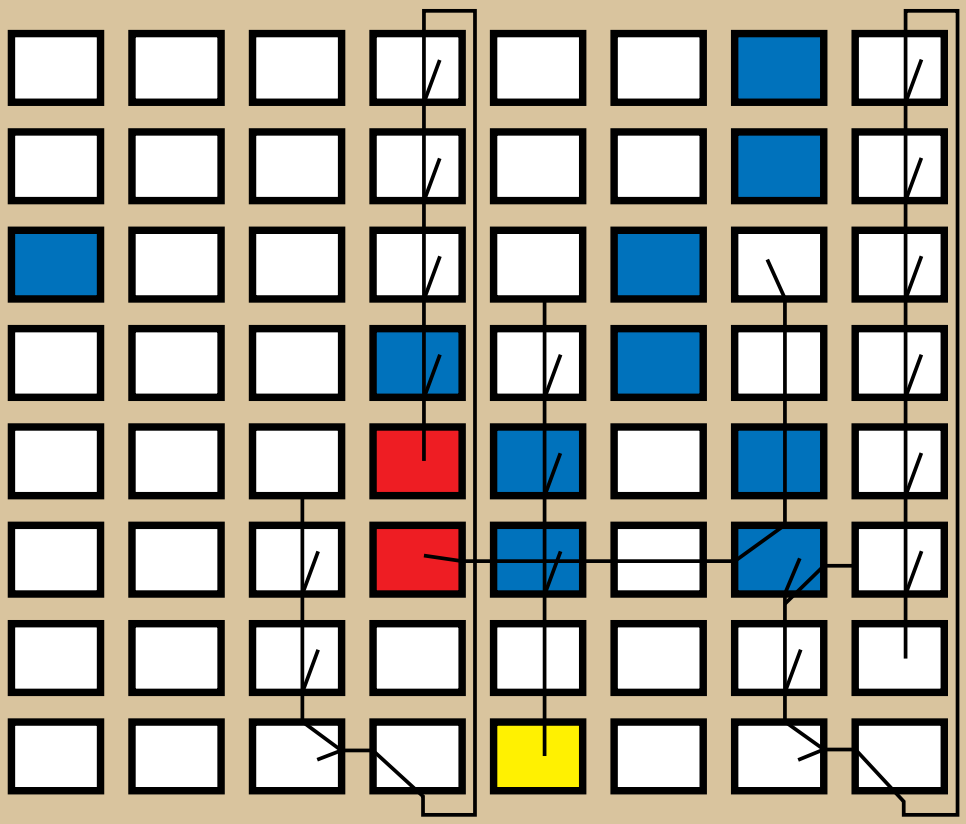
The Advanced Architectures Project



James Rice

The Advanced Architectures Project at Stanford University's Knowledge Systems Laboratory seeks to gain higher performance for expert system applications through the design of new, innovative software and hardware architectures. This research concentrates particularly on the use of parallel machines to gain speedup and the design of the software to exploit emergent parallel hardware architectures. This article describes the project and details its goals and the work performed in the pursuance of these goals. A brief description is given of each of the project components, and a complete bibliography appears of the publications produced for the project.

The Advanced Architectures Project (AAP)[1] has been running for a number of years at Stanford University's Knowledge Systems Laboratory (KSL). The project is large and has a number of components that have been documented at length. These components have never been drawn together in one document; thus, this article describes the project and gives a taste of the individual subprojects that have kept the project members so busy for so long. A large number of publications have emerged from the project, so a full bibliography of the work appears for the reader who wants to follow up on any intriguing topics.

AAP straddles a number of research areas and, thus, does not fall easily into any one sphere of interest. A certain amount of work has been done on the parallelizing of expert systems, most notably by Gupta (1986). Similarly, some machines on the market resemble in some respects the machines that project members have been designing, most notably the Ametek machine. This article does not in any significant way relate the work of AAP to the work in other fields; for such comparative studies, you can refer to the bibliography. The reader should keep in mind, however, that I know of no projects that have been comparable to AAP, and so it is often hard to find a reasonable point of comparison. For instance, considerable effort is being spent on concurrent image recognition on massively parallel machines such as the Connection Machine™, [2]but this research has had little influence on the work of this project.

At this point, I should make a brief disclaimer. The subject matter for AAP is complex and not in the mainstream of experience for members of the AI community. Thus, the AAP project members are concerned that the conclusions they have drawn from their work might be misinterpreted, trivialized, or overgeneralized. It is unrealistic in such a short article to try to do justice to the full lessons that can be learned from the project's extensive quantitative experiments. For this reason, I do not attempt to draw any great conclusions here. The reader is strongly encouraged to read any of the 50-odd papers that are cited here to gain a deeper understanding of the work and the lessons learned. All these publications are available from KSL, and many have also been published elsewhere. This article, then, is more of a concept or a research in progress article than one presenting scientific goals.

Project Goals

The project's primary goal is to find ways to increase the performance of expert systems through the use of the new, emergent, parallel hardware designs.

The number of possible implementation strategies for such a project is huge. One only has to look at the large number of different hardware designs that are emerging and at the number of different problem-solving methods to see how combinatorial the problem would be if the project tried to investigate all the reasonable and plausible architectural combinations. It was decided, therefore, that the project members could learn a great deal by simply making a commitment to one, or at least a small number of, different options at each point in the system's makeup. Thus, it was decided that the project would take a "vertical slice" through the space of possible solutions. Clearly, the members did not intend to investigate any options that seemed nonuseful. Although they could not prove they had the best design to meet their goals, they knew from the outset their design would at least be a plausible architecture for a future computational environment.

. . . it was decided that the project would take a "vertical slice" through the space of possible solutions.

The project members viewed the task of implementing concurrent expert systems as one that was split into a number of implementation layers. If they could achieve speedup at each one of these layers, then they could hope for a substantial overall performance improvement compared to existing AI systems. The model of the layers into which the project could be split appears in figure 1.

