concurrency. Some of the possible parallelism that can be exploited are:

- knowledge parallelism -- the knowledge sources and rules within each knowledge source can run concurrently;

- pipeline parallelism -- transfer of information from one level to another allows pipelining; and

- data parallelism -- the blackboard can be partitioned into solution components that can be operated on concurrently.

In addition, the dynamic and flexible control structure can be extended to control parallelism.

These characteristics of blackboard systems have prompted investigators, for example Lesser and Corkill [Lesser 83] and Ensor and Gabbe [Ensor 85], to build distributed and/or parallel

blackboard systems. The study of parallelism in blackboard systems goes back to Hearsay-II [Fennell 77].

The blackboard problem-solving metaphor itself is very simple; it entails a collection of intelligent agents gathered around a blackboard, looking at pieces of information written on it, thinking about them and writing their conclusions up as they come to them. This is shown in Figure 1-1.
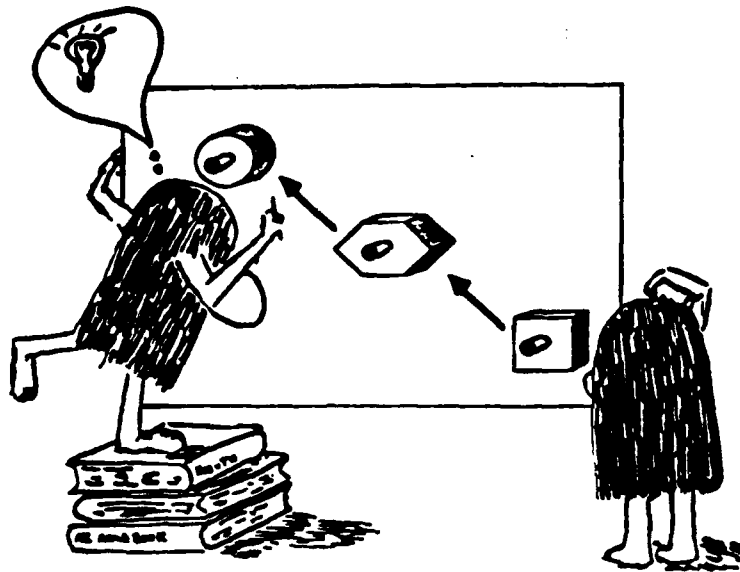
Figure 1-1:   The Blackboard Metaphor

There are some assumptions made in this model that are so obvious that they might be missed. An understanding of the implications of these assumptions is vital to an understanding of the problem of achieving parallelism in blackboard systems.

- All of the agents can see all of the blackboard all of the time, and what they see represents the current state of the solution.

- Any agent can write his conclusions on the blackboard at any time, without getting in anyone else's way.

- The act of an agent writing on the blackboard will not confuse any of the other agents as they work.

The implications of these assumptions are that a single problem is being solved asynchronously and in parallel. However, the problem solving behavior, if it were to be emulated in a computer, would result in very inefficient computation. For example, for every agent to "see" everything would entail stopping everything until every agent has looked at everything.

Existing, serial blackboard systems make a number of modifications to the pure blackboard metaphor in order to make a reasonable implementation on conventional hardware. In effect, they modify the blackboard metaphor so that it *cannot* be executed in parallel. Some of these modifications are shown in Figure 1-2 and are described below.
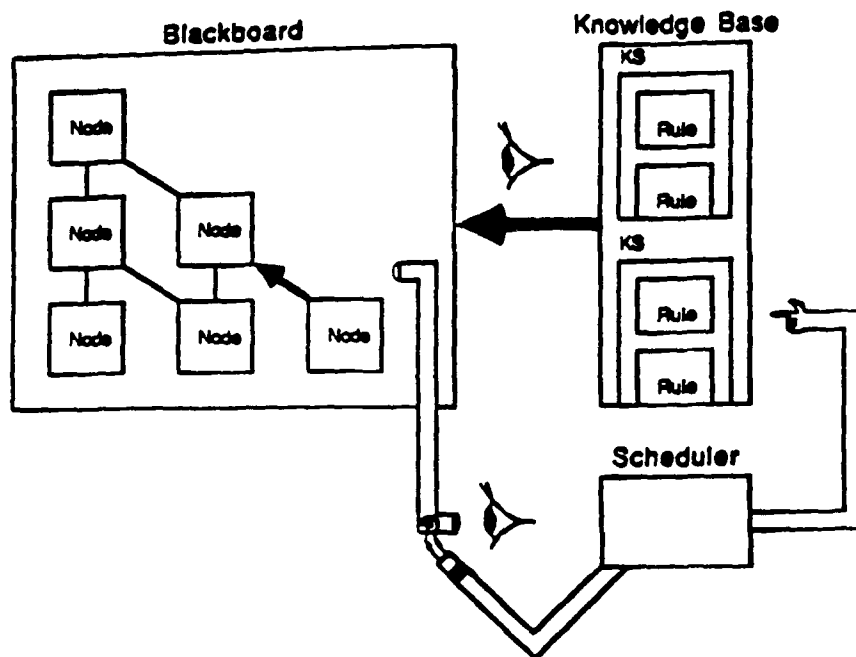
Figure 1-2:   The Serial Blackboard Model

- Agents are represented as *knowledge sources*. These knowledge sources are schedulable entities and only one can be running at any time. It will be shown later that one of the possible sizes for computational grains is the knowledge source.

- To coordinate the execution of knowledge sources, a scheduling or control mechanism is implemented. This is, in many ways, an efficiency gaining mechanism, which uses control knowledge to select only the most "valuable" knowledge source at any given moment to work on the problem.

- The blackboard is not truly "globally visible" in the sense prescribed by the blackboard metaphor. Instead, the blackboard is implemented as a data structure, which is sufficiently interconnected that it is possible for a knowledge source to find its way from one data item to a related one easily. Knowledge sources can only work on a limited area of the blackboard -- knowledge sources and their context of invocation are, in fact, treated as self-contained subproblems.

- An imp'icit assumption is made that a knowledge source operates within a valid, or consistent, context and that the "ordered" execution of knowledge sources, even when the ordering is done dynamically, preserve the consistency of the blackboard data.

Trying to directly parallelize serial blackboard systems characterized above have certain

limitations. First, only a modest speed-up can be achieved by a central scheduler determining the knowledge sources to be run in parallel. The performance levels off very quickly at a very low number (a gain of less than a factor of three in our experiments) no matter how many knowledge sources are run in parallel and no matter how many processors are used. Second, one of the most difficult problems in parallel computation is to maintain consistent data values. In concurrent blackboard systems, the data consistency problems occur in three different contexts: (1) on the entire blackboard, maintaining consistent solution states; (2) in the contents of the nodes, assuring that all slot values are from the same problem solving state; and (3) in the slots, keeping the value being evaluated from changing before the evaluation is completed.

2. The Advanced Architectures Project

Cage [Aiello 86] and Poligon [Rice 86], two frameworks for concurrent problem solving, are being developed within the Advanced Architectures Project (AAP) at the Knowledge Systems Laboratory of Stanford University. The objective of the AAP is the development of broad system architectures that exploit parallelism at different levels of a system's hierarchical construction. To exploit concurrency one must begin by looking for parallelism at the application level and be able to formulate, express, and utilize that parallelism within a problem-solving framework, which, in turn, must be supported by an appropriate language and software/hardware system. The system levels chosen and some issues for study are:

- Application level: How can concurrency be recognized and exploited?

- Problem solving level: Is there a need for a new problem-solving metaphor to deal with concurrency? What is the best process and data granularity? What is the trade-off between knowledge and control?

- Programming language level: What is the best process and data granularity at this level? What are the implications of choices at the language level for the hardware and system architecture?

- System/hardware level: Should the address spaces be common or disjoint? What should the processor and memory characteristics and granularity be? What is the best communication topology and mechanisms? What should the memory-processor organization be?

At each system level one or more specific methods and approaches have been implemented in an attempt to address the problems at that level. These programs are then vertically integrated to form a family of experimental systems -- an application is implemented using a problem-solving framework using a particular knowledge representation and retrieval method, all of which use a specific programming language, which in turn runs on a specific system/hardware architecture simulated in detail on the Lisp-based CARE simulator [Delagi 86a]. Each family

of experiments is designed to evaluate, for example, the system's performance with respect to the number of processors, the effects of different computational granularity on the quality of solution and on execution speed-up, ease of programming, and so on. The results of one such family of experiments have been reported by Brown and Schoen [Brown 86, Schoen 86].

Within the context of this AAP organization, Cage and Poligon are two systems that are implemented to study the problem-solving level. Both Cage and Poligon use frames and condition-action rules to represent knowledge. The target system architecture for Cage is a shared-memory multi-processor; the target architecture for Poligon is a distributed-memory multi-processor, or multi-computer.

Both Cage and Poligon aim to solve a particular, but broad, class of applications: real-time interpretation of continuous streams of errorful data, using many diverse sources of knowledge. Each source of knowledge contributes pieces of a solution which are integrated into a meaningful description of the situation. Applications in this class include a variety of signal understanding, information fusion, and situation assessment problems. The utility of blackboard formulations has been successfully demonstrated by programs written to solve problems in our target application class [Brown 82, Mccune 83, Nii 82, Shafer 86, Spain 83, Williams 84].

Most of the systems in this class use the recognition style of problem solving with knowledge bases of facts and heuristics; numerical algorithms are also included as a part of the knowledge. Some search methods are employed but are generally confined to a few of the sub-problems.

In designing a concurrent blackboard system for the AAP, two distinct approaches seemed possible -- one, to extend a serial blackboard system, and the other, to devise a new architecture to exploit the event-driven nature of blackboard systems. Each has its own problems and its own advantages, which will be described in the following sections.

3. Extending the Serial System - Cage

Cage is a concurrent blackboard framework system, based on the (serial) AGE [Nii 79] blackboard system. AGE uses a set of rules as a representation for its knowledge sources; it uses a set of event tokens as preconditions (a trigger) for the knowledge sources, and each significant change to the blackboard posts an event in a global data structure. The controller selects an event and executes a knowledge source whose precondition matches the selected event[2]. In addition to the basic functionality found in AGE, Cage allows user-directed control

---

[2]There are more elaborate constructs in AGE, but this description suffices for the current purpose.

over the concurrent execution of many of its constructs (see Figure 3-1). Otherwise, the two systems are functionally identical.
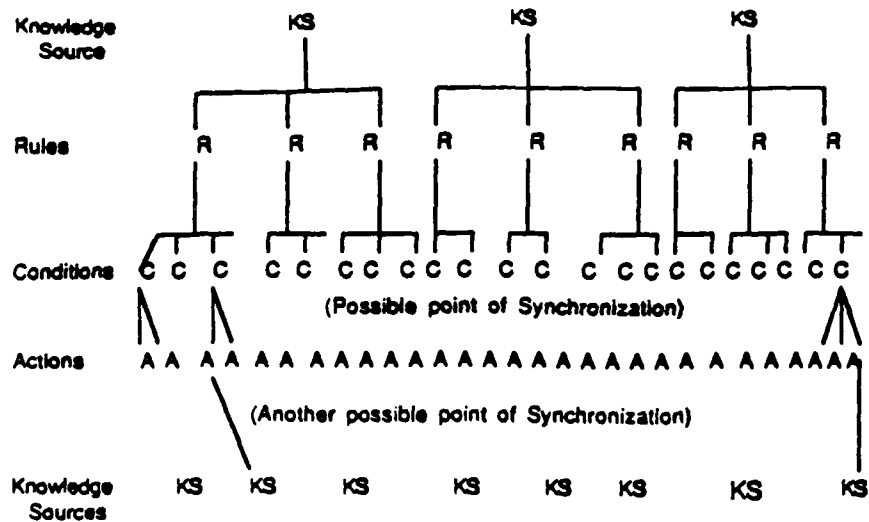
Figure 3-1: Parallel Components of CAGE

3.1. The Cage Architecture

The basic components of a system built with Cage are:

- A global data store (the blackboard) on which emerging solutions are posted. Objects on the blackboard are organized into hierarchical levels, and each object is described with a set of attribute-value pairs.

- Globally accessible lists on which control information is posted (for example, lists of events, expectations, and so on).

