Appendix: CAOS: An Early Experiment

In this section we describe an early end-to-end application experiment, performed in 1984-85. The application is a knowledge-based ELINT (ELectronics INTelligence) system for interpreting processed, passively acquired radar emissions from aircraft. The ELINT application was implemented in CAOS, an experimental concurrent, asynchronous object-oriented framework built on Zetalisp [Weinreb 81]. The CAOS framework, in turn, relied on the services provided by the underlying multiprocessor machine system. Figure 1 illustrates the relationship between the various software components of the experiment.

Figure 1 The software component hierarchy of the experiment (* p.3)

The ELINT-CAOS-CARE experiment investigated both qualitative and quantitative aspects of the performance of the overall system. The machine model uses dynamic, cut-through (as 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, and often did in the experiments, 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 multicomputer grid. A third objective was to gain some understanding of the difficulties in coding and debugging a reasonably complex knowledge-based system using a concurrent object-oriented programming system and executing on a multiple address space, message-passing multicomputer system.

1 The CAOS Programming Model

CAOS is a programming system which supports the encoding and the execution of knowledge-based systems on a multicomputer. It represents an early attempt to bridge the gap between the application specification and the multiprocessor system programming primitives. The design of CAOS was predicated on the belief that many highly parallel architectures (for example, hundreds of processors) will emphasize communication between processor-memory pairs rather than uniformly shared memory. When CAOS was designed, we assumed that such an architecture would be communication limited and favor relatively coarse-grained problem decomposition with little synchronization between processors. (Since then our experimental results have lead us to question some of these assumptions — see Section 4.) CAOS was intended for use in soft real-time, data interpretation applications such as process monitoring and radar and sonar signal interpretation (see, for example, [Nii 78]). CAOS is based on an object-oriented programming paradigm, and it drew many of its ideas from the Flavors system [Flavors] and the Actors paradigm [Hewitt 73].

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.\textsuperscript{1}

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. The agent paradigm provides a form of consistency maintenance since processes execute only in response to messages and execute only in the context of the receiving agent. In particular, there are no remote side effects.

CAOS was designed to express parallelism at a relatively coarse grain-size (several or more milliseconds execution time per process). For example, in the ELINT experiment, the message handlers (more specifically, the methods) which implement the message responses are written as Lisp procedures averaging about one hundred lines of Lisp code each. CAOS supports no mechanism for finer-grained concurrency such as within the execution of agent processes, but neither does it rule it out.

1.1 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 process replication. Pipelining is most appropriate for representing the flow of information between levels of abstraction in an interpretation system. Process replication provides means by which the interpretation system can cope with high data rates.

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

\textsuperscript{1}The active processes in a CAOS agent are not scheduled preemptively. Instead, an executing agent process either runs to completion or until it is "blocked" awaiting some remote service.
its immediate predecessor, but also on the results of more distant predecessors, or worse, some distant successor (for example, 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 usually achieves less than an \( n \)-times increase in throughput.

**Process Replication**

Concurrency gained through process 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 experiments show 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 a knowledge-based 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 1.2 where we describe the CAOS agent types for the ELINT application.

### 1.2 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 [Shoen 86].

**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 processor 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 agent's clock ticks, the set of methods associated with that clock are queued for execution.

Critical-Methods: This attribute declares certain sets of methods as being mutually "critical regions" for their owning agents. Each such set of critical methods has an associated lock. Before an owning 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.

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 instantiated (and possibly deleted) during program execution. For programmer convenience, we allow code in agent message handlers and default values of local-variables to reference 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 site location specification. The remote-address of the newly-created agent instance is returned. The remote-address of an agent includes the processor address where the agent resides and a pointer to the agent in the address space of that site. A dynamically created agent may not be referenced symbolically, however, its remote-address may be exchanged freely.
Communications Between Agents

Agents communicate with each other by asynchronously exchanging messages. CAOS and the supporting machine model does not guarantee when messages reach their destinations. Due to excessive message traffic messages may be significantly delayed during routing. In particular, since there is no handshaking or fixed message sending cycle, message arrival order may be different than message sending order. It is the responsibility of the application program to detect and recover from such delayed and/or out-of-order 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 data structure called a future which represents a "promise" for an eventual value.2

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 machine system 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) 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 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.

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. A typical timeout-action is setting the future's value to a default value.

---

2Futures are also used in Multilisp[]. The HEP Supercomputer[] implemented a simple version of futures as a process synchronization mechanism.
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 the machine model.

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.

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 the machine model. 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 [Shoen 86].

The implementation of CAOS described in this report was written in Zetalisp [Weinreb 81] and the machine model's primitive operators using Zetalisp's object-oriented programming tool, Flavors.

Each processor 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 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 referred 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 communication 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 processor site's scheduler in the scheduling of these user processes.

---

3Neither CAOS nor the machine model supported a "predicated multicast" mode wherein messages would be sent to all agents satisfying a particular predicate. Messages could only be multicast to a fully-specified list of agents. Receiving agents could, of course, apply arbitrary predicates to the message in order to determine their consequent action.
2. ELINT in CAOS

We describe now the agent types and their organization for the ELINT application (described in Chapter 3) as implemented in the CAOS. This implementation illustrates some of the benefits and some of the drawbacks of CAOS. As discussed earlier, ELINT is a knowledge-based 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 2.

Figure 2. The basic ELINT agent processing pipeline (* p. 15)

2.1 ELINT Agent Types

The ELINT agent types described here are those used by the CT control strategy version of ELINT in CAOS (see Section 3).

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 Chapter 3. 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 Emitter-managers coordinate emitter instantiation activity between observation-handlers. 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 processor 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 knowledge-based 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 1.1). Consider the following strategies that the observation-handler agents could use to create new emitter agents:

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

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.