It was originally anticipated that the needs of the applications would drive the development of the problem-solving frameworks and the needs of the problem-solving frameworks would drive the development of the knowledge retrieval layer and so on until eventually the hardware would be designed under the constraints passed down from above (figure 1). In practice, however, this process did not occur. Because of the difficulty of finding and

0738-4602/89/$3.50 ©1989 AAAI

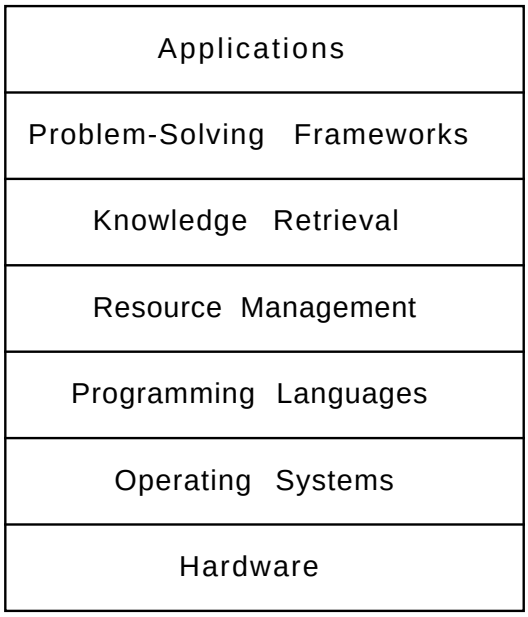


Figure 1. The Layers of System Implementation Through Which Project Members Hoped to Achieve Computational Speedup.

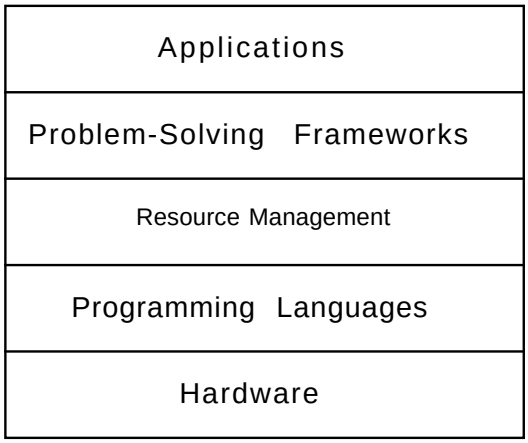


Figure 2. Layers Actually Tackled in the Project. Resource management appears in small type because it is a recent addition, and most of the work has been done without the help of this layer.

mounting an application suitable to the project's needs and the early availability of personnel interested in the hardware design aspect, hardware design proceeded more rapidly than the other layers. Thus, the designs were more hardware driven than application driven. Such a result is not necessarily bad because an entirely top-down design process could have easily resulted in low-level system requirements that could not have been implemented.

The levels of abstraction actually implemented differed significantly from those shown in figure 1. Figure 2 gives a more realistic representation of what the project members actually did, as opposed to what they intended to do.

KSL staff members have considerably more expertise in software than in hardware. Project members decided early not to build any hardware; many other research groups are better at hardware development. Therefore, they decided to simulate their hardware, allowing them to modify the software and hardware designs easily and extract the maximum insight with the minimum effort.

The remainder of this article is split into sections that reflect the major layers shown in figure 2. In each of these sections, the work of the relevant subprojects is discussed. Because of the project's bottom-up emphasis, the components are discussed in bottom-up order. This format reduces the number of forward references made because a discussion of the higher layers inevitably has to refer to the substrates on which they are implemented.

Personnel

This project has employed a large number of people over the years. It seems appropriate to name them all here because otherwise they might only appear as authors referenced in the bibliography: Ed Feigenbaum, Bob Engelmore, Penny Nii, Bruce Delagi, Harold Brown, Hiroshi Okuno, John Delaney, Byron Davies, Hirotoshi Maegawa, Nelleke Aiello, James Rice, Nakul Saraiya, Sayuri Nishimura, Eric Schoen, Greg Byrd, Max Hailperin, Russel Nakano, Masafumi Minami, Chris Rogers, Alan Noble, Jean-Christophe Bandini, Manu Thapar, Pandu Nayak, Djuki Muliawan, Jerry Yan, and Sam Hahn.

Hardware

As mentioned earlier, hardware design led the way in AAP. In this section, I discuss a little bit of the motivation for the hardware designs and briefly describe the current generation of hardware designs that project members are working on and the simulator they are using.

Simple and Helios

The hub of all the work done on AAP has been the circuit simulator; everything else is