- An arbitrary number of knowledge sources, each consisting of an arbitrary number of rules.

- Control information that can help to determine (1) which blackboard elements are to be the focus of attention and (2) which knowledge sources are to be used at any given point in the problem solving process.

- Declarations that specify which components are to be executed in parallel (knowledge sources, rules, condition and action parts of rules), and at what points synchronization is to occur.

The user can run Cage serially (at which point Cage behavior is identical to that of AGE), or can run with one or more of the components running concurrently. In the serial mode, the basic control cycle begins with the selection and execution of a knowledge source. A resulting change to the blackboard may cause several knowledge sources to become relevant and

candidates for execution. Cage uses a global list structure to record the changes to the blackboard, called events. The controller selects one of the events. The user can specify how the event is to be selected, such as FIFO, LIFO, or any user defined best-first method. The event in focus is then matched against the knowledge source preconditions. The knowledge sources, whose preconditions match the focus events, are then executed in some predetermined order. The rules within each knowledge source are evaluated, and the action part of the rule is executed for those rules whose condition parts are satisfied. The user may choose to allow only one rule to fire per knowledge source activation or many rules to fire. Each action part may cause one or more changes on the blackboard and a corresponding number of events is recorded on the event list. Figure 3-2 shows the serial Cage control cycle.
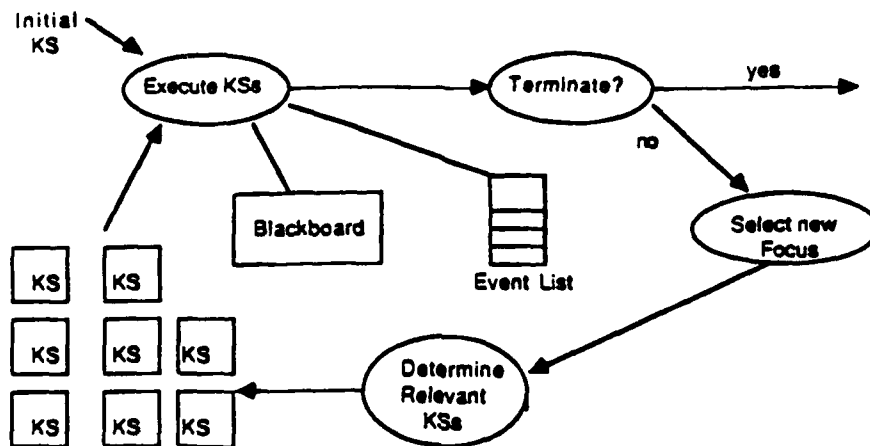
Figure 3-2: CAGE Serial Control Cycle

Using the concurrency control specifications, the user can alter the simple, serial control loop of Cage by requesting the concurrent execution of application components. Cage allows for a range of granularity for these concurrent processes; from knowledge sources all the way down to predicates in the condition parts of rules. The various concurrency operations that can be specified, together with the serial version, are summarized below and shown in Figure 3-1.

Knowledge Source Control

    Serial:

        Pick an event and execute the associated knowledge sources.

    Parallel:

        1. As each event is generated execute the associated knowledge sources in parallel, OR

        2. Wait until all active knowledge sources complete execution, generating a number of events, and then execute the knowledge sources relevant to those events concurrently, OR

        3. Wait until several events are generated then select a subset and execute

the relevant knowledge sources for all the subset events in parallel.

Within Each Knowledge Source
Serial:
1. Perform context evaluation.
2. Evaluate the condition parts, then execute the action part of one rule whose condition side matched, OR
3. Evaluate all the condition parts then execute all the actions of those rules whose condition side matched, serially.

Parallel:
1. Perform context evaluation in parallel.
2. Evaluate all condition parts in parallel, then
   a. synchronize (that is, wait for all the condition side evaluations to complete) and choose one action part, OR
   b. synchronize and execute the actions serially (in lexical order), OR
   c. execute the actions in parallel as the condition parts match.

Within Rules
Serial:
Evaluate each clause then execute each action.
Parallel:
Evaluate the condition-part clauses in parallel then execute the actions of the action part in parallel.

3.2. Discussion of the Concurrent Components
Each of the potential concurrent components are discussed below.

3.2.1. Knowledge Source concurrency:
Knowledge sources are logically independent partitions of the domain knowledge. A knowledge source is selected and executed when changes made to the blackboard are relevant to that knowledge source. Theoretically, many different knowledge sources can be executed at the same time as long as the relevant blackboard changes occur close to each other. But, the knowledge sources are often serially dependent and some synchronization must be introduced.

In the class of applications under consideration, the solution is built up in a pipeline-like fashion up the blackboard hierarchy. That is, the knowledge source dependencies form a chain from the knowledge sources working on the most detailed level of the blackboard to those working on the most abstract level. (When the program is model-driven, this pipeline works in the reverse direction.) Knowledge sources can be running in parallel, processing the data

along the pipe.

Thus, there are two potential sources of knowledge source parallelism: (1) knowledge sources working on different regions (partial solutions) of the blackboard asynchronously, that is, "data parallelism," and (2) knowledge sources working in a pipelined fashion exploiting the flow of information up (or down) the data hierarchy.

3.2.2. Rule concurrency:

Each knowledge source is composed of a number of rules. The condition parts of these rules are evaluated for a match with the current state of the solution, and the action parts of those rules that match the state are executed. The condition parts of all the rules in a knowledge source, being side-effect-free, can be evaluated concurrently. In cases where all the matched rules are to be executed, the action parts can be executed as soon as the condition part is matched successfully. If only one of the rules is to be selected for execution, the system must wait until all the condition parts are evaluated, and one rule, whose action part is to be executed, must be chosen.[3] The situation in which all rules are evaluated and executed concurrently potentially has the most parallelism. However, if the rules access the same blackboard data item, memory contention becomes a hidden point of serialization. At the same time, the integrity of the information on the blackboard cannot be guaranteed. The problem is of two types: timeliness and consistency. First, the state which triggered the rule may be modified by the time the action part is executed. The question is then; is the action still relevant and correct? Second, if a rule accesses attributes from different blackboard objects, there is no guarantee that the values from the objects are consistent with respect to each other.

Condition-part concurrency: Each condition part of a rule may consist of a number of clauses to be evaluated. These clauses can often be evaluated concurrently. In the chosen class of applications, these clauses frequently involve relatively large numeric computations, making parallel evaluation worthwhile. However, as discussed above, if the clauses refer to the same data item, memory contention would force a serialization.

Action-part concurrency: Often, when a condition part matches, more than one potentially independent action is called for, and these can often be executed in parallel.

This problem of data consistency occurs both in Cage and in Poligon. It can be partially alleviated by defining an atomic operation that includes both read and write. This ensures that

---

[3]Note that this is very similar to the OPS conflict-resolution phase. Refer to [Gupta 86] for the results of running OPS rules in parallel.

between the time that an item of data is read, processed, and the result stored, there is no change in the state of the node[4]. However, this makes a commitment to a certain level of granularity, for example, read the data for the condition part of a rule and execute the rule. In order to enable experimentation with granularity, atomic actions are kept small and locks, block reads, and block writes are provided in Cage. Although an atomic read/write operation does not solve the problems of timeliness or of global coherence, it does assure that the data within the nodes are consistent. And, although locks have a potential for causing deadlocks, they are provided for the user to construct larger critical sections.

3.2.3. Concurrency Control

The action parts of rules generate events, and knowledge sources are activated by the occurrences of these events. In the (serial) AGE system events are posted on a global event-list and, working on these events, a control monitor activates one or more knowledge sources. In order to eliminate the serialization inherent in this control scheme, a mechanism to activate the knowledge source immediately upon event generation is needed. This immediate activation of knowledge sources bypasses the control module and effectively eliminates global control. In some cases, this is acceptable. In other cases where knowledge sources are serially dependent, some control mechanism is needed. Centralized control mechanisms, such as selecting many events to be processed in parallel, causing many knowledge sources to run concurrently, are also provided.

Some answers to the many questions raised about Cage's architecture are embedded in the system. However, much of the burden is passed on to the applications programmer. Some useful programming techniques that were discovered are discussed below.

3.3. Programming with Cage

There are a number of problems that crop up during concurrent execution that do not appear during serial execution. The solutions to some of these problems involved reformulating the application problem; some involved the use of programming techniques not commonly used in serial systems. Both Cage and Poligon have been used to implement a signal understanding system called Elint [Brown 86]. It is described briefly below.

---

[4] In Lamina [Delagi 86b], a another programming framework developed for the AAP project, the atomic action is read-process-write.

3.3.1. The Elint Application

The problem is one of receiving multiple streams of reports from radar systems, abstracting these into hypothetical radar emitting aircrafts and tracking them as they travel through the monitored airspace. These aircraft are themselves abstracted into clusters -- perhaps formations -- which are themselves tracked. Sometimes an aircraft in a cluster would split off, forcing the splitting of the cluster node and rationalization of the supporting evidence. The nature of the radar emissions from the aircraft(s) are interpreted in order to determine the intentions and degree of threat of each of the clusters of emitters.

The Elint application has a number of characteristics which are of significance.

- The system must be able to deal with a continuous data stream. It is not acceptable to wait until all of the data has been read in and then figure out what was going on.

- The application domain is potentially very data parallel. The ability to reason about a large number of aircraft simultaneously is very important.

- The aircrafts themselves, as objects in the solution space, are quite loosely coupled.

3.3.2. Pitfalls, Problems and Solutions

The following programming techniques arose while implementing Elint in Cage.

1. When the computational grain size is limited to a knowledge source, it is possible to read all the slots of a node that are referenced in the knowledge source by locking the node once and reading all of the slots at once. This is in contrast to locking the node every time a slot is read by the rules. This is equivalent to reading all of the blackboard data accessed from a knowledge source before any rules are evaluated. This approach accomplishes two important things: (1) It reduces the number of references to the blackboard, thereby reducing the opportunities for memory contention, and (2) it ensures that all the rules are looking at data from the same point in the evolving solution.

2. In a serial blackboard system one precondition may serve to describe several changes to the blackboard adequately. For example, suppose one rule firing causes three changes to be made serially. The last change, or event, is generally a sufficient precondition for the selection of the next knowledge source. In a concurrent system, all three events must be included in a knowledge source's precondition. This is to ensure that all three changes have actually occurred before the knowledge source is executed.

In general, a simple precondition consisting of an event token is not sufficient for Cage. Either a sophisticated scheduler with detailed specification of the activation requirements of the knowledge sources, or a complex, knowledge-source precondition that contain the same

requirements is needed.

3. It is important when writing the conditions of rules for a Cage application to keep in mind the feasibility of running the condition clauses concurrently, that is, keeping them independent of each other in the sense of not accessing the same data.

4. Occasionally two knowledge sources running in parallel may attempt to change a slot at almost the same time. It is possible that the first change would invalidate the firing of the second rule. To overcome this type of race condition, a conditional action -- an action which checks the value of a slot before making a change -- was added. It allows the action to check the most recent updates before making further changes. The alternative would have been to lock a node for an entire knowledge source execution which would seriously limit parallelism.

3.3.3. A problem with Continuous Input Streams

Since Elint is a real-time system, it is time dependent. Processing a continuous stream of data can lead to *out-of-order events* caused by delay of one kind or another; an example might be a knowledge source stuck in a memory queue delaying its changes to the blackboard. This means that new data at time t may have to be analyzed before all the ramifications of data from an earlier time ($t - n$) have been executed -- at any point the data can be out of order. The Elint application had to be reformulated to address this problem. Time tags had to be associated with each event and blackboard value, and the rules had to be re-written to use the time tags to reason about unordered events.

3.3.4. Incremental Introduction of Parallelism

Experiments with Cage indicate that it is much more difficult to program a parallel system than a serial one. It lends subjective support to our supposition that an incremental approach to parallelism is easier to program than an all-at-once approach. We began with a serial version of Elint and turned on clause level concurrency first and debugged it, then experimented with rule level, and finally knowledge source level concurrency. Only after Elint was working correctly with each of the these concurrent operations, were they combined.

