October 1986

Report No. STAN-CS-86-1136
Also numbered KSL-86-69

An Experiment in Knowledge-based Signal Understanding Using Parallel Architectures

by

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

Department of Computer Science

Stanford University
Stanford, CA 94305

Knowledge Systems Laboratory
Report No. KSL 86-69

October 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

This research was supported by DARPA Contract F30602-85-C-0012, NASA Ames Contract NCC 2-220-S1, and Boeing Contract W26687.5. Eric Schoen was supported by a fellowship from NL Industries. Bruce Delagi is currently a visiting research scientist at Stanford from Digital Equipment Corporation.

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.

Table of Contents

List of Figures

List of Tables

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 l-l. 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? Figure l-l 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.

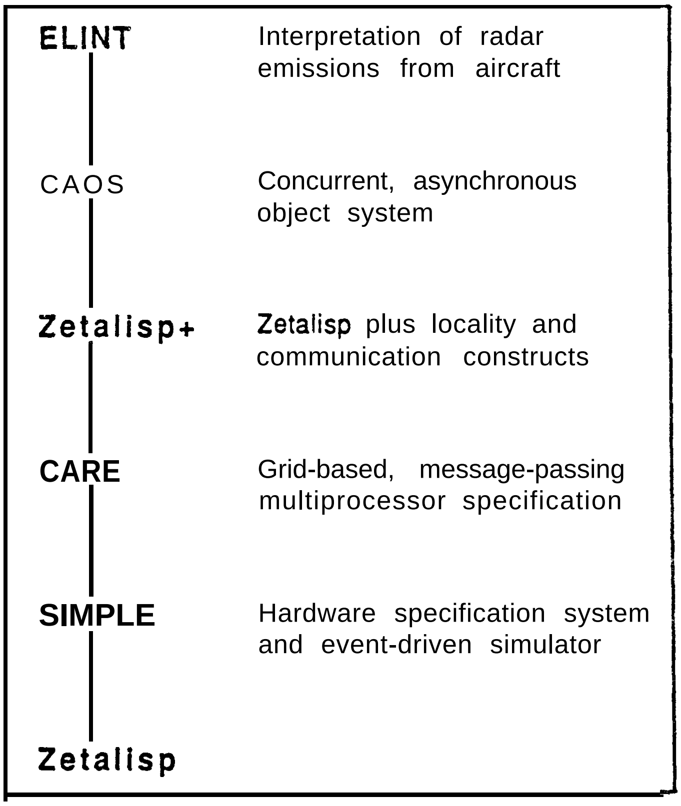

Figure 1-1: The software component hierarchy of the experiment.

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.*

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. ELTNT maintains the track and activity histories of the clusters

2.1. ELINT's Inputs

The inputs to the ELTNT 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-l: Elint observation record.

Field	Contents
Observation-Time	**An integer rime-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 obsavation
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 obsavation
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 Al 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 current 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 [I] 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? 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.

'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 **multipsst-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-l

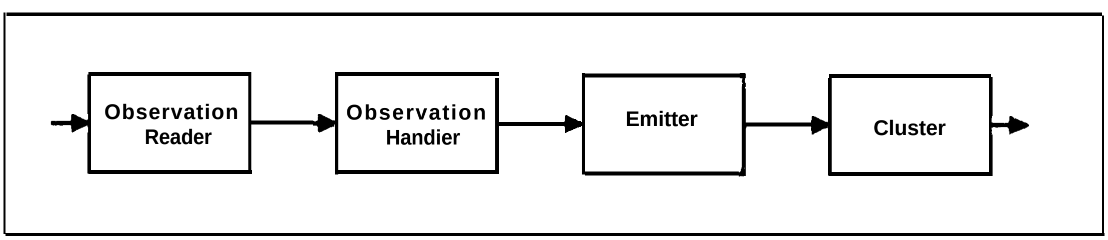

Figure 4-1: The basic ELINT a :nt processing pipeline,

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 emi tter-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 ELTNT 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 ***lastfobserved*** 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]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+l.