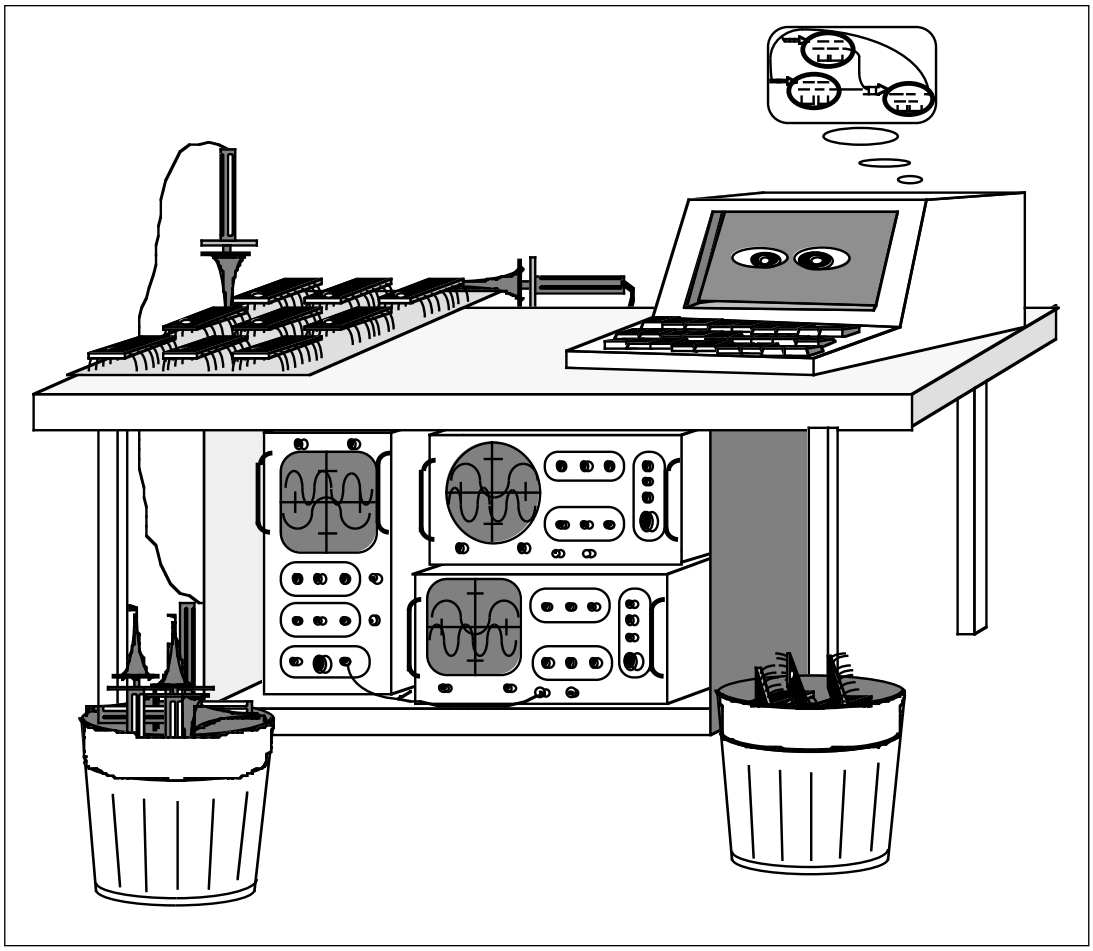


Figure 3. The Simple System Provides a Tool Kit from Which to Build Circuits to Be Simulated, a Collection of Probes to Connect to the Circuit, and a Set of Instruments to Connect to the Probes.

built on it. This simulator is called Simple. It is an event-driven simulator devised to allow the user to design and specialize digital circuits in a simple and modular way using a circuit design tool called Helios. A sophisticated set of instrument tools allows the user to design and specialize simulated probes that can be connected to the circuit while it is running. This setup allows a number of instruments to be connected to the probes that permit the user to see the behavior of the circuit as it operates without interfering with the system's behavior. Project members like to view this model as a laboratory workbench equipped with collections of instruments, probes, and circuit-building components from which the user can build systems and on which the user can perform quantitative experiments (figure 3) (Delagi et al. 1986, 1987).

It was found early on that simulations of the sort project members wanted to do would be computationally expensive. An attempt was made, therefore, to parallelize the simulator itself in an attempt to reduce the time taken for the simulations, which often exceeded one day. This attempt resulted in AIDE, a distributed version of Simple (Saraiya 1986). Unfortunately, project members were unable to achieve any speedup at all for their simulations, largely because of the communications bandwidth necessary and the latency associated with communicating between the multiple Symbolics machines used by way of Ethernet. Also, the simulator was event driven and required frequent synchronization on the event queue, which serialized the processing.

CARE

The Simple simulator was used to design and build what project members refer to as the CARE^3 machine and simulation system (Delagi et al. 1988) (figure 4). The CARE machine is the simulated machine on which all the experiments mentioned in the following paragraphs were performed. The

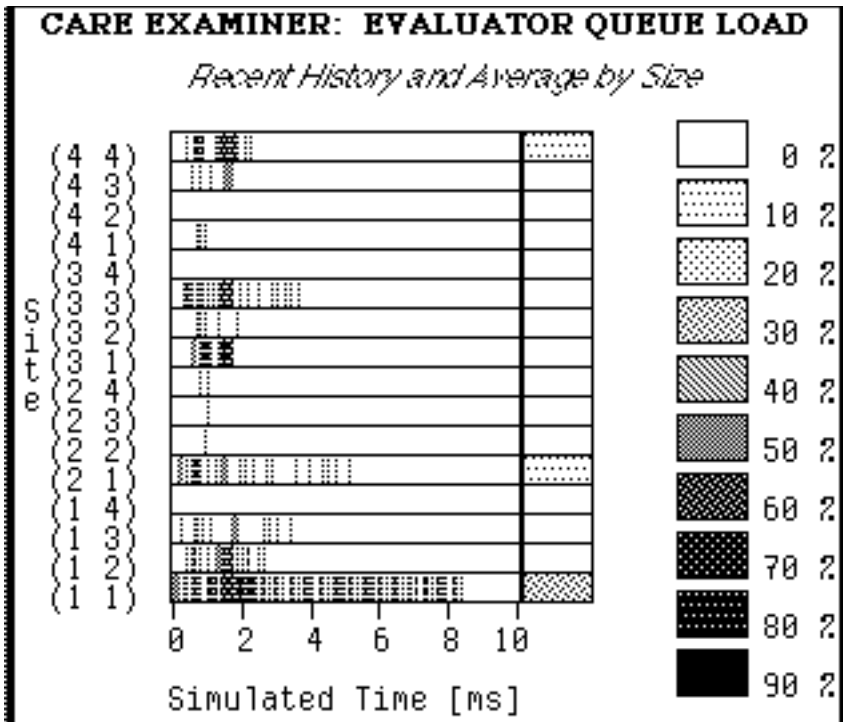


Figure 4. An Example Piece of Instrumentation from the CARE System. This figure shows the lengths of the task queues on the different processors plotted against time.

machine's design has a few key features that are worthy of note:

- Dynamic cut-through routing to optimize network throughput (Byrd, Nakano, and Delagi 1987a)
- Toroidal topology (Byrd and Delagi 1987)
- Nonblocking message sending to encourage pipeline processing
- Communications network with alternative paths between points to reduce communications problems resulting from busy communication paths
- A separate communications controller to support operating system functions and implement the nonblocking send functionality mentioned previously

The work on the CARE subproject has focused mainly on the design of interprocessor communication networks, as is appropriate. Thus, project members have been able to ignore the instruction-level behavior of the processors themselves. The application programs that the members run are merely timed as they run between the points at which code fragments cause communication between processors. Being able to avoid doing register-level simulation of the processors themselves has allowed project members to execute much more complex and realistic programs on their simulated machines. Therefore, they traded accuracy in their processor simulation—assuming that the processing elements will behave much like existing Lisp machine processors—in favor of greater realism in terms of the system's performance under the load of real programs.

A number of system design aspects have not been addressed in detail, and the simulations do not take these into account. Perhaps most significant is the fact that memory usage, code distribution, and garbage collection are not simulated.

The CARE/Simple simulator system is perhaps the most valuable tangible product of the project. It is now being used in a number of research departments, both corporate and academic, outside Stanford. Like all AAP software, it is in the public domain. CARE/Simple will soon be running under Common Lisp, CLUE, and X11 on a number of different platforms.