As discussed earlier, Cage can execute multiple sets of rules, in the form of knowledge sources, concurrently. If the rule parallelism within each knowledge source can provide a speed-up in the neighborhood cited by Gupta, and if many knowledge sources can run concurrently without getting in each other's way, we can hope to get a speed up in the tens. The extra parallelism comes from working on many parts of the blackboard, in other words, by solving many sub-problems in parallel. It was found, however, that the use of a central controller to determine which knowledge sources to run in parallel drastically limits speed-up, no matter how many knowledge sources are executed in parallel. Amdahl's limit and synchronization come strongly into play. The implication for Cage is that knowledge-source invocation should be distributed,

without synchronization. This will eliminate two major bottlenecks -- a data-hot spot at the event list, and waiting for the slowest process to finish during synchronization. Still, within a shared-memory, multi-processor system, the interface to the blackboard is a bottleneck. One solution to this is to distribute the blackboard, which is one of the main characteristics of Poligon.

4. Pursuing a Daemon-driven Blackboard System - Poligon

Control in the blackboard model could be summarized as follows: *knowledge sources respond opportunistically to changes in the blackboard.* As discussed earlier, in reality, and especially in serial systems, the blackboard changes are recorded and a control module decides which change to pursue next. In other words, the knowledge sources do not respond directly to changes on the blackboard. A control module generally dictates the problem-solving behavior. This is a serializing process.

The basic question that led to the design of Poligon is: *What if we attach the knowledge sources to the data elements in the blackboard which, when changed, would result in the activation of those knowledge source?* Instead of waiting until a control module activates a knowledge source, why not immediately execute the knowledge source as the relevant data are changed, and get rid of the control module? A blackboard change would serve as a direct trigger for knowledge source activations. Next, assign a processor-memory pair for each blackboard node, and have the knowledge sources (now on the blackboard processing element) communicate changes to other nodes by passing messages via a communication network. (see Figure 4-1).

Because a knowledge source is activated by a blackboard change, and because a knowledge source is a collections of rules, one can view the rules as being activated (indirectly, to be sure) by a change to some blackboard node. A rule could be activated by a change to a particular slot on a blackboard node. Slots with a property that trigger rules are called "trigger slots". When the action part of a rule is executed, the changes to the blackboard are communicated to the nodes to be changed. If a change is made to a trigger slot, then the condition parts of the "triggered rules" are evaluated; changes to non-trigger slots do not directly cause any processing.

Poligon was designed from the start to exploit "fine"-grained parallelism -- "fine" grain here referring to parts of rules. It is generally thought that a shared-memory hardware architecture is not able to deliver increasing performance as more processors are added. This is a result of memory contention an.' of physical limits in the bandwidths of the busses and switches used to connect the processors to the memory. Thus, Poligon was designed from the start to be run on a form of distributed-memory multiprocessor, the elements of which communicate by sending
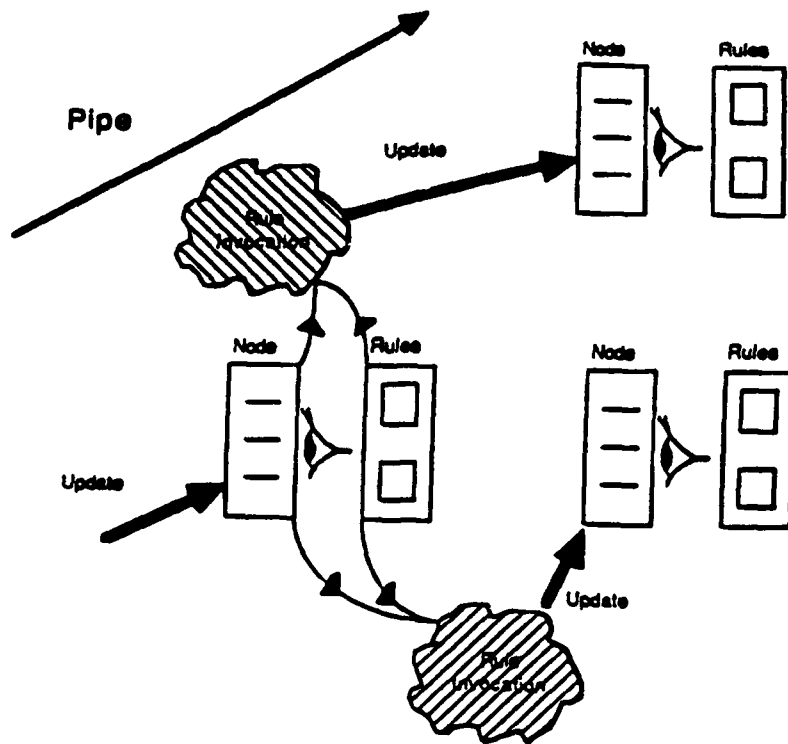
Figure 4-1: Organization of Poligon

messages to one another. Its match to the hardware will be seen clearly in the next section where we discuss the structure of Poligon and what makes it different from existing, serial implementations of blackboard systems.

4.1. The Structure of Poligon

In this section we describe the key features of Poligon. Instead of a detailed description of the implementation, a number of points which are central to Poligon's computational model are highlighted and contrasted with conventional blackboard implementations.

As has been mentioned above, Poligon is designed to run on hardware which provides message-passing primitives as the mechanism for communication between processing elements. It is important to note that the way in which information flows on the blackboard can be viewed, at an implementation level, as a message-passing process. This allows a tight coupling between the implementation of a system such as Poligon and the underlying hardware.

- *Poligon has no centralized scheduler.* This was motivated by a desire to remove any bottlenecks that might be caused by the serial execution of such a scheduler and by multiple, asynchronous processes trying to put events onto the scheduler queue, causing memory contention. (The problems was clearly manifested in Cage.) This required the definition of a different knowledge invocation mechanism. Not only was a centralized scheduler eliminated but all global synchronization was eliminated as well. This means that it is likely that different parts of a Poligon program will run at different speeds and will have have different ideas of how the solution is

progressing.

- Having eliminated the scheduler, there is clearly no need for any -- presumably serializing -- separation of the knowledge sources from the blackboard. The Poligon programmer, therefore, specifies at compile-time the classes of blackboard node that a particular piece of knowledge is interested in. At compile-time and at system initialization time, *knowledge is associated directly with the nodes on the blackboard that might invoke it.* This eliminates any communication delay and memory contention that might be caused by having to find a matching rule in a remote knowledge base.

- In conventional blackboard systems, knowledge sources are taken to denote both units of knowledge and units of scheduling. If all that a system attempted to execute in parallel was its knowledge sources then a great deal of potential parallelism might be lost by the failure to exploit parallelism at a finer grain. In Poligon, therefore, *knowledge sources are not scheduling units,* they are simply collections of knowledge. *All of the rules in a knowledge source can, in principle, be invoked in parallel and parallelism at a finer grain than this can also be exploited during the execution of rules.*

- Having eliminated the scheduler a new mechanism was needed that would cause the application's knowledge to be executed. It was decided to go for a very simple mechanism. *Poligon's rules are triggered as daemons by updates to slots in nodes.* The association between rules and the slots that trigger their invocation is made at compile-time, allowing efficient, concurrent invocation of all eligible rules after an event on the blackboard.

- The message-passing metaphor for the implementation and the distribution of the knowledge base over the blackboard mentioned above, allowed the development of a *computational model which views a blackboard node as a process, responsible for its own housekeeping and for processing messages,* for instance, for slot updates and slot read operations.

- Serial blackboard systems generally don't have a significant problem with the creation of new blackboard nodes. This is because of the atomic execution of knowledge sources. Such systems can usually be confident that, when a new node is created, no other node has been created that represents the same object. In parallel systems multiple, asynchronous attempts can be made to create nodes which are really intended to represent the same real-world object. Poligon provides mechanisms to allow the user to prevent this from happening.

- It was found necessary occasionally to share data between a number of nodes. Poligon ...ows no global variables at all so it was necessary to find a suitable way of defining sharable, mutable data, whilst still trying to reduce the bottlenecks that can be caused by shared data structures. Poligon, like many frame systems, has a generalized class hierarchy with the classes themselves being represented as blackboard nodes. *Poligon uses class nodes as managers,* not only for node

creation, as mentioned above, but also to store data to be shared between all of the instances of a class and to support operations which apply to all members of a class.

• Most blackboard systems represent the slots in nodes simply as value lists associated with the name of the slot. The serial operation of such systems allows the programmer to make assumptions about the order of elements in the value list. This assumption allows operations on all of the elements of the value list in the knowledge that no modification will have happened to the value list since it was read, because knowledge source executions are atomic. In Poligon, because a large number of rules can asynchronously be attempting to perform operations on a slot simultaneously, it was imperative to find mechanisms that would help to keep the operation of the system coherent without slowing down the access to slots too much, causing large critical sections and reducing parallelism. *Poligon, therefore, provides "smart" slots.* They can keep their values in the correct order and index them for flexible and focused data retrieval. They can also have user defined behavior which allows them to make sure that operations performed on them leave them consistent.

4.2. Shifting the Metaphor

Poligon's design looks very much like a frame-based program specialized for a particular implementation of the blackboard model. The expected behavior of the system is much closer than the serial systems to the blackboard problem-solving metaphor in one respect -- the knowledge sources respond to changes in the blackboard *directly*[5]. As in Cage there are two major sources of concurrency in this scheme: (1) Each blackboard node can be active simultaneously to reflect data parallelism -- the more blackboard nodes, the more potential parallelism. (2) Rules attached to a node can be running on many different processing elements simultaneously providing knowledge parallelism. This daemon-driven system with a facility for exploiting both data and knowledge parallelism poses some serious problems, however. First, it is easy to keep the processors and communication network busy, but the trick is to keep them busy converging toward a solution. Second, solutions to a problem will be non-deterministic -- that is, each run will most likely produce different answers. Worse, a solution is not guaranteed since individual nodes cannot determine if the system is on the right path to an overall solution -- that is, there is no global control module to steer the problem solving. Within the AI paradigm that looks for satisficing answers, non-determinism, per se, is not a cause for alarm; however, non-convergence or an incorrect solution is. One remedy to these problems is to introduce some global control mechanisms. Another solution is to develop a problem-solving scheme that can operate without a global view or global control. We have

---

[5]As an historical note, this takes us back to Selfridge's Pandemonium [Selfridge 59], which influenced Newell's ideas of blackboard-like programs [Newell 62]. It also has some of the flavor of the actor formalism [Hewitt 73].

focussed our efforts in Poligon on the latter approach.

4.2.1. Distributed, Hierarchical Control

A hierarchical control mechanism is introduced that exploits the structure of the blackboard data. The levels, in the AGE sense, of the blackboard are organized as a class hierarchy. Each level is a class and a blackboard node is an instance of that class. Class nodes contain information about their instances (number of instances, their address, and so on), and knowledges sources can be attached to class nodes to control their instance nodes. To minimize confusion, class nodes will be referred to using a more concrete term, *level manager*. Similarly, a super-manager node can control the class nodes.

Within Poligon, the potential for control is located in three types of places:

1. Within each node, where action parts of the rules can be executed serially, for example.

2. In the level manager which can, for example, be used to monitor the activities of its nodes. Since the level manager is the only agent that knows about the nodes on its level, a message that is to be sent to all the instance nodes must be routed through their manager node. The level manager also controls the creation and garbage collection of the nodes, and attaches the relevant rules to newly created nodes.

3. In the super-manager, whose span of control includes the creation of level managers and their activities, and indirectly their offspring.

The introduction of control mechanisms solves some of the difficulties, but it also introduces bottlenecks at points of control, for example, at the level manager nodes. One solution to this type of bottleneck is to replicate the nodes, that is, create many copies of the manager nodes. The CAOS experiments, mentioned earlier, took this approach [Brown 86]. Although Poligon supports this strategy, our research is leading us to try a different tactic.

4.2.2. A New Role for Expectation-driven Reasoning

It was initially conjectured that model-driven and expectation-driven processing would not play a significant role in concurrent systems -- at least not from the standpoint of helping with performance. One view of top-down processing is that it is a means of gaining efficiency in serial systems in the following way: In the class of applications under consideration, the interpretation of data proceeds from the input data up an abstraction hierarchy -- the amount of information being processed is reduced as it goes up the hierarchy. Expectations, posted from a higher level to a lower level, indicate data needed to support an existing hypothesis; data expected from predictions; and so on. Thus, when an expected event does occur, the bottom-up analysis need not continue up -- the higher level node is merely notified of the

event and it does the necessary processing, for example, increases the confidence in its hypothesis. When the analysis involves a large search space, this expectation-driven approach can save a substantial amount of processing time in serial systems.