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?*

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

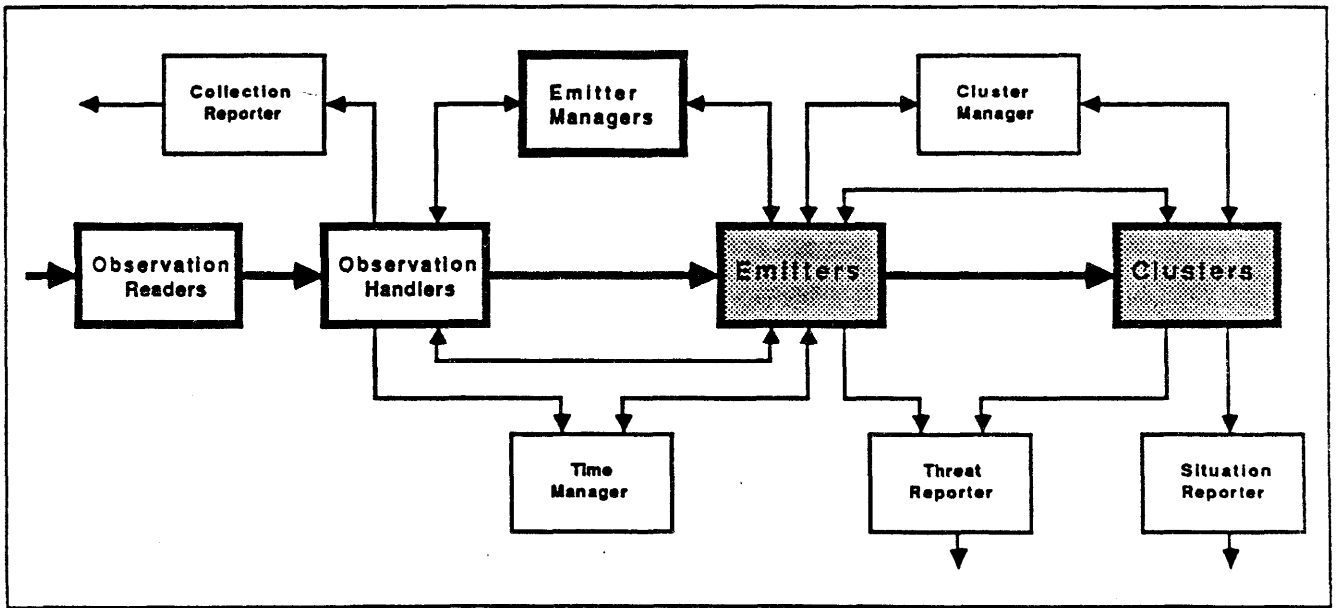

Figure 4-2: The overall ELINT agent communication organization.

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.