Operating Systems and Languages

A considerable amount of effort has been spent working at the operating system level of abstraction. Curiously, the project members have written no operating systems. The CARE machine itself features a dual processor for each processing element. This arrangement allows much of the work of the operating system, particularly interprocessor communication, to be done by a dedicated processor in parallel with the execution of user code. The behavior of this communications processor is coded directly into the simulated hardware. Work in this area has been done on concurrent object-oriented systems, concurrent Lisp dialects, programming models, and resource allocation.

CAREL

CAREL (Davies 1986) was one of the first programs written to run on the CARE simulated machine. It was an early attempt to find a Lisp language interface to the distributed-memory hardware provided by CARE. It took as its basis Scheme (Abelson and Sussman 1983) and QLisp (Gabriel and McCarthy 1984) and included primitives to allow remote function calls and remote CONSing. It was quickly found that because of the cost of process creation, it was desirable to make the best use of any processes which were spawned. Therefore, there was a need to store application-dependent data in nonephemeral spawned processes.

Work in this area has been done on concurrent object-oriented systems, concurrent Lisp dialects, programming models, and resource allocation.

State of this type was implemented in CAREL as writable closure variables. These process closures could be used as elements in pipeline computations or as representations of mutable communicating program objects, for instance, to represent real-world objects with state. State, as encapsulated in communicating objects, and the idea of pipeline parallelism have been pivotal in the design of the other systems developed in the project. The CAREL project was used primarily as a feasibility study and was soon discontinued.

CAOS

The first implementation of the Elint application, described later, was made without the benefit of any problem-solving framework. It was anticipated that the application could easily be mounted almost directly on the CARE machine and that some experiments could be run quickly, which would allow project members to learn some important lessons early in the project.

To mount the application, a distributed object-oriented system was implemented; at the time, the CARE system did not come with its own preferred object system. The system was called CAOS (concurrent asynchronous object-oriented system) (Schoen 1986). It was implemented using the Flavors system supported by the project's Lisp machines. It had a number of key features.

First, each CAOS object was potentially a multiprocess object, although executing on a single processor, with at least one stack group associated with each CAOS object.

Second, CAOS objects were intentionally large grained. Because it was anticipated that the communications network would be the resource most competed for, the programmer was encouraged to perform a lot of computation to reduce the number or size of the messages sent.

Third, message passing was used as the metaphor for communication in the language extensions provided by CAOS.

Finally, a large number of different message-sending primitives were defined, including nonblocking sends that did not require a reply from the message target, sends which returned futures to the values returned by the targets, and send operations that immediately blocked to wait for a reply from their targets.

The CAOS system proved to be too expensive to use for future experiments. Contrary to the intuition of the project members, the communications network proved to be the least loaded of the CARE machine's resources during our experiments on CAOS. The computational expense of supporting its complex object model caused the granularity of the resulting computations to be too large.

LAMINA

Lamina (Delagi, Saraiya, and Byrd 1986) is the object system that was designed after the lessons learned from the CAOS experiments. It was originally intended to provide a small, lightweight layer on top of the CARE machine so that distributed object-oriented programs could be efficiently implemented. A significant part of the motivation for Lamina's design was the desire to reduce the overhead suffered by the CAOS system in terms of associating large stack groups with each of the CAOS objects. Lamina introduced the idea of objects with restartable, rather than resumable, code segments, which do not require stacks to preserve their state when they are not running. Since its first appearance, Lamina has been extensively developed, and although still small and lightweight, it now provides a platform for the development of computational models for functional and shared-variable, as well as object-oriented, programming.

Lamina has been used to implement a number of programs, both for direct implementations of the two real-time expert systems being investigated (see Applications)—AirTrac and Elint—and for a number of numeric programs. Lamina is now the preferred core programming system for the CARE machine. Applications in Lamina have consistently shown the highest performance of all programs running on the CARE machine.

Interprocessor and Interprocess Communication

Different mechanisms for interprocessor and interprocess communication have been

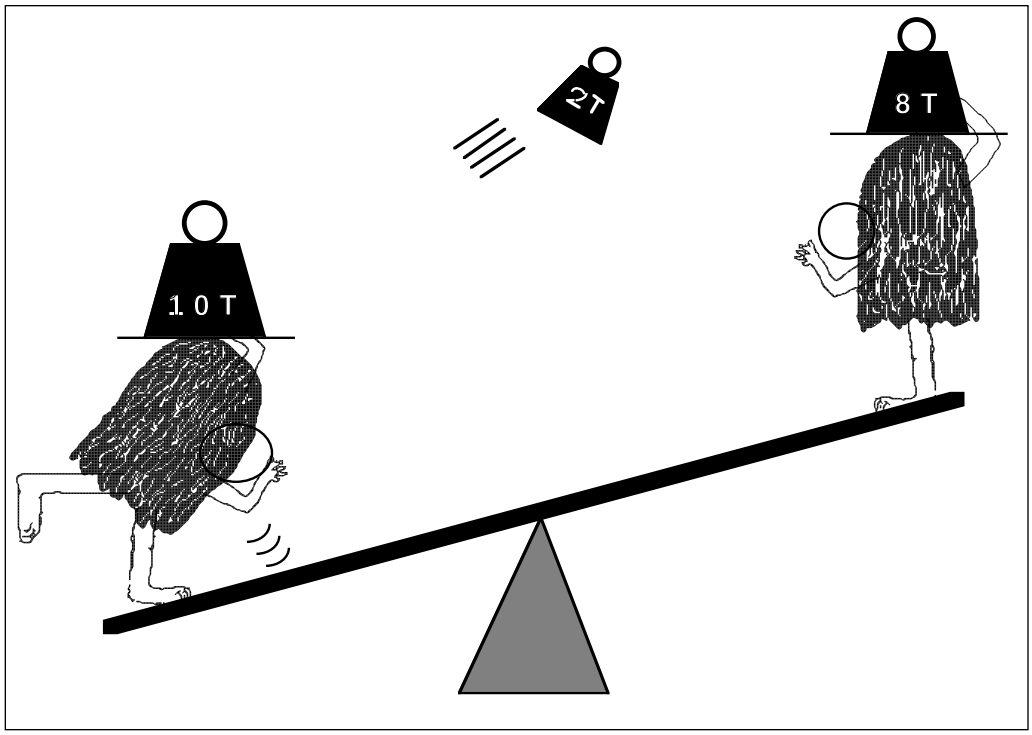


Figure 5. Load Balancing.

extensively investigated. For distributed-memory machines, project members believe that the efficient distribution of work for large applications is crucially linked to the efficient implementation of multicast communication (Byrd, Nakano, and Delagi 1987; Byrd, Saraiya, and Delagi 1988). Although the principal emphasis has been on the development of distributed-memory hardware, the fact that the CARE simulator can also simulate shared-memory machines has allowed project members to investigate the relative performance of these two distinct classes of machines and the relative performance and appropriateness of shared-variable and message-passing, object-oriented programming models (Byrd and Delagi 1988). Current work focuses on the design of hardware that might provide efficient support for both the shared-variable and the message-passing programming models (Byrd 1989).