In Poligon hot-spots often occur at a node to which many lower level nodes communicate their results (a fan-in). The upward message traffic can be reduced by posting expectations on the lower level nodes and having them report back only when *unexpected* events occur. This approach, currently under investigation, is one way for a node to distribute parts of the work to lower level nodes, and hopefully relieves the type of bottlenecks caused by fan-ins at a node without restoring to node replication.

It is generally expected that, within the abstraction hierarchy of the blackboard, information volume is reduced as one goes up the hierarchy. This translates into the following desiderata for concurrent systems: For an arbitrary node to avoid being a hot-spot, there must be a decrease in the rate of communication proportional to the number of nodes communicating to it. That is, the wider the fan, the less communication is allowable from each node. It was found while re-implementing the serial ELINT application in Poligon, that the highest level nodes had to be updated for almost every new data item. Such a formulation of the problem, while posing no problem in serial systems, reduces parallelism in concurrent problem solvers.

4.2.3. A New Form of Rules

If, for any given data item, there are many rules that check its state, then the system must ensure that this data item does not change until all of those rules have checked it. A typical example is as follows: Suppose there are two rules that are mutually exclusive, one performs some action if a data value is "on" and the other performs some other action if the value is "off." How can we ensure that between the time the first rule accesses the data and the second does so, there is not some other action that changes the data? In was found in Poligon (and also in Cage) that these mutually exclusive rules need to be written in the form of case-like conditionals to assure data consistency of the form described above. Since the need for process creation, and subsequent maintenance, is reduced through combining rules, this form of rule also aids in speeding up the overall rule execution. It does mean, however, that the grain size of some of the rules has been made bigger, at least at the source code level, and the programmers must think differently about rules than they do in current expert systems.

4.2.4. Agents with Objectives

At any given point in the computation, the data at different nodes can be mutually inconsistent or out of date. There are many causes for this, but one cause is that blackboard changes are communicated by messages and the message transit time is unpredictable. In the applications under consideration, where there are one or more streams of continuous input data, the problem appears as scrambled data arrival -- the data may be out of temporal

sequence or there may be holes in the data. Waiting for earlier data does not help, since there is no way to predict when that data might appear. Instead, the node must do the best it can with the information it has. At the same time, it must avoid propagating changes to other nodes if its confidence in its output data or inferences is low.

Put another way, *each node must be able to compute with incomplete or incorrect data, and it must 'know' its objectives to enable it to evaluate the resulting computation. A result is passed on only if it is known to be an improvement on a past result.* This represents a change from the problem-solving strategies generally employed in blackboard systems where the control/scheduling module evaluates and directs the problem solving. With no global control module to evaluate the overall solution state and with asynchronous problem-solving nodes, a reasonable alternative is to make each node evaluate its own local state. Of course, there is no guarantee that the sum total of local correctness will yield global correctness. However, the way that blackboard systems are generally organized -- each blackboard level representing a class of solution islands, the span of knowledge sources being limited to a few levels, and having functionally independent knowledge sources -- appears at this point, to provide an appropriate methodology for creating loosely-coupled nodes that can be provided with local objectives and a capability for self-evaluation.[6]. The "smart" slots mentioned earlier are used to implement this strategy.

The design of Poligon poses an interesting question -- is it still a blackboard system? There is a substantial shift in the problem-solving behavior and in the way the knowledge sources need to be formulated. The structure of the solution is not globally accessible. There is no control module to guide the problem solving at run time. The metaphor shifts to one in which each "blackboard" node is assigned a narrow objective to achieve, doing the best it can with the data passed to it, and passing on information only when the new solution is better than the last one. The collective action of the "smart" agents results in a satisficing solution to a problem[7].

---

[6]It is interesting to note that the need for local goals does not seem to change with process granularity. Although the methods used to generate the goals are very different, Lesser's group has found that each node in its distributed system needs to have local goals [Durfee 85]. In this system each node contains a complete blackboard system; each system (node) monitors the activities in a region of a geographic area which is monitored collectively by the system as a whole.

[7]In retrospect, these characteristics for concurrent problem solving seem obvious. When a group of humans solve a problem collectively by subdividing a task, we assume each person has the ability to evaluate his or her own performance relative to the assigned task. When there are "uncaring" people, the overall performance is bad, both in terms of speed and solution quality.

Although there is a substantial shift away from the conventional problem solving metaphor, Poligon evolved out of the mechanisms that were present in AGE. Most of the same opportunities for concurrency made available to the user in Cage are built into the system in Poligon. The Poligon language forces the user to think in terms of blackboard levels and knowledge sources. But the underlying system has no global data. Whether such a formulation makes the job of constructing concurrent, knowledge-based systems easier or more difficult for the knowledge engineer still remains to be seen. A difficulty might arise because the semantics of the Poligon language, that is, the mapping of the blackboard model to the underlying software and hardware architecture, is hidden from the user. For example, there is no notion of message-passing or of a distributed blackboard reflected in the Poligon language. In contrast, the choice of what, and how, to run concurrently is completely under user control in Cage.

5. Conclusions

In this paper we discussed the relationship between the blackboard model, its existing serial implementations, and the degree to which the intuitively inherent parallelism is really present.

Cage and Poligon, two implementations of the blackboard model designed to operate on two different parallel hardware architectures, were described briefly, both in terms of their structure and the motivation behind their design.

Our framework development, application implementations on these frameworks, and initial performance experiments to date has taught us that: (1) it is difficult to write a real-time, data interpretation programs in a multi-processor environment, and (2) performance gains are sensitive to the ways in which applications are formulated and programmed. In this class of application, performance is also sensitive to data characteristics.

The "obvious" sources of parallelism in the blackboard model, such as the concurrent processing of knowledge sources, do not provide much gain in speed-up if control remains centralized. On the other hand, decentralizing the control, or removing the control entirely, creates a computational environment in which it is very difficult to control the problem-solving behavior and to obtain a reasonable solution to a problem. As granularity is decreased, to obtain more potential parallel components, the interdependence among the computational units tends to increase. making it more difficult to obtain a coherent solution *and* to achieve a performance gain at the same time. We described some of the methods employed to overcome these difficulties.

In the application class under investigation, much of the parallelism came from data parallelism -- both from the temporal data sequence and from multiple objects (aircrafts, for

example) -- and from pipe-lining up the blackboard hierarchy. The ELINT application was unfortunately knowledge poor, so that we were unable to explore knowledge parallelism, except as a by-product of data and pipeline parallelism. ELINT has been implemented in both Cage and Poligon, and experiments are now being performed. The experiments are designed to measure and to compare performance by varying different parameters: process granularity, number of processors, data rate, data arrival characteristics, and so on.

It is clear that much more research is needed in this area before a combination of a computational and problem-solving model can be developed that is easy to use, that produces valid solutions reliably, and that can increase performance by a significant amount.

References

[Aiello 86]   Nelleke Aiello.
User-Directed Control of Parallelism: The CAGE System.
Technical Report KSL Report 86-31, Knowlege Systems Laboratory, Computer
    Science Department, Stanford University, April, 1986.

[Amdahl 67]   Gene M. Amdahl.
Validity of a Single Processor Approach to Achieveing Large Scale Computing
    Capabilities.
Proceedings of AFIPS Computing Conference 30, 1967.

[Brown 82]   Harold Brown, Jack Buckman, et. al.
Final Report on HANNIBAL.
Technical Report, ESL, Inc., 1982.
Internal document.

[Brown 86]   Harold D. Brown, Eric Schoen, and Bruce Delagi.
*An Experiment in Knowledge-based Signal Understanding Using Parallel
    Architectures.*
Technical Report KSL-86-69, Knowledge Systems Laboratory, Computer Sciece
    Department, Stanford University, October, 1986.

[Corkill 83]   Daniel D. Corkill and Victor R. Lesser.
The Use of Meta Level Control for Coordination in a Distributed Problem
    Solving.
Procceding of the 7 th International Conference on Artificial Intelligence
    :748 -755, 1983.

[Delagi 86a]   Bruce Delagi.
CARE Users Manual.
Technical Report KSL-86-36, Knowledge Systems Laboratory, Computer
    Science Department, Stanford University, 1986.

[Delagi 86b]   Bruce A. Delagi, Nakul P. Saraiya, Gregory T. Byrd.
LAMINA: CARE Applications Interface.
Technical Report KSL-86-76, Knowledge Systems Laboratory, Computer
    Science Department, Stanford University, 1986.

[Durfee 85]   Edmund Durfee, Victor Lessor, and Daniel Corkill.
Coherent Cooperation Among Communicating Problem Solvers.
Technical Report, Department of Computer and Information Sciences,
    September, 1985.

[Ensor 85]   J. Robert Ensor and John D. Gabbe.
Transactional Blackboards.
*Proceedings of the 9th International Joint Conference on Artificial
    Intelligence* :340 - 344, 1985.

[Fennell 77]   Richard D. Fennell and Victor R. Lesser.
Parallelism in AI Problem Solving: A Case Study of HEARSAY-II.
IEEE Transactions on Computers :98 - 111, February, 1977.

[Forgy 77]   Charles Forgy and John McDermott.
                OPS, A Domain-Independent Production System Language.
                Proceedings of the 5th International Joint Conference on Artificial
                    Intelligence 1:933-939, 1977.

[Gupta 86]   Anoop Gupta.
                Parallelism in Production Systems.
                PhD thesis, Department of Computer Science, Carnegie-Mellon University,
                    March, 1986.

[Hewitt 73]   Hewitt, C., P. Bishop, and R. Steiger.
                A Universal, Modular Actor Formalism for Artificial Intelligence.
                Proceedings of the 3rd International Joint Conference on Artificial
                    Intelligenc·:235 - 245, 1973.

[Lesser 83]   Victor R. Lesser and Daniel D. Corkill.
                The Distributed Vehicle Monitoring Testbed: A Tool for Investigation
                    Distributed Problem Solving Networks.
                The AI Magazine Fall:15 - 33, 1983.

[Mccune 83]   Brian P. McCune and Robert J. Drazovich.
                Radar with Sight and Knowledge.
                Defense Electronics , August, 1983.

[McDermott 83] John McDermott and Allen Newell.
                Estimating the Computational Requirements for Future Expert Systems.
                Technical Report, Internal Memo, Computer Science Department, Carnegie-
                    Mellon University, 1983.

[Newell 62]   Allen Newell.
                Some Problems of Basic Organization in Problem-Solving Programs.
                In M. C. Yovits, G. T. Jacobi, and G.D. Goldstein (editors), *Conference on*
                    Self-Organizing Systems, pages 393 - 423. Spartan Books, Washington,
                    D.C., 1962.

[Nii 79]   H. Penny Nii and Nelleke Aiello.
                AGE: A Knowledge-based Program for Building Knowledge-based Programs.
                Proc. of IJCAI 6 :645 - 655, 1979.

[Nii 82]   H. Penny Nii, Edward A. Feignebaum, John J. Anton, and A. Joseph
                    Rockmore.
                Signal-to-Symbol Transformation: HASP/SIAP Case Study.
                AI Magazine 3:2:23 - 35, 1982.

[Nii 86]   H. Penny Nii.
                Blackboard Systems.
                Technical Report KSL-86-18, Knowledge Systems Laboratory, Computer
                    Science Department, Stanford University, April, 1986.
                Also in *AI Magazine*, vol. 7-2 and vol. 7-3, 1986.

[Rice 86]   James Rice.
                Poligon: A System for Parallel Problem Solving.
                Technical Report KSL-86-19, Knowledge Systems Laboratory, Computer
                    Science Department, Stanford University, April, 1986.

[Schoen 86]   Eric Schoen.
                The CAOS System.
                Technical Report KSL-86-22, Knowledge Systems Laboratory, Computer
                    Science Department, Stanford University, April, 1986.
                Also in this Proceedings.