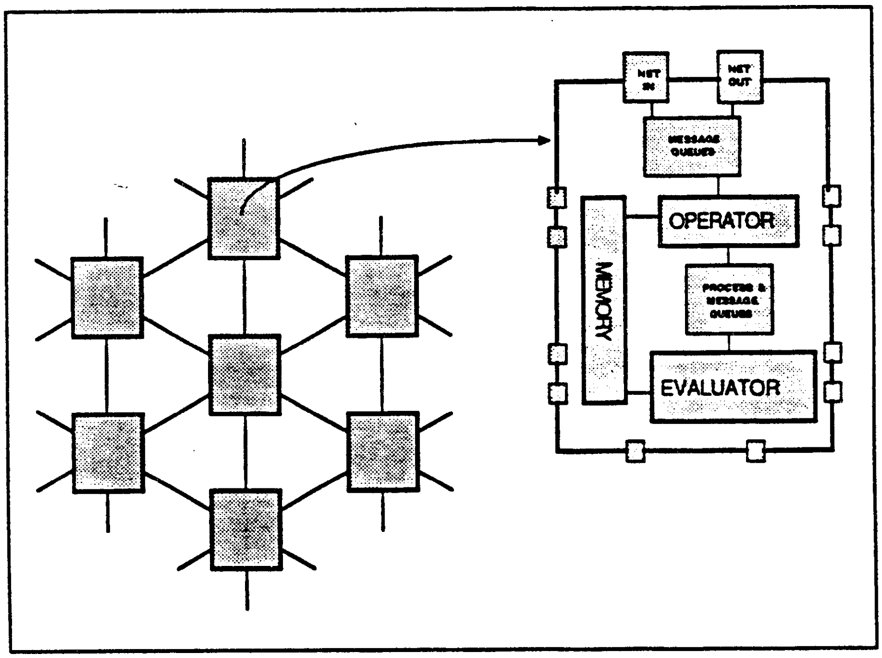

Figure 5-1: A hexagonally connected CARE grid.

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 withir 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].

6.13. 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 loo-processor realization of a particular architecture perform ten times better than a lo-processor realization of the same architecture? Does ıt 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 lo-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 t:s 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 ELXNT 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-l: 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 high locally due to poor load balancing so that some subset of the emitters are overloaded.

The Cl' 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.

Fix&: 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-l 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 site 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.**	
c c		**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 data

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

Control strategy	Grid size		
	4	16	36
NC		>16118msg.	
c c		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 speedup 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

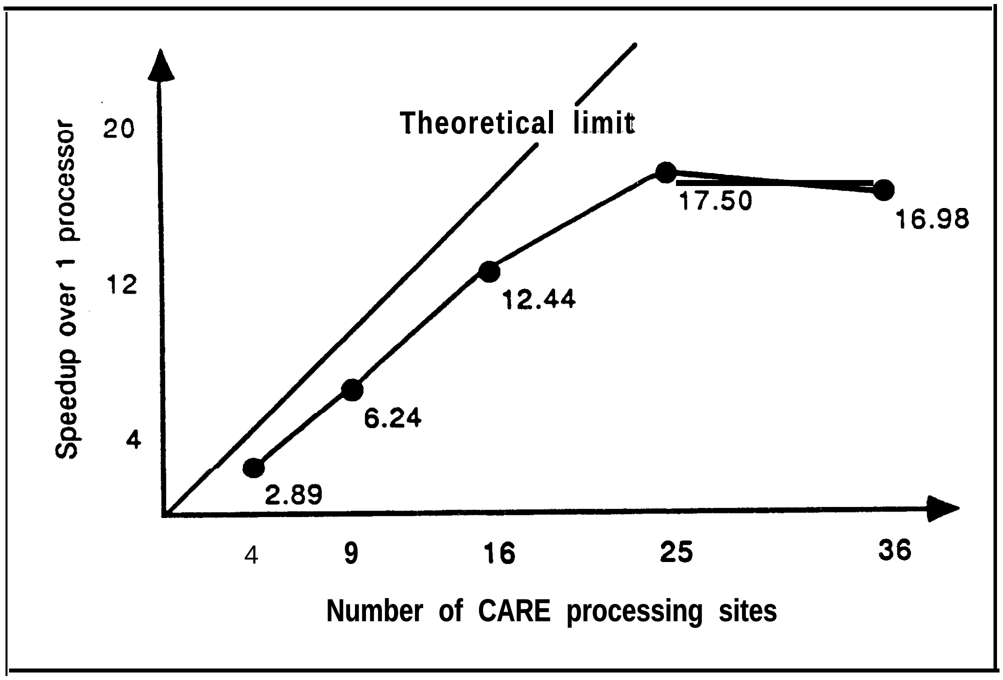

Figure 6-1: The relative speedup of ELINT executions on various size CARE grids.

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	l-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. Plow 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.