Load Balancing

Project members have started examining load-balancing problems (figure 5) within the context of the AAP vertical slice (Hailperin 1988). In particular, this work focuses on a load-balancing method intended to migrate Lamina objects in large (thousands of processing elements) CARE multicomputers to improve the performance of soft–real-time signal-interpretation systems such as Elint and AirTrac (see Applications).

Without load balancing, only a lightly loaded multicomputer that dynamically creates processes can, in general, achieve real-time performance. The work focuses on how to achieve global load balancing, which would be an attractive solution to this problem because it would allow the effective use of massively parallel ensemble architectures for larger soft–real-time problems.

The challenge is to replace quick global communication, which is impractical in a massively parallel system, with statistical techniques. In this vein, a novel approach to decentralized load balancing is being investigated based on statistical time-series analysis. Each processing element estimates the systemwide average load using information about past loads of individual sites and attempts to equal this average. This estimation process is practical because the soft–real-time systems in which the project is interested naturally exhibit loads that are periodic (in a statistical sense akin to seasonality in econometrics).

A load-balancing system for Lamina/CARE was designed using this load-characterization technique. It has been implemented, and experiments with it in the context of ELINT and AIRTRAC have begun.

Concurrent and High-Performance Lisp

To understand the behavior of Lisp on shared-memory machines, work was done on the QLisp system (Okuno and Gupta 1987). Although this work was not directly used by other parts of the project, it involved examining some of the constraints on parallelizing production systems by studying the OPS5 language.

In the search for higher-performance symbolic computation, work was also done on the development of high-performance Lisp interpreters (Okuno, Osato, and Takeuchi 1987). This work was also not directly used in

the project because all the code used in the project's experiments has been compiled.

Problem-Solving Frameworks

One of the key layers in the AAP strategy was problem-solving frameworks. Faced with a large number of different problem-solving models, the project committed itself early to the blackboard problem-solving model (Engelmore and Morgan 1988). This choice was not entirely arbitrary. The blackboard metaphor was successfully applied in the area of real-time signal processing (Nii et al. 1982), the selected problem domain for AAP. Also, it was anticipated that the blackboard metaphor would help project members extract parallelism from the application in the way that the problems were formulated because the metaphor has a model of asynchrony built into it. For reasons detailed in Rice (1988c), the blackboard model turned out to not be as parallel as was hoped, but project members still know of no better one for concurrent execution.

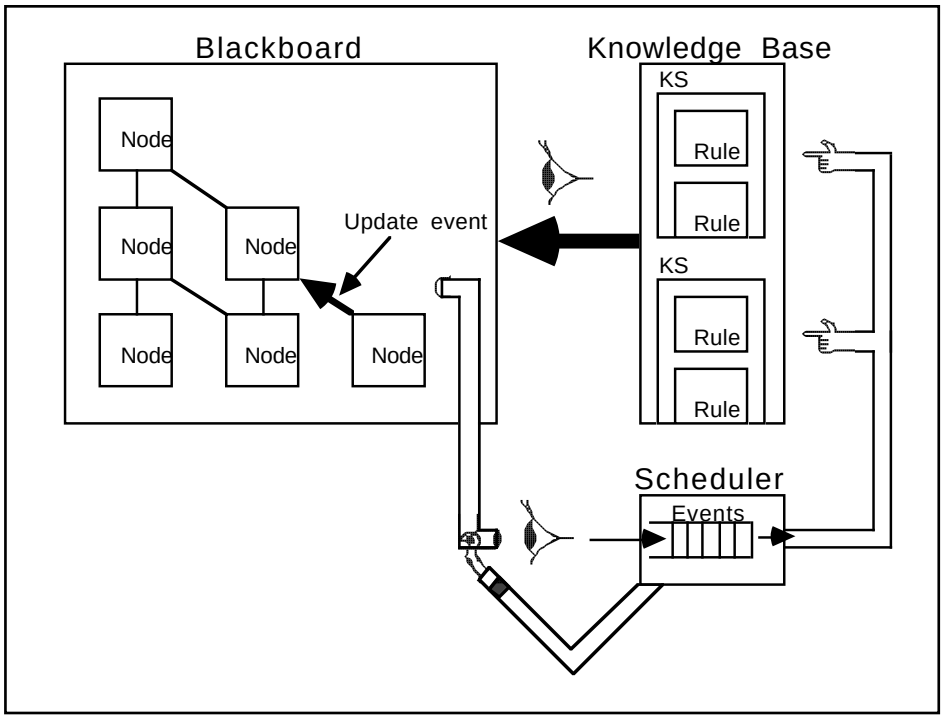


Figure 6. The Cage Architecture.
Update events are perceived by the scheduling component and collected in a global event queue. The scheduler selects the knowledge sources that are interested in any given event and can execute them in parallel. In turn, these knowledge sources inspect the blackboard and perform updates that are seen by the scheduler.

The development of problem-solving frameworks took two distinct courses. First was the development of a fairly conservative concurrent implementation of an existing blackboard system to run on existing shared-memory machines. This system is described in the next subection. Second was the need to rethink the blackboard metaphor from scratch in the hope of achieving really high performance on distributed-memory multiprocessors, such as the CARE machine. The result was the Poligon system. Three generations of papers were produced describing project strategy, the Cage and Poligon systems as they evolved, and the experimental results produced by these systems (Nii 1986; Nii, Aiello, and Rice 1988a, 1988b).