[Selfridge 59] Oliver G. Selfridge.
                Pandamonium: A Paradigm for Learning.
                Proceedings of the Symposium on the Mechanization of Thought Processes
                    :511 - 529, 1959.

[Shafer 86]   Steven A. Shafer, Anthony Stentz, and Charles Thorpe.
                An Architecture for Sensor Fusion in a Mobile Robot.
                *Proceedings of the 1986 IEEE International Conference on Robotics and
                    Automation* , 1986.
                To appear in the proceedings.

[Smith 81]   Smith, B.J.
                Architecture and Applications of the HEP Multiprocessor Computer System.
                In *The Proceedings of the International Society for Optical Engineering.* San
                    Diego, California, August 25-28, 1981.

[Spain 83]   David S. Spain.
                Application of Artificial Intelligence to Tactical Situation Assessment.
                Proceedings of the 16th EASCON83 :457 - 464, September, 1983.

[Williams 84]   Mark Williams, Harold Brown, and Terry Barnes.
                TRICERO Design Description.
                Technical Report ESL-NS539, ESL, Inc., May, 1984.

DISTRIBUTION LIST

addresses	number of copies
Northrup Fowler III RADC/COES	20
RADC/DOVL GRIFFISS AFB NY 13441	'
RADC/DAP GRIFFISS AFB NY 13441	2
ADMINISTRATOR DEF TECH INF CTR ATTN: DTIC-DDA CAMERON STA BG 5 ALEXANDRIA VA 22304-6145	5
RADC/COTD BLDG 3, ROOM 16 GRIFFISS AFB NY 13441-5700	1
HQ USAF/SCTT Pentagon Wash DC 20330-5190	1
DIRECTOR DMAHTC ATTN: SDSIM Wash DC 20315-0030	1

Director, Info Systems   1
CASD (C3I)
Rm 3E127
Pentagn
Wash DC 20301-3040

Fleet Analysis Center   1
Attn: GIDEP Operaticns Center
Code 30G1 (E. Richarcs)
Corona CA 91720

HQ AFSC/DLAE   1

 ANDREWS AFB DC 20334-5000

HQ AFSC/XRK   1
ANDREWS AFB MD 20334-500

HQ SAC/SCPT   1
OFFUTT AFB NE 68113-5001

CTESA/RQEE   1
ATTN: LARRY G. MCMANUS
2501 YALE STREET SE
Airport Plaza, Suite 102
ALBUQUERQUE NM 87106

HQ TAC/DRIY   1
Attn: Mr. Westerman
Langley AFB VA 23665-5001

HQ TAC/DRCA   1
LANGLEY AFB VA 23665-5001

AUL/LSE
MAXWELL AFB AL 36112-5564   1

ASD/ENEMS   2
Wright-Patterson AFB OH 45433-65C3

ASD-AFALC/AXP   1
WRIGHT-PATTERSON AFB OH 45433

ASD/AFALC/AXAE   1
Attn: W. H. Dungey
Wriight-Patterson AFB OH 45433-6533

AFIT/LDEE   1

BUILDING 640, AREA B
WRIGHT-PATTERSON AFB OH 45433-6583

AFWAL/MLPO   1
WRIGHT-PATTERSON AFB OH 45433-6533

AAMRL/HE   1
WRIGHT-PATTERSON AFB OH 45433-6573

Air Force Human Resources Laboratory   1
Technical Documents Center
AFHRL/LRS-TDC
Wright-Patterson AFB OH 45433

2750 ABW/SSLT   1
Bldg 262
Post 11S
Wright-Patterson AFB OH 454433

```
Defense Communications Engineering Ctr          1
Technical Library
1860 Wiehle Avenue
Reston VA 22090-5500


COMMAND CONTROL AND COMMUNICATIONS DIV          2
DEVELOPMENT CENTER
MARINE CORPS DEVELOPMENT & EDUCATION   COMMAND
ATTN:   CODE DIOA
GUANTICO VA 22134-5080


AFLMC/LGY                                        1
ATTN:   CH, SYS ENGR DIV
GUNTER AFS AL 36114


U.S. Army Strategic Defense Command             1
Attn:  DASD-H-MPL
P.O. Box 1500
Huntsville AL 35807-3801


COMMANDING OFFICER                              1
NAVAL AVIONICS CENTER
LIBRARY - D/765
INDIANAPOLIS IN 46219-2189


COMMANDING OFFICER                              1
NAVAL TRAINING SYSTEMS CENTER
TECHNICAL INFORMATION CENTER
BUILDING 2068
ORLANDO FL 32813-7100


COMMANDER                                       1
NAVAL OCEAN SYSTEMS CENTER
ATTN:   TECHNICAL LIBRARY, CODE 9642B
SAN DIEGO CA 92152-5000


COMMANDER (CODE 3433)                           1
ATTN:   TECHNICAL LIBRARY
NAVAL WEAPONS CENTER
CHINA LAKE, CALIFORNIA 93555-6001
```

SUPERINTENDENT (CODE 1424)   1
NAVLA POST GRADUATE SCHOOL
MONTEREY CA 93943-5000

COMMANDING OFFICER   2
NAVAL RESEARCH LABORATORY
ATTN: CODE 2627
WASHINGTON DC 20375-5000

SPACE & NAVAL WARFARE SYSTEMS COMMAND   1
PMW 153-30P
ATTN: R. SAVARESE
WASHINGTON DC 20363-5100

CDR, U.S. ARMY MISSILE COMMAND   2
REDSTONE SCIENTIFIC INFORMATION CENTER
ATTN: AMSMI-RD-CS-R (DOCUMENTS)
REDSTONE ARSENAL AL 35898-5241

Advisory Group on Electron Devices   2
Hammond John/Technical Info Coordinator
201 Varick Street, Suite 1140
New York NY 10014

UNIVERSITY OF CALIFORNIA/LOS ALAMOS   1
NATIONAL LABORATORY
ATTN: DAN BACA/REPORT LIBRARIAN
P.O. BOX 1663, MS-P364
LOS ALAMOS NM 87545

RAND CORPORATION THE/LIBRARY   1
HELFER DORIS S/HEAD TECH SVCS
P.O. BOX 2138
SANTA MONICA CA 90406-2138

AEDC LIBRARY (TECH REPORTS FILE)   1
MS-100
ARNOLD AFS TN 37389-9998

USAG   1
Attn: ASH-PCA-CRT
Ft Huachuca AZ 85613-6000

JTFPMO   1
Attn: Technical Director
1500 Planning Research Drive
McLean VA 22102

AFEWC/ESRI   4
SAN ANTONIO TX 78243-5000

485 EIG/EIER (DMC)   2
GRIFFISS AFB NY 13441-6348

ESD/AVS   1
ATTN: ADV SYS DEV
HANSCOM AFB MA 01731-5000

ESD/ICP   1
HANSCOM AFB MA 01731-5000

ESD/AVSE   2
BLDG 1704
HANSCOM AFB MA 01731-5000

HQ ESD SYS-2   1
HANSCOM AFB MA 01731-5000

The Software Engineering Institute
Attn: Major Dan Burton
Joint Program Office   1
Carnegie-Mellon University
Pittsburgh PA 15213-3890

DIRECTOR   1
NSA/CSS
ATTN: T513/TDL (DAVID MARJARUM)
FORT GEORGE G MEADE MD 20755-6000

```
DIRECTOR                                              1
NSA/CSS
ATTN:  R24
FORT GEORGE G MEADE MD 20755-6000


DIRECTOR                                              1
NSA/CSS
ATTN:  R21
9800 SAVAGE ROAD
FORT GEORGE G MEASDE MD 20755-6000

DIRECTOR                                              1
NSA/CSS
ATTN:  R5
FORT GEORGE G MEADE MD 20755-6CCC


DIRECTOR                                              1
NSA/CSS
ATTN:  R8
FORT GEORGE G MEADE MD 20755-6CCC


DIRECTOR                                              1
NSA/CSS
ATTN:  SC31
FORT GEORGE G MEADE MD 20755-6000


DIRECTOR                                              1
NSA/CSS
ATTN:  S21
FORT GEORGE G MEADE MD 20755-6000


DIRECTOR                                              1
NSA/CSS
ATTN:  V33 (S. Friedrich)
FORT GEORGE G MEADE MD 20755-6000


DIRECTOR                                              1
NSA/CSS
ATTN:  W3
FORT GEORGE G MEADE MD 20755-6000
```

DoD COMPUTER SECURITY CENTER   1
ATTN: C4/TIC
9800 SAVAGE ROAD
FORT GEORGE G MEADE MD 20755-6000

Dr. Edward A. Feigenbaum   5
Knowledge Systems Laboratory
Stanford University
701 Welch Road, Bldg C
Palt Alto, CA 94304

ESD-MITRE Software Center Library   2
c/o Ms J.A. Clapp
MITRE Corp, D-70
Burlington Road
Bedford, MA 01730

Software Engineering Institute Tech Lib.   2
ATTN: Korola Fuchs
Carnegie--Mellon University
Pittsburgh, PA 15232

Col J. Green   2
Director, STARS JPO
Rm C-107
1211 South Fern Street
Arlington, VA 22202

                                                     1
Dr. Steven Vere
Lockheed AI Center, SC-06/259
Lockheed Research & Development Division
3251 Hanover St.

Dr. Austin Tate, Director   1
AIAI
University of Edinburgh
80 South Bridge
Edinburgh, Scotland EHI IHV

Dr. Kevin J. Greene   1
Case Center
Syracuse University
Syracuse, N.Y. 13244

N.S. Sridharan   1
FMC Corp
Central Eng. Lab
1205 Coleman Ave, Box 580
Santa Clara, CA 95052

Joseph C. Batz   1
ADA Joint Program Office
1211 South Fern Street
Arlington, VA 22202

Maj William R. Price   1
Automated Systems Program Office
ASPO/PYGW
Gunter AFS, Alabama 36114-6340

Mr Robert Drazovich   1
Advanced Decision Systems
201 San Antonio Circle , Suite 286
Mountain View, CA 94040

Dr Brian P. McCune   1
Advanced Decision Systems
201 San Antonio Circle, Suite 286
Mountain View, CA 94040

Daniel G. Shapiro   1
Advanced Decision Systems
201 San Antonio Circle, Suite 286
Mountain View, CA 94040

Dr Richard Wishner   1
Advanced Decision Systems
201 San Antonio Circle, Suite 286
Mountain View, CA 94040

Dr. Drew McDermott   1
Dept of Computer Science
Yale University
P.O. Box 2153 Yale Station
New Haven, CT 06520

W. A. Frawley   1
GTE Labs
40 Sylvan Road   .
Waltham, MA   02254

Dr. John D. Ramsdell   1
The MITRE Corporation
Burlington Road
Bedford, MA 01730

Lt C. Hugh L. Burns
AFHRL/IDI
Brooks AFB, TX 78235

1   1
Maj Stephen E. Cross
AFWAL/AAAA
Wright-Patterson AFB, OH 45433

Capt Dan Snyder   1
AAMRJ/HEC
Wright-Patternson AFB, OH 45433-6573

Dr. Randell Schumaker   1
Code 7510
NRL
4555 Overlook Ave, SW
Washington, DC 20375

Dr. Gerald M. Powell   1
HQ CECOM
COMM-ADP
ATTN: AMSEL-COM-IR-1
Fort Monmouth, NJ 07703-5204

Dr. John Dimmock   1
Technical Director
AFOSR/CD
Bolling AFB, DC 20332

                                                    1

Dr Abe Waksman
AFOSR/NM
Bolling AFB DC 20332

                                                    1

Director
Mathematical & Information Sciences
AFOSR/NM
Bolling AFB, DC 20332

                                                    1

Chief Scientist
AFSC/DLZ
Andrews AFB, DC 20334

Capt Carl S. Lizza   1
AFWAL/CCU
Wright-Patterson AFB, OH 45433-6523

Dr Nils R. Sandell   1
Alphatech, Inc.
3 New England Executive Park
Burlington, MA 01303