The handlers could rely on a single emitter-manager agent to create all emitters. While this approach is safe from a consistency standpoint, when it was tried the single emitter-manager agent proved to be a processing bottleneck.

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.

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 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.\(^4\)

\(^4\)The algorithm simply computes the source identifier modulo the number of emitter-managers and maps that number to a particular manager.
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 Chapter 2) 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 easily 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 chose to implement ELINT using only a single cluster-manager.

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 existing and it will delete itself and all of its relational links from ELINT's situation board.\(^5\)

In an asynchronous message passing system it is difficult for an emitter 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.\(^6\) This solution was rejected for the reported implementation of ELINT because of a perceived excessive message overhead.\(^7\)

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

---

\(^5\)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 \(t + 1\).

\(^6\)Since each input observation stream is in observation-time sequential order, each observation-handler eventually knows when such a time boundary is crossed.

\(^7\)This overhead was more perceived than actual. The LAMINA implementation of ELINT uses such "end-of-observation-time-interval" messages, and the results indicate that the associated cost is not excessive (see Section 7).
used during application program debugging to asynchronously output debugging traces in a manner that minimally impacts system timing dependencies.

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 3.

Evaluation

The multiprocessor machine models used for the ELINT-CAOS experiment were of square grids of between four and forty-nine processing sites interconnected via a dedicated hexagonal communications network. That is, each site, excluding those at the edges of the grid, is connected to six of its eight nearest neighbors. The network uses cut-through (as opposed to store and forward) dynamic routing.

As shown in Figure 4, each processing 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 can establish a connection with a net-output at any site, and all such connections at a site can be simultaneously active.

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.

A more detailed description of the machine models and the simulation system were given in Chapter 2.

3.1 Evaluation of 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.
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 succeeded in meeting this goal for CAOS.8 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.

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 settings of machine parameters used in the experiment, machine-level messages were exchanged in about 2 to 3 milliseconds, while CAOS messages required about 30 milliseconds. It is this cost which forced us to decompose applications coarsely, since more fine-grained decompositions would have inevitably required more message traffic.

We conclude that CAOS does not make efficient use of the underlying machine.

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, or only just as well? 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 high data rates is by increasing the number of processors.

We believe CAOS scales well with respect to the number of available processors, at least up to a few tens of processors. The potential limiting factors to its scaling are increased software contention, such as the inter-pipeline bottlenecks described in Section 1.1, 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. 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 experiment.

---

8Although only CAOS's designers ever wrote CAOS applications.
Unfortunately, processor loading remained an issue. A configuration with poor load balancing in which some processing 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 load balancing was not really addressed by CAOS as agents were simply assigned to processing sites on a round-robin basis with no attempt to keep potentially busy agents apart (see Section 6 for a discussion of load balancing issues).

3.2 Evaluation of ELINT-CAOS Performance

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 two primary determiners of throughput and solution quality are the strategies used in making individual agents cooperate in producing the desired interpretation and 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 inter-agent cooperation strategies were used in our experiment:

NC: This "no cooperation" 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 primarily intended only as a baseline against which to compare more realistic strategies. What was surprising was that the strategy also produced quantitatively poor results (see below).

CC: In this strategy, agents cooperate in the creation of new agents via manager agents as described in Section 2. 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).

CT: The CT ("creation and time control") strategy was designed to additionally 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 2.

Table 1 illustrates the qualitative effects of the various control strategies and grid sizes. The table presents the five 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 1. | ELINT solution quality versus control strategies and grid sizes (* p 26) |
The major qualitative performance attributes are:

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 processing sites are overloaded or by the data rate being too high locally due to poor load balancing so that some subset of the processing sites are overloaded.

The CT 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 sufficiently 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 Chapter 3). 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 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 Chapter 3, 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 cooperation 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 achieved significant concurrency.

We interpret from Table 1 that the strategy used had a very significant 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.
Tables 2 and 3 clearly show that the processing cost of added cooperation was far outweighed by the benefits in its use. Far less message traffic was generated, and the overall simulated execution time was reduced. Note that for the runs whose execution times are shown in Table 2, the input data 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 was the minimum possible simulated execution time for these runs.

Table 3.  CAOS message counts for ELINT executions with various control strategies and grid sizes (*p. 29)

Table 4 and Figure 3 show the quantitative effect of the number of processing sites when the CT 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 2. The minimum possible simulated execution time for the runs used to produce Table 4 was 0.36 seconds.

Table 4.  Simulated ELINT execution time versus grid sized for production runs using CT control strategy (*p. 29)

As shown in Figure 5, the speedup achieved by increasing the number of processing sites is nearly linear in the 1 to 25 processing site range. However, the 36-site grid was slightly slower than the 25-site grid.

Figure 5.  The relative speedup of ELINT on various size CARE grid (*p.30)

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 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 5.  Simulated ELINT execution times and speedup for larger data sets.  (*p.30)

As shown in this table, for an input data set representing twice as many emitters and observations than the basic data set, the 36-site grid achieved 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.

9Because 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 output 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 variations in run times were as much as five percent. This could account for the slightly longer execution time on 26 versus 25 processors.
Some Lessons Learned.

Application dependent concurrency.

“Parallel” knowledge versus “serial” knowledge.

Utility of agent approach – a form of consistency maintenance.

Measuring performance.

Process instantiation and switching costs versus communication costs

Underutilization of resources.

Use of “manager” objects for coordination

Utility of object-local scheduling

Process switching costs

Stream versus futures

Hot spots

ELINT-CAOS load balancing scheme

Solution quality dependent on process scheduling

process granularity