Cage

Cage (Concurrent AGE) (Aiello 1986) is a reimplementation of the AGE (Nii and Aiello 1979) blackboard system developed by the Heuristic Programming Project at Stanford. The central idea behind Cage is that the blackboard model, by its nature, provides a certain amount of parallelism. Therefore, it should be possible to exploit this parallelism without any major redesign or rethinking of the problem-solving model. Cage is, therefore, an implementation, that is designed to allow the concurrent execution of a blackboard system through the concurrent execution of the knowledge sources and rules in the application (figure 6).

At the outset, it was not known how difficult it would be to program such a system and how much performance could be expected. However, project members thought that such an architecture might well be suitable for the current generation of multiprocessors, which mostly have a shared-memory design. Blackboard systems are typically implemented using a central, shared database to represent the blackboard. The match between the shared blackboard and the shared-memory resource seemed to be worth investigating. The Cage system was implemented first on a simple emulator, which emulated the functionality of a QLisp implementation without paying the costs of detailed simulation. It was later ported to run on the CARE simulator

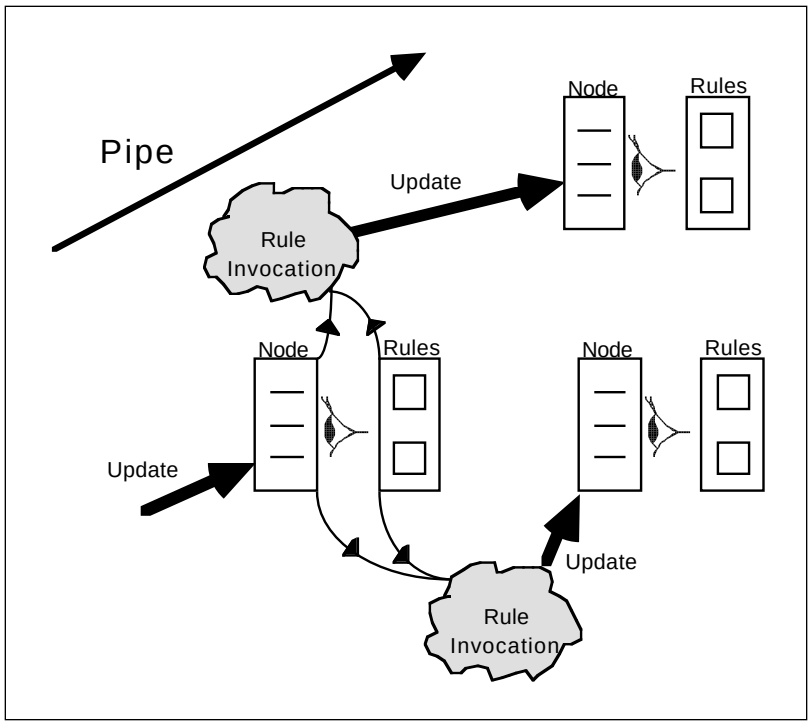


Figure 7. The Poligon Architecture.
Updates on the blackboard are observed by rules that watch specific slots of blackboard nodes. These rules can fire in parallel, causing further updates to the same or other nodes. This flow of updates from one node to another implicitly forms pipes, which increase the parallelism realizable by the system.

using an implementation of QLisp built on top of the Lamina shared-variable programming interface (Saraiya 1988).

The Elint application, discussed later, was mounted on the Cage system, and experiments were performed on it. These experiments are detailed in Aiello (1988) and Rice and Aiello (1989). The Cage system showed that blackboard programs can, indeed, be run in parallel in a relatively simple manner. The performance of Cage, however, is restricted by a number of factors (Nii, Aiello, and Rice 1988b): (1) its implementation, which was not highly tuned; (2) its architecture, which exhibits significant contention for global shared resources such as the event queue; (3) the QLisp substrate on which it is built; and (4) the shared-memory hardware on which it runs.

Thus, the Cage architecture is viable for existing shared-memory hardware systems, but because of the close link between the Cage programming model and its underlying hardware, project members do not anticipate that future concurrent expert system tools will be built like Cage. Instead, they believe that the trend of multiprocessor design is moving away from shared-memory machines toward distributed-memory designs because of their greater ability to scale. Software design is likely to track this trend.

Poligon

The expectation is that for reasons of simplicity, performance and cost, the next generation of multiprocessors is likely to be distributed-memory machines. This expectation required rethinking the blackboard model so that it could be mounted on such machines in a manner likely to deliver good performance. Poligon (Rice 1986a, 1986b, 1989) was developed in an attempt to address these needs. It took the view that processors were going to be cheap and plentiful; thus, if necessary, it was quite acceptable to allocate one processor or more to each node on the blackboard.

First, the serializing, centralized control mechanism of conventional blackboard systems was discarded. Distributing the nodes of the blackboard over the processor network allowed the knowledge base to be spread over the blackboard as well; this arrangement eliminated any performance bottleneck as a result of the costs of communication between the knowledge base and the blackboard. The simplest available rule-invocation mechanism was selected to maximize performance; rules were directly attached to slots of the nodes on the blackboard. Modifying a slot resulted in invoking the rule attached to it. Rule invocations were spun off into different processes on different processors for execution; thus, the length of the critical sections on the processors holding blackboard nodes was minimized, and multiple, simultaneous rule invocations for the same modified blackboard object were allowed (figure 7).

In practice, these mechanisms did, indeed, result in good performance, but they also resulted in significant problems. Lots of uncontrolled asynchronous processes all reading and writing in a shared database are bound to cause problems when it comes to getting a coherent or correct answer. Extra mechanisms had to be implemented, which allowed the blackboard nodes to have goals and the ability to evaluate their own performance with respect to the overall system goal. These mechanisms allowed the blackboard nodes to have the final decision about whether to perform any modification operation attempted by a rule. The result was a sort of distributed hill-climbing behavior. Nodes iterated toward a good solution.

These mechanisms did not come without

associated costs in terms of granularity. Although the Poligon system delivers high performance when compared to other blackboard systems such as AGE, it significantly lacks the performance provided by an application written directly in Lamina. Therefore, Poligon provides a fairly general concurrent implementation of the blackboard problem-solving model with all the accompanying advantages of abstraction and modularity. It does so, however, at a price. A detailed description of Poligon's design and implementation can be found in Rice (1989), which also describes how Poligon's performance could be improved by superior compilation if it were to be turned into a production-quality system.