Lee Duke   1
Ames Dryden Flight Test Center
P. O. Box 273
Edwards AFB, CA 93523

Susan Ennis   1
Amoco Production
P. O. Box 3385
Tulsa, OK 74102

Dr Ray Sidorski   1
ARI
5001 Eisenhower Avenue
Alexandria, VA 22333

Director, U.S. Army Ballistic Research Lab   1
ATTN: AMXBR-SECAD (Richard Kaste)
Aberdeen Proving Ground, MD 21005-5066

U.S. Army Communications-Electronics Command   1
ATTN: AMSEL-TCS-CR (Martin I. Wolfe)
Fort Monmouth, NJ 07703

Dr Willard Holmes   1
U.S. Army Missile Command
Systems Simulation and Development Directorate
AMSMI-RDW
Redstone Arsenal, AL 35898-5252

Dr Jimmie R. Suttle   1
Army Research Office
Electronics Division
P. O. Box 12211
Research Triangle Park, NC 27709

Dr Gordon D. Prichett   1
Babson College
Math Department
Babson Park, MA 02157-0901

Russ Bennett
Bennett Enterprises   1
27012 Floresta
Mission Viejo, CA 92691

Robert Lawler
Boeing Computer Services   1
Advanced Technology Applications Division
P.O. Box 24346
Seattle, WA 98124-0346

Lester R. LaBrecque   1
General Electric Company
901 Broad Street MD 716
Utica, NY 13503

Art Nagai   1
Boeing AI Center
MS 7A-03
P.O. Box 24346
Seattle, WA 98124

R. Bruce Roberts   1
Bolt Beranek and Newman Inc.
Department of Artificial Intelligence
10 Moulton Street
Cambridge, MA 02238

Dr Albert L. Stevens   1
Bolt Beranek and Newman Inc.
Department of Artificial Intelligence
10 Moulton Street
Cambridge, MA 02238

Dr Eugene Charniak   1
Brown University
Box 1910
Providence, RI 02912

Dr Gary Kahn   1
Carnegie Group, Inc.
Station Square
Pittsburgh, PA   15219

Dr Jaime Carbonell   1
Carnegie-Mellon University
Computer Science Department
Schenley Park
Pittsburgh, PA 15213

Dr Mark Fox   1
Carnegie-Mellon University
Intelligent Systems Laboratory
The Robotics Institute
Pittsburgh, PA 15213

Dr Clint Kelly   1
DARPA/ISTO
1400 Wilson Boulevarc
Arlington, VA 22209-2389

Dr Jacob Schwartz   1
DARPA/ISTO
1400 Wilson Boulevarc
Arlington, VA 22209-2389

Dr Craig I. Fields   1
DARPA/ISTO
1400 Wilson Boulevarc
Arlington, VA 22209-2389

Lt C. Robert Simpscn   1
DARPA/ISTO
1400 Wilson Blvd
Arlington, VA 22209-2389

Dr. Allen Sears   1
DARPA/ISTO
1400 Wilson Boulevarc
Arlington, VA 22209-2389

Stephen Squires   1
DARPA/ISTO
1400 Wilson Boulevarc
Arlington, VA 22209-2389

John N. Entzminger, Director   1
DARPA/TTO
1400 wilson Boulevarc
Arlington, VA 22209-2389

Lt Ccl Russ Frew   1
DARPA/ISTO
1400 wilson Boulevarc
Arlington, VA 22209-2389

Dr Jeffrey L. Dawson   1
Digital Equipment Corporation
Artificial Intelligence Technology Group
77 Reed Road (HL02-3/MJ6)
Hudson, MA 01749

Francis S. Lynch   1
Digital Equipment Corporation
Intelligent Systems Technology Group
77 Reed Road (HL02-3/C10)
Hudson, MA 01749

Mr William Alford   1
HQ DMA
Building #56
U.S. Naval Observatory
Washington, D.C. 2C3C5

Dr Donald W. Loveland   1
Duke University
Computer Science Department
Durham, NC 27706

Dr Perry W. Thorndyke   1
FMC Corporation
Central Engineering Laboratories
1205 Coleman Avenue, Box 580
Santa Clara, CA 95C52

Dr Piero P. Bonissone   1
General Electric Company
Corporate Research and Development
1 River Road, Building 37-567
Schenectady, NY 12345

J. David McGonagle   1
General Electric Corporation R&D
Building KW, Room C619
P.O. Box 8
Schenectady, NY 12301

Jim Kornell   1
General Research Corporation
P. O. Box 6770
Santa Barbara, CA 93160

Dr William J. Frawley   1
GTE Laboratories, Inc.
Fundamental Research Laboratory
40 Sylvan Road
Waltham, MA 02254

Thomas E. Cheatham   1
Harvard University
Aiken Computation Laboratory
33 Oxford Street
Cambridge, MA 02138

John R. Beane   1
Honeywell, Inc.
Systems & Research Center MN17-2346
2600 Ridgway Parkway NE
Minneapolis, MN 55413

Robert C. Schrag   1
Honeywell, Inc.
Systems & Research Center MN17-2346
2600 Ridgway Parkway NE
Minneapolis, MN 55413

Dr. Philip Klahr   1
Inference Corporation
5300 West Century Boulevard
Los Angeles, CA 90045

Dr Paul Morris   1
IntelliCorp
1975 El Camino Real West
Mountain View, CA 94040-2216

Dr Thomas P. Kehler   1
IntelliCorp
1975 El Camino Real West
Mountain View, CA 94040-2216

Ralph F. Kromer   1
IntelliCorp
1975 El Camino Real West
Mountain View, CA 94040-2216

Dr John C. Kunz   1
IntelliCorp
1975 El Camino Real West
Mountain View, CA 94040-2216

Mike Williams   1
IntelliCorp
1975 El Camino Real West
Mountain View, CA 94040-2216

Capt Rodney G. Nuss   1
JSTPS/JPY
Building 500, Room 306
Offutt AFB, NE 68113

Dr Gerard T. Capraro   1
Kaman Sciences Corporation
258 Genesee Street
Utica, NY 13502

Dr Cordell Green   1
Kestrel Institute
1801 Page Mill Road
Palo Alto, CA 94304

Dr. John Lemmer   1
Knowledge Systems Concepts, Inc.
225 North Washington Street
P. O. Box 508
Rome, NY 13440

Dr Jerry Plante   1
Knowledge Systems Concepts, Inc.
225 North Washington Street
P. O. Box 508
Rome, NY 13440

Dr Christine A. Montgomery   1
Logicon, Inc.
Operating Systems Division
21031 Ventura Boulevard
Woodland Hills, CA 91364

Richard H. Hill   1
Microelectronics and Computer Technology Corp.
Echelon Building #1, Suite 200
9430 Research Boulevard
Austin, TX 78759

Richard L. Martin   1
Software Engineering Institute
Carnegie-Mellon University
Pittsburgh, PA 15213

Dr Randall Davis   1
MIT Artificial Intelligence Laboratory
Room NE43-801A
545 Technology Square
Cambridge, MA 02139-1986

Dr Charles Rich   1
MIT Artificial Intelligence Laboratory
Room NE43-839
545 Technology Square
Cambridge, MA 02139-1986

Dr Patrick Winston   1
MIT Artificial Intelligence Laboratory
Room NE43-816
545 Technology Square
Cambridge, MA 02139-1986

Dr Ramesh S. Patil   1
MIT Laboratory for Computer Science
Room NE43-316
545 Technology Square
Cambridge, MA 02139-1986

Dr Peter Szolovits   1
MIT Laboratory for Computer Science
Clinical Decision Making Group
545 Technology Square
Cambridge, MA 02139-1986

Dr J.A. Robinson   1
University Professor
Syracuse University
Syracuse, N.Y. 13244

Dr Richard H. Brown   1
The MITRE Corporation
P.O. Box 208
Bedford, MA 01730

Edward L. Lafferty   1
The MITRE Corporation
Mail Stop A350
Burlington Road
Bedford, MA 01730

Dr John W. Benoit   1
The MITRE Corporation
Westgate Research Park
1820 Dolly Madison Boulevard
McLean, VA 22102

Peter R. Bonasso   1
The MITRE Corporation
Mail Stop W418
1820 Dolly Madison Boulevard
McLean, VA 22102

Norman S. Glick   1
National Security Agency/T303
Ft George G. Meade, MD 20755-6000

Charles H. M. Saylor, P.E.   1
Niagara Mohawk Power Corporation
Research & Development (C-3)
300 Erie Blvd., West
Syracuse, New York 13202

Dr Richard Platek, President   1
Odyssey Research Associates, Inc.
609 West Clinton Street
Ithaca, NY 14850

Dr Robert B. Grafton   1
Office of Naval Research
Code 433
800 North Quincy Street
Arlington, VA 22217

DL-21

Alan L. Meyrowitz   1
Office of Naval Research
Code 433
800 North Quincy Street
Arlington, VA 22217

Dr B. Chandrasekaran   1
Ohio State University
Department of Computer and Information Sciences
2036 Neil Avenue
Columbus, OH 43210

Dr John Josephson   1
Ohio State University
Department of Computer and Information Sciences
2036 Neil Avenue
Columbus, OH 43210

Dr Michael J. Zoracki   1
FAR Technology Park
220 Seneca Turnpike
New Hartford, NY 13413

Dr Leo Young   1
Director for Research and Laboratory Management
Office of the Undersecretary of Defense for R&E
The Pentagon
Washington, D.C. 20301

Dr Jude E. Franklin   1
Planning Research Corporation
Research and Development Technology Division
1500 Planning Research Drive
McLean, VA 22102

Dr Fred Diamond   1
Chief Scientist
RADC/CA
Griffiss AFB, NY 13441-5700

Data & Analysis Center for Software   1
RADC/COED
Griffiss AFB, NY 13441-5700

John Parker   1
RADC/IRAA
Griffiss AFB, NY 13441-5700

John Feldman   1
RADC/IRAE
Griffiss AFB, NY 13441-5700

Edward Kobesky   1
RADC/IRAP
Griffiss AFB, NY 13441-5700

Andrew Hall   1
RADC/IRDE
Griffiss AFB, NY 13441-5700

Robert Ruberti   1
RADC/COES
Griffiss AFB, NY 13441-5700

Mr Yale Smith   2
RADC/CO
Griffiss AFB, NY 13441-5700

Mr. Anthony FR. Snyder   2
RADC/CO
Griffiss AFB, NY 13441-5700

Anthony Spina   1
RADC/IRDP
Griffiss AFB, NY 13441-5700

Dan Ventimiglia   1
RADC/IRDP
Griffiss AFB, NY 13441-5700

Lt Mike Richards   1
RADC/IRRE
Griffiss AFB, NY 13441-5700

R. Schneible   1
RADC/OCSA
Griffiss AFB, NY 13441-5700

Vincent Vannicola   1
RADC/OCTS
Griffiss AFB, NY 13441-5700

Anthony Coppola   1
RADC/RBET
Griffiss AFB, NY 13441-5700

Dale W. Richards   1
RADC/RBET
Griffiss AFB, NY 13441-5700

Sanjai Narain   1
The Rand Corporation
Information Sciences Department
1700 Main Street
Santa Monica, CA 9C4C6

Dr Harvey Rhody   1
RIT Research Corporation
75 Hightower Road
Rochester, New York 14623

Dr Casimir A. Kulikowski   1
Rutgers University
Department of Computer Science
Hill Center, Busch Campus
New Brunswick, NJ C89J3

Dr Tom Mitchell   1
Rutgers University
Department of Computer Science
Hill Center, Busch Campus
New Brunswick, NJ C8903

Dr Sholom Weiss   1
Rutgers University
Department of Computer Science
Hill Center, Busch Campus
New Brunswick, NJ 08903

Dr Jay M. Tenenbaum   1
Schlumberger Palo Alto Res Center
3340 Hillview Ave.
Palo Alto, CA 94304

Dr David Barstow   1
Schlumberger-Doll Research Center
Old Quarry Road
Ridgefield, CT 06877-4108

Dr Elaine Kant   1
Schlumberger-Doll Research Center
Old Quarry Road
Ridgefield, CT 06877-4108

Bob Young   1
Schlumberger-Doll Research Center
Old Quarry Road
Ridgefield, CT 06877-4108

Raymond E. Sandborgh   1
Sperry Corporation
Knowledge Systems Center
3001 Metro Drive, Suite 223
Bloomington, MN 55420

Dr. James E. Carrig   1
Sperry Corp - CTC
12010 Sunrise Valley Drive
Reston, VA 22091

Dr Stuart L. Brodsky   1
Sperry Corp - CTC
12010 Sunrise Valley Drive
Reston, VA 22091

Dr Thomas D. Garvey   1
SRI International
Artificial Intelligence Center
333 Ravenswood Avenue
Menlo Park, CA 94025-3493

Dr John Lawrence   1
SRI International
Artificial Intelligence Center
333 Ravenswood Avenue
Menlo Park, CA 94025-3493

Dr Stan Rosenschein   1
SRI International
Artificial Intelligence Center
333 Ravenswood Avenue
Menlo Park, CA 94025-3493

Dr William M. Tyson   1
SRI International
Artificial Intelligence Center
333 Ravenswood Avenue
Menlo Park, CA 94025-3493

Dr Richard J. Waldinger   1
SRI International
Artificial Intelligence Center
333 Ravewnswood Avenue
Menlo Park, CA 94025-3493

Dr Mark S. Moriconi   1
SRI International
Computer Science Laboratory
333 Ravenswood Avenue
Menlo Park, CA 94025-3493

Dr Jan Aikens   1
Computer Science Department
Stanford University
Margaret Jacks Hall
Stanford, CA 94305

Dr Zohar Manna   1
Computer Science Department
Stanford University
Margaret Jacks Hall
Stanford, CA 94305

Dr Nils J. Nilsson, Chairman   1
Computer Science Department
Stanford University
Margaret Jacks Hall
Stanford, CA 94305

Richard W. Weyhrauch   1
Computer Science Department
Stanford University
Margaret Jacks Hall
Stanford, CA 94305

Dr David C. Luckham   1
Stanford University
Computer Systems Laboratory
Stanford, CA 94305

Dr Nelleke Aiello   1
Stanford University
Heuristic Programming Project
701 Welch Road, Building C
Palo Alto, CA 94304

Dr Harold Brown   1
Stanford University
Heuristic Programming Project

701 Welch Road, Building C
Palo Alto, CA 94304

Dr Bruce G. Buchanan   1
Stanford University
Heuristic Programming Project
701 Welch Road, Building C
Palo Alto, CA 94304

Dr Robert Engelmore   1
Stanford University
Heuristic Programming Project
701 Welch Road, Building C
Palo Alto, CA 94304

Dr Larry Fagan   1
Stanford University
Heuristic Programming Project
701 Welch Road, Building C
Palo Alto, CA 94304

Dr Edward A. Feigenbaum   1
Stanford University
Heuristic Programming Project
701 Welch Road, Building C
Palo Alto, CA 94304

Dr Michael R. Genesereth   1
Stanford University
Heuristic Programming Project
701 Welch Road, Building C
Palo Alto, CA 94304

Dr Barbara Hayes-Roth   1
Stanford University
Heuristic Programming Project
701 Welch Road, Building C
Palo Alto, CA 94304

Dr H. Penny Nii   1
Stanford University
Heuristic Programming Project
701 Welch Road, Building C
Palo Alto, CA 94304

Thomas C. Rindfleisch, Director   1
Stanford University
Heuristic Programming Project
701 Welch Road, Building C
Palo Alto, CA 94304

Dr Edward H. Shortliffe   1
Stanford University Medical Center
Division of General Internal Medicine
Medical Computer Science TC-135
Stanford, CA 94305

Bruce Delagi   1
Stanford/DEC
Stanford, CA 94305

Dik Gabriel   1
Stanford/LUCID
Stanford, CA 94305

DL-28

Dr Michal Cutler   1
SUNY/Binghamton
Computer Science Department
Watson School of Engineering
Binghamton, NY 13901

Dr Leslie C. Lander   1
SUNY/Binghamton
Computer Science Department
Watson School of Engineering
Binghamton, NY 13901

Dr Stuart C. Shapiro   1
SUNY/Buffalo
Computer Science Department
226 Bell Hall
Buffalo, NY 14260

Dr Sargur N. Srihari   1
SUNY/Buffalo
Computer Science Department
226 Bell Hall
Buffalo, NY 14260

Dr Richard O. Duda   1
Syntelligence
1000 Hamlin Court
Sunnyvale, CA 94088

Dr Peter E. Hart   1
Syntelligence
1000 Hamlin Court
Sunnyvale, CA 94088

Dr Bruce Berra   1
Syracuse University
Dept of Electrical and Computer Engineering
111 Link Hall
Syracuse, NY 13210

Dr Kenneth A. Bowen   1
Syracuse University
School of Computer and Information Science
313 Link Hall
Syracuse, NY 13210

Dr J. Alan Robinson   1
Syracuse University
School of Computer and Information Sci...
313 Link Hall
Syracuse, NY 13210

Dr Bradley J. Strait   1
Syracuse University
Managing Director, CASE Center
120 Hinds Hall
Syracuse, NY 13210

Miriam B. Bischoff   1
Teknowledge, Inc.
1850 Embarcadero
Palo ALto, CA 94301

Dr Frederick Hayes-Roth   1
Teknowledge, Inc.
1850 Embarcadero
Palo Alto, CA 94301

Bruce Bullock   1
Teknowledge Federal Systems
501 Marin Street, #214
Thousand Oaks, CA 91360

Gary Edwards   1
Teknowledge Federal Systems
501 Marin Street, #214
Thousand Oaks, CA 91360

Dr Brian Phillips   1
Tektronix, Inc.
Computer Research Lab
P.O. Box 500, Mail Station 50-662
Beaverton, OR 97077

Dr Roger Bate, Director   1
Texas Instruments Central Research Laboratories
Computer Science Laboratory
P.O. Box 226015, MS 238
Dallas, TX 75266

Dr Edward Riseman   1
University of Massachusetts
Computer & Information Science Department
Room A213 Lederle Graduate Research Center
Amherst, MA C1003

Dr Beverly Woolf   1
University of Massachusetts
Computer & Information Science Department
Amherst, MA C1003

Dr Timothy W. Finin   1
University of Pennsylvania
The Moore School
Department of Computer and Information Science
Philadelphia, PA 19104

Dr Harry E. Pople   1
University of Pittsburgh
Decision Systems Laboratory
1360 Scaife Hall
Pittsburgh, PA 15261

Dr James F. Allen   1
University of Rochester
Department of Computer Science
Rochester, NY 14627

Richard Pelavin   1
University of Rochester
Department of Computer Science
Rochester, NY 14627

Dr Robert M. Balzer   1
University of Southern California
Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292-6695

Dr Lee Erman   1
Teknowledge, Inc
1850 Embarcadero
Palo Alto, CA 94301

Dr Daniel G. Bobrow   1
Xerox Corporation
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, CA 94304

Dr Johan de Kleer   1
Xerox Corporation
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, CA 94304

Dr Mark Stefik   1
Xerox Corp
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, CA 94304

Dr Christopher Riesbeck   1
Yale Univ (Computer Science)
P.O. Box 2158, Yale Station
New Haven, CT 06520

Dr Elliot Soloway   1
Yale Univ (Computer Science)
P.O. Box 2158, Yale Station
New Haven, CT 06520

Dr Ruven Brooks   1
126 Hunter's Creek Road
Shelton, CT 06484

Steven Dolins   1
5647 Anita
Dallas, TX 75206

Michael Gabin   1
Texas Instruments, Inc.
P.O. Box 226015 MS 3646
Dallas, Texas 75266

Harry Barrow   1
Schlumberger Palo Alto Research
3340 Hillview Avenue
Palo Alto, CA 94304

James Baumann   1
AFIT/Engineering
Wright-Patterson AFB, Ohio 45433

Lee Baumann
Science Applications International
    Corporation   1
1710 Goodridge Drive, T-10-4
McLean, VA 22102

1
Charles Bisbee
Air Force Institute of Technology
Dept of Electrical Engineering
Wright-Patterson AFB, Ohio 45433

1   1
Allen Brown
General Electric Corporation R&D
P.O. Box 8
Bldg K1-Room 5C-8A

Mark Burstein   1
BBN Laboratories, Inc.
10 Mouton Street
Cambridge, MA 02238

Jaime Carbonell   1
Carnegie-Mellon University
Robotics Institute
Schenley Park
Pittsburgh, PA 15213

George Cefoldo   1
Texas Instruments
P.O. Box 660246 MS 3645
Dallas, Texas 75266

David Chapman   1
MIT AI Laboratory
545 Technology Square
Cambridge, MA 02139

Guy Clayton   1
McDonnell Douglas Corporation
DE422/13LDG33/LVL5/5C
P.O. Box 516
St. Louis, MO 63166

Ernest Davis   1
NYU - Courant Institute
251 Mercer Street
New York, NY 10012

Thomas Dean   1
Brown University
Computer Science Department
Providence, RI 02912

John Delaney   1
Stanford University
Heuristic Programming Project
701 Welch Road, Building C
Palo Alto, CA 94040

George Doddington   1
Texas Instruments
P.O. Box 226015
M-S 238
Dallas, Texas 75266

Rich Doerr   1
AFWAL/AAAT-1
Wright-Patterson AFB
Ohio 45433-6543

Lee Erman   1
Teknowledge, Inc.
1850 Embarcadero Road
P.O. Box 10119
Palo Alto, CA 94303

Scott Fouse   1
Teknowledge, Inc.
1850 Embarcadero Road
P.O. Box 10119
Palo Alto, CA 94303

Peter Friedland   1
NASA Ames Research Center
RIA:244-17
Moffett Field, CA 94035

Carl Friedlander   1
The BDM Corporatin
1300 North 17th Stret
Arlington, VA 22209

Dale Gaucas   1
General Electric
1 River Road
Schenectady, NY 12305

Roger Geesey   1
The BDM Corporation
1300 N. 17th Street
Arlington, VA 22209

Mike Georgeff   1
SRI International
AI Center
333 Ravenswood
Menlo Park, CA 94022

Michael Greenberg   1
University of Massachusetts
Computer & Info Science Department
Amherst, MA C1003

Jim Guffey   1
McDonnell Aircraft
717/33/5N/505
P.O. Box 516
St. Louis, MO 63166

Gregg Gunsch   1
AFWAL/AAAT-1
Wright-Patterson AFB
Ohio 45433-6543

Doug Hager   1
AFWAL/AAAT-1
Wright-Patterson AFB
Ohio 45433-6543

Kristian Hammond   1
The University of Chicago
1100 East 58th Street
Ryerson 152
Chicago, IL 60637

Bud Hammons   1
Texas Instruments, Inc.
P.O. Box 226015
Dallas, TX 75266

Karen Harbison-Moss   1
Texas Instruments, Inc.
P.O. Box 226015
Dallas, TX 75266

Patrick Harrison   1
US Naval Academy
Computer Science Department
Annapolis, MD 21402-5002

Fred Hollander   1
Texas Instruments, Inc.
P.O. Box 226015
Dallas, TX 75266

Tom Hummel   1
AFWAL/FIGL
Wright-Patterson AFB
Ohio 45433-6523

Kirby Keller   1
McDonnell Aircraft
31 3/33/5N/505
P.O. Box 516
St. Louis, MO 63166

Paul Kline   1
Texas Instruments
P.O. Box 226015, MS 238
Dallas, TX 75266

Richard Korf   1
UCLA
Computer Science Department
Los Angeles, CA 90024

Ted Kral   1
Space and Naval Warfare Systems
     Command
Code 3214A
Washington, DC

Jay Lark   1
Teknowledge, Inc.
1850 Embarcadero Road
P.O. Box 10119
Palo Alto, CA 94303

Paul Lehner   1
George Mason University
Info Technology & Engineering
4400 University Drive
Fairfax, VA 22033

Ted Linden   1
ADS
201 San Antonio Circle, Suite 286
Mountain View, CA 94040-1270

Tomas Lozano-Perez   1
MIT AI Laboratory
545 Technology Square
Cambridge, MA 02139

Bob MacGregor   1
ISI
4676 Admiralty Way
Marina Del Rey, CA 90292

Matt Ginsberg   1
Stanford University
Computer Science Department
Stanford, CA 94305