The Elint application, described in the next section, was implemented in the Cage, Poligon, and Lamina systems. The results of these experiments are reported in Rice (1988b); Rice and Aiello (1989); Nii, Aiello, and Rice (1988b); and Nii and Rice (1989). Another application called ParAble, (Bandini 1989) implemented using the Poligon framework, is also described in the next section.

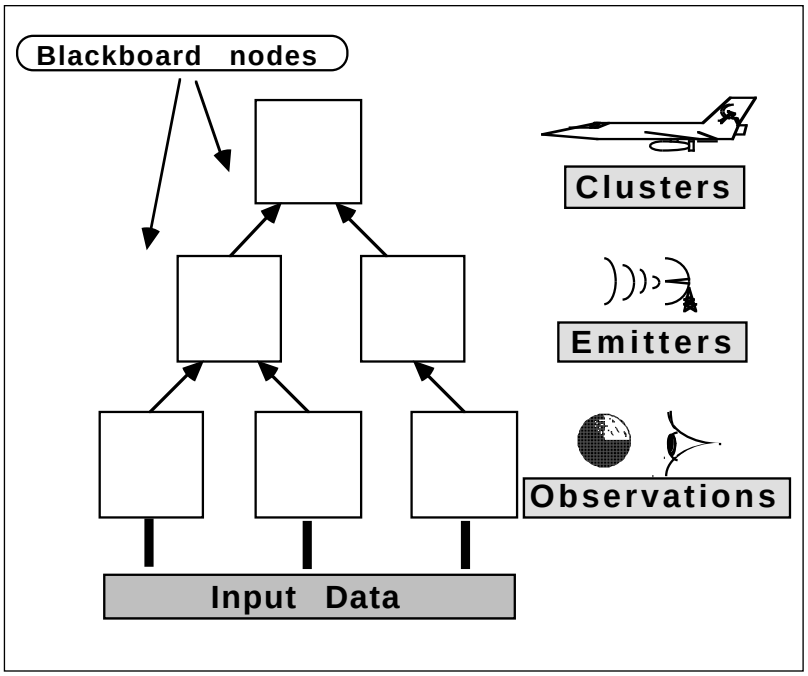


Figure 8. The Elint Application.
Sensor data are abstracted into hypothetical radar emitters, which are tracked as clusters of emitters.

Applications

As mentioned in the introduction, the project members expected at the outset that AAP would be application driven. In the search for an application domain, which would need significant speedup for expert systems to be fielded and yet held a certain obvious potential for concurrent execution, they picked the field of real-time signal understanding. Existing blackboard systems, such as HASP/SIAP (Nii et al. 1982) and Tricero (Williams, Brown, and Barnes 1984) showed that the blackboard problem-solving model was appropriate for this domain and that the performance deliverable using the existing blackboard tools was entirely inadequate to field such systems.

What was needed, therefore, was a problem complex enough to give us a reasonable model of a real system and yet simple enough to prevent project members from expending too much effort on the mechanics of its implementation. The unavailability of a satisfactory application at the start of the project caused it to become somewhat more hardware driven than originally expected. It was eventually decided that project members would focus on a problem called Elint, a system for understanding passive radar signals. This application is described in the next section.

After a fair bit of experimentation, it was determined that the ability to exploit parallelism was being constrained by the problem being used; it was not sufficiently complex. In the search for a more knowledge-rich and computationally intensive application, project members developed AirTrac (described later). Work was also done in areas other than real-time signal understanding; ParAble (described later), a system for fault finding in particle accelerator beamlines, was developed using the Poligon framework. A number of numeric or seminumeric programs were also developed during the hardware-related experiments.

Elint

Elint is a soft–real-time system for interpreting passive radar signals. Data are collected from a number of receiving stations and integrated to allow the system to track radar-emitting aircraft as they pass through the monitored airspace. The data are abstracted into hypothetical radar-emitting platforms. In turn, these platforms are collected into clusters of emitters, which might represent a number of planes or a single plane using multiple radar systems, as is often the case with modern military aircraft (figure 8). Elint was first implemented using the CAOS system. It was originally thought that this work would

. . . the project members expected at the outset that AAP (Advanced Architectures Project) would be application driven.

take only a couple of months. However, the complete task—implementation, experimentation, and analysis of results—took 18 months. Project members learned early that it is by no means a trivial matter to reimplement an existing serial application in a parallel environment. These initial experiments are detailed in Brown, Schoen, and Delagi (1986).

Since the CAOS implementation, Elint has been implemented three times, using Lamina (Delagi and Saraiya 1988; Saraiya 1989) and the Cage (Aiello 1988) and Poligon (Rice 1988b; Nii, Aiello, and Rice 1988b) frameworks, and a number of experiments have been performed.

AirTrac

The development of the Elint application showed that the amount of parallelism that could be demonstrated was much more dependent on the application than had been anticipated. Project members had hoped that by extracting parallelism at the different levels of the system's implementation hierarchy, they could gain significant speedup. However, they were unable to demonstrate speedup in this way. They were able to demonstrate that their experiments showed poor speedup largely because the application itself had run out of parallelism.

What project members needed was an application that would really stretch the hardware and software they were developing in a realistic manner. As a result, the AirTrac application was developed (Delaney 1986).

The AirTrac problem domain sounds superficially like that of Elint. It was a system for interpreting radar data, although in this case, the radar systems being modeled were active not passive. Unlike Elint, AirTrac was designed to go much further than simply tracking aircraft and finding likely threats.

The scenario for AirTrac was the detection of smugglers flying across a border. The problem faced by existing radar users is that a large number of legitimate aircraft travel in the same airspace as smugglers. Smugglers can take advantage of variations in terrain to find areas of poor or no radar reception. They also resort to other evasive tactics.

The system was designed in a number of layers so that different implementation efforts could be decoupled. The first subsystem implemented was called the data-association component (Nakano and Minami 1987) and is the subsystem that most closely matches the Elint application. Initially, this component was to be implemented using the Poligon framework. However, it was found that simulating the Poligon system for a problem as complex as AirTrac would take prohibitively long. Consequently, AirTrac was directly implemented in Lamina. Substantial speedup was shown, which seemed to increase linearly with the number of processors. This result was encouraging.