Mike McMahan   1
Texas Instruments, Inc.
P.O. Box 226015

Dallas, TX 75266

Phillip Merkel   1
BDM Corporation
1300 N. 17th Street - #950
Arlington, VA 22209

David Miller   1
Virginia Polytechnic
   and State University
Computer Science Department
Blacksburg, VA

Doug Paul   1
MIT/Lincoln Laboratory, Room B-353
Speech Systems Technical Group
P.O. Box 73
Lexington, MA 02173-0073

David Payton   1
Hughes Research Laboratories
3011 Malibu Canyon Road
Malibu, CA 90265

Lise Pfau   1
General Electric
CRD
1 River Road
Schenectady, NY 12305

Raja Rajasekaran   1
Texas Instruments
P.O. Box 226015
M-S 238
Dallas, TX 75266

Doug Rouse   1
AFWAL/FIGR
Wright-Patterson AFB
Ohio 45433

Reid Simmons   1
MIT AI Laboratory
545 Technology Square
Cambridge, MA 02139

David Smith   1
Lockheed-Georgia Co.
Dept 72-99 Zone 410
Marietta, GA 30063

David Smith   1
Stanford University
Computer Science Department
Stanford, CA 94305

Steven Smith c/o Patty Hoogson   1
Carnegie-Mellon University
Robotics Institute
Schenley Park
Pittsburgh, PA 15213

Rolf Stachowitz   1
Lockheed AI Center, C/90-06, B/30E
2100 E. St. Elmo Rd.,
Austin, TX 78744
512 448-9718

Lou Steinberg   1
Rutgers University
Department of Computer Science
New Brunswick, NJ C8903

Charles Thorpe   1
Carnegie-Mellon University
Computer Science Department
Robotics Institute
Pittsburgh, PA 15213

Richard Treitel   1
Intellicorp
1975 El Camino Road West
Mountain View, CA 94C40-2216

David Tsseng   1
Hughes Research Laboratories
3011 Malibu Canyon Road
Malibu, CA 90265

Ron VanDerWeert   1
AFWAL/FIGR
Wright-Patterson AFB
Ohio 45433-6523

Jan D. Wald   1
Honeywell Systems and Research Center
3660 Technology Drive
Minneapolis, MN 55418

Raj Wall
Texas Instruments, Inc.   1
Artificial Intelligence Laboratory
P.O. Box 655474, M/S 238
Dallas, TX 75265

Cliff Weinstein   1
MIT/Lincoln Laboratory, Room B-335
Speech Systems Technical Group
P.O. Box 73
Lexington, MA 02173-0073

David Wilkins   1
SRI International
AI Center
333 Ravenswood
Menlo Park, CA 94025

Ben Wise   1
Dartmouth College
Thayer School of Engineering
Hanover, NH 03755

Lockheed Austin Division   1
ATTN: Dr. Bill Wedlake
ALBM Program Manager
6800 Burleson Road, O/T1-90 B/30E
Austin, TX 78744

Advanced Decision Systems   1
ATTN: Mr. Jim Marsh
ALBM Program Manager
201 San Antonio Circle Suite 286
Mountain View, CA 94040-1289

Bolt Beranek Newman Laboratories   1
ATTN: Mr. Fred Kulik
ALBM Program Manager
10 Moulton Street
Cambridge, MA 02238

Intellicorp, Inc.   1
ATTN: Mr. John Gaiser
ALBM Program Manager
1975 El Camino Real West
Mountain View, CA 94040-2216

Titan Systems, Inc.   1
ATTN: Mr. Leon Bloom
ALBM Program Manager
P.O. Box 2123
Chatsworth, CA 91311

Martin Marietta   1
ATTN: Dr. Bob Douglas
ALV Program Manager
P.O.Box 179
Denver, CO 9C201

Hughes AI Center   1
ATTN: Dr. David Tseng
ALV Program Manager
23901 Calabasas Road
Calabasas, CA 91302

Erim Company   1
ATTN: Dr. Robert Franklin
ALV Program Manager
P.O.Box 8618
Ann Arbor, MI 48107

Advanced Decision Systems   1
ATTN: Dr. Ted Linden
ALV Program Manager
201 San Antonio Circle Suite 286
Mountain View, CA 94040-1289

Texas Instruments   1
ATTN: Mr. Steve Olson MS 3646
FRESH Program Manager
P.O. Box 660246
Dallas, TX 75266

Bolt Beranek Newman Laboratories   1
ATTN: Dr. Ed. Walker
CASES Program Manager
10 Moulton St.
Cambridge, MA 02238

Honeywell Systems & Research Cntr   1
ATTN: Dr. Jan Wald
CPS Planner Program Manager
3660 Technology Drive
Minneapolis, MN 55418

Lockheed Georgia Compmay   1
ATTN: Mr. J Barnette 072-64 Z410
PA Program Manager
86 Sc. Cobb Drive
Marietta, GA 30063

```
Teknowledge Federal Systems                              1
ATTN: Mr. Allen Smith
FA Program Manager
501 Marin Street   Suite 114
Thousand Oaks, CA   91360

Search Technology                                        1
Mr. N Geddes, PA Program Manager
Bldg. 5550A   Suite 500
Peach Tree Parkway
Norcross, GA   30092

Loral Systems                                            1
ATTN: Mr. Dan Davidson

FA Program Manager
1210 Massillon Road
Akron, OH   44315

Titan Systems                                            1
ATTN: Mr. Bill Williams
FA Program Manager
9191 Towne Centre Drive
San Diego, CA   92122

McDonnell Aircraft Co.                                   1
ATTN: Dr. Jack D. Corrigan
FA Program Manager
Box 516
St. Louis, MO   63166

Texas Instruments                                        1
ATTN: Mr. George Cefoldo   MS 3645
FA Program Manager
P.O.Box 660246
Dallas, TX   75266

Texas Instruments                                        1
ATTN: Ms. Joyce Graham   MS 3645
RAV Program Manager
P.O.Box 660246
Dallas, TX   75266

Advanced Decision Systems                                1
ATTN: Mr. Bob Drazovich
ADPIES Program Manager
201 San Antonio Circle   Suite 286
Mountain View, CA   94040-1289
```

Hughes Aircraft Company - EDSG   1
ATTN: Mr. Julius Bogdanowicz
SCORPIUS PM, B-E52 MS D213
P.O.Box 902
El Segundo, CA   90245

NRJ Inc.   1
ATTN: Mr. Bob Ready
ADRIES Program Manager
10455 White Granite Dr   Suite 20C
Oakton, VA   22124

SAIC   1
ATTN: Dr. Richard Kruger
ADRIES Program Manager
5151 E. Broadway   Suite 900
Tucson, AZ   85711

The Analytic Sciences Corp. (TASC)   1
ATTN: Dr. Hal Jones
ADRIES Program Manager
55 Walkers Brook Drive
Reading, MA   01867

Lockheed Georgia Company   1
ATTN: Mr. J Barnette D72-64 Z410
SW Program Manager
86 So. Cobb Drive
Marietta, GA   30063

Teknowledge Federal Systems   1
ATTN: Mr. Allen Smith
SW Program Manager
501 Marin Street   Suite 114
Thousand Oaks, CA   91360

Titan Systems   1
ATTN: Mr. Bill Williams
SW Program Manager
9191 Towne Centre Drive
San Diego, CA   92122

Honeywell Inc.   1
ATTN: Mr. Richard Lahn
SW Program Manager
3660 Technology Drive
Minneapolis, MN   55418

Boeing Military Aircraft Company   1
ATTN: Mr. Bill Podlena MS K8C-12
SW Program Manager
3801 S. Oliver
Wichita, KA   67210

```
General Electric                                              1
ATTN: Dr. Gerry Albers
        PA Program Manager
P.O. Box 2500
Daytona Beach, FL   32015


FMC Central Engineering Labs.                                1
ATTN: Dr. Perry Thorndyke
        PA Program Manager
Box 580
Santa Clara, CA   95052


Texas Instruments                                            1
ATTN:  Ms. Joyce Graham  MS 3645
        RAV Program Manager
P.O.Box 660246
Dallas, TX   75266


Hughs Aircraft Co., Missle Systems                           1
ATTN: Dr. John T. Hall B265,MS X47
        SW Program Manager
8433 Fallbrook Avenue
Canoga Park, CA   91304-0445


United Technologies Corp., ASD                               1
ATTN: Mr. Lee Best
        SW Program Manager
10180 Telesis Court
San Diego, CA   91221-2719


General Dynamics Convair                                     1
ATTN: Mr. Robert Borris MZ 42-6210

        SW Program Manager
5001 Kearny Villa
San Diego, CA   92123


SRI International                                            1
ATTN: Dr. Franklin F. Kuo MS EL290
        SW Program Manager
337 Ravenswood
Menlo Park, CA   94025


Loral Systems                                                1
ATTN: Mr. Dale Barcin
        SW Program Manager
1210 Massillon Road
Akron, OH   44315


JAYCOR                                                       1
ATTN: Ms. Nancy Pruitt
        SW Program Manager
1608 Spring Hill Road
Vienna, VA   22180-2270
```

DL-46

Northrop, ElectroMechanical Div.   1
ATTN: Mr. Don Longmire MS 7200/Y34
      SW Program Manager
500 E. Orangethrope Avenue
Anaheim, CA 92801

Advanced Decision Systems   1
ATTN: Mr. Jim Marsh
      SW Program Manager
201 San Antonio Circle Suite 286
Mountain View, CA 94040-1289

EDM Corp.   1
ATTN: Mr. Phil Merkel
      SW Program Manager
1300 N. 17th Street - # 950
Arlington, VA 22209

Texas Instruments   1
ATTN: Mr. Bill Sterns MS 3648
      SW Program Manager
P.O.Box 660246
Dallas, TX 75266

AVCO Research Laboraties   1
ATTN: Mr. Jim Anapol
      SW Program Manager
2385 Revere Beach Parkway
Everett, MA 02149

Bolt Beranek Newman Laboratories   1
ATTN: Dr. Sheldon Barron
      SW Program Manager
10 Moulton Street
Cambridge, MA 02238

VICOM   1
ATTN: Dr. William Pratt
      SW Program Manager
2520 Junction Avenue
San Jose, CA 95134-1989

AMERINEX Corp   1
ATTN: Mr. Robert Sundies
      SW Program Manager
Park 80 West, Plaza 2
Saddlebrook, NJ 07662

Dr Ed Taylor   1
TRW Defense & Space Group
Building R2/2094
One Space Park
Redondo Beach, CA 90278

Dr Fred Petry   1
Tulane University
Department of Computer Science
New Orleans, LA 70118

Renee Elio   1
University of Alberta
Department of Computing Science
Edmorton, Alberta T6G 2H1

Jim Hollan   1
University of California, San Diego
Institute for Cognitive Science
C015
La Jolla, CA 92093

Creighton Lewis   1
University of Colorado -- Boulder
Department of Computer Science
Boulder, CO 80309

Dr Paul Cohen   1
University of Massachusetts
Computer & Information Science Department
Amherst, MA 01003

Dr W. Bruce Croft   1
University of Massachusetts
Computer & Information Science Department
Amherst, MA 01003

Dr Victor R. Lesser   1
University of Massachusetts
Computer & Information Science Department
Amherst, MA 01003

Martin Marietta Elect&Missiles Gr   1
ATTN: Mr. Wade Pemberton MP 40C
       SW Program Manager
P.O.Box 5837
Orlando, FL   32855

Martin Marietta Space Systems Co.   1
ATTN: Mr. Richard Luhrs MS 0427
       SW Program Manager
P.O.Box 179
Denver, CO   80201

Northrop, ElectroMechanical Div.   1
ATTN: Norman Huffnagele MS Y34W
       Manager Advanced Concepts
500 E. Orangethrope Avenue
Anaheim, CA   92801

System Planning Corp.   1
ATTN: Mr. William H. Harris
       SW Program Manager
1500 Wilson Blvd.
Arlington, VA   22209-2454

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control, Communications and Intelligence (C³I) activities. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C³I systems. The areas of technical competence include communications, command and control, battle management, information processing, surveillance sensors, intelligence data collection and handling, solid state sciences, electromagnetics, and propagation, and electronic, maintainability, and compatibility.