The second component of AirTrac, Path Association (Noble and Rogers 1988), was significantly more knowledge intensive than the first. This subsystem was also initially implemented directly in Lamina. However, programming in the raw Lamina framework was too complex and time consuming, so a layer called ELMA was built on top of Lamina that provided the abstraction model needed for the implementation (Noble 1988b).

The final, most abstract component of AirTrac has not been implemented. Project members have not yet learned all they can from the second layer and were not able to show all the speedup that they thought was possible in this layer, so work is continuing in this area.

ParAble

The ParAble project (Bandini 1989) was an attempt to test the generality of the problem-solving model offered by Poligon by choosing a completely different application domain. To achieve this goal, a parallel implementation of the ABLE system (Selig 1987), which was also developed at Stanford, was made.

The objective of the ABLE project was to find a fast way to diagnose difficulties with particle accelerator beam lines. These large and complex machines are prone to beam alignment problems because (1) the magnets that steer and focus the beam are misaligned or (2) the power supplies to these magnets are incorrect, resulting in the magnets not having the desired strength. These systems

are so complex that it can take many months of knob twiddling simply to commission them.

The ABLE system used an analytic model of the beam-line–component transfer functions and a number of heuristics that employ successive model runs and compare the results with real data to locate the faults. It was thus able to find faults in particle accelerator systems in about 10 minutes. As these systems become more complex, there might well be a need to control them in real time. Thus, although no immediate need exists for higher performance in the ABLE system, it is reasonable to suppose the need might exist in the future.

A number of experiments were performed on ParAble and are detailed in Bandini (1989). Again, the realizable parallelism in this project was found to be limited mostly by the availability of data parallelism.

Numeric and Seminumeric Programs

The expert systems mentioned here are not ideal applications for multiprocessor execution. They are irregular and data dependent. A large body of applications already exists in the area of numeric and seminumeric processing that will require the speedup associated with parallel execution. Indeed, such programs are already being run on a number of multiprocessors. Therefore, it is essential that any machine designed to be general purpose must also be able to execute these regular, argorithmic problems efficiently. A number of small numeric programs have been developed that allow project members to test their hardware and software ideas in a much more controllable way than is possible with any expert system application. Among these are a Gaussian elimination algorithm, a partial differential equation solver, and an integrated circuit line simulator.

Conclusions

AAP has been running for a number of years now. The project members have found that they have moved from having a good understanding of AI problem-solving techniques and knowing little about parallel computation to having have a good understanding of all the issues involved; the problems facing the implementers of concurrent problem-solving systems, and the gains they can reasonably expect. The project members hope that the successes and failures reported in the publications cited in the bibliography will help the rest of the research community in the move to concurrent computational platforms.

Notes

1. The project's real name is Expert Systems on Multiprocessor Architectures, but all the project members have always felt more comfortable calling it either the Advanced Architectures or the Architectures project for short. This name more effectively captures the fact that new architectures for both hardware and software will be needed to meet the project's goals.

2. Connection Machine is a registered trademark of Thinking Machines Corp.

3. The expansion for this acronym seems to have been lost somewhere in the wash. I think that it has something to do with the words concurrent and array.

Bibliography
Advanced Architectures
Project Publications

Aiello, N. 1988. Cage: The Performance of a Concurrent Blackboard Environment, Technical Report, KSL-88-80, Heuristic Programming Project, Dept. of Computer Science, Stanford Univ.

Aiello, N. 1986. User-Directed Control of Parallelism: The CAGE System, Technical Report, KSL-86-31, Heuristic Programming Project, Dept. of Computer Science, Stanford Univ. Also in 1986. Proceedings of the Defense Advanced Research Projects Agency Expert Systems Workshop, 146–151. Arlington, Va.: Science Applications International Corp.

Bandini, J. C. 1989. Poligon Applications, Technical Report, KSL-89-43, Heuristic Programming Project, Dept. of Computer Science, Stanford Univ.

Brown, H.; Schoen, E.; and Delagi, B. 1986. An Experiment in Knowledge-Base Signal Understanding Using Parallel Architectures, Technical Report, STAN-CS-86-1136, Heuristic Programming Project, Dept. of Computer Science, Stanford Univ. Also in 1986. Proceedings of the Defense Advanced Research Projects Agency Expert Systems Workshop, 93–105. Arlington, Va.: Science Applications International Corp.

Byrd, G. 1989. Support for Fine-Grained Message Passing in Shared-Memory Multiprocessors, Technical Report, KSL-89-15, Heuristic Programming Project, Dept. of Computer Science, Stanford Univ., 1989.

Byrd, G., and Delagi, B. 1988. A Performance Comparison of Shared Variables versus Message Passing, Technical Report, KSL-88-10, Heuristic Programming Project, Dept. of Computer Science, Stanford Univ. Also in 1988. Proceedings of the Third International Conference on Supercomputing, 1–7. Boston, Mass.: International Supercomputing Institute.

Byrd, G., and Delagi, B. 1987. Considerations for Multiprocessor Topologies, Technical Report, KSL-87-07, Heuristic Programming Project, Dept. of Computer Science, Stanford Univ. Also in 1987. Proceedings of the Defense Advanced Research Pro-

jects Agency Knowledge-Based Systems Workshop, 119–122. Arlington, Va.: Science Applications International Corp.

Byrd, G; Nakano, R.; and Delagi, B. 1987a. A Dynamic, Cut-Through Communications Protocol with Multicast, Technical Report, STAN–CS–87–1178, Heuristic Programming Project, Dept. of Computer Science, Stanford Univ.

Byrd, G.; Nakano, R.; and Delagi, B. 1987b. A Point-to-Point Multicast Communications Protocol, Technical Report, KSL-87-02, Heuristic Programming Project, Dept. of Computer Science, Stanford Univ.

Byrd, G.; Saraiya, N.; and Delagi, B. 1988. Multicast Communication in Multiprocessor Systems, Technical Report, KSL-88-81, Heuristic Programming Project, Dept. of Computer Science, Stanford Univ.

Davies, B. 1986. CAREL: A Visible Distributed Lisp, Technical Report, KSL-86-14, Heuristic Programming Project, Dept. of Computer Science, Stanford Univ. Also in 1986. Proceedings of the Defense Advanced Research Projects Agency Expert Systems Workshop, 171–178. Arlington, Va.: Science Applications International Corp.

Delagi, B., and Saraiya, N. 1988. ELINT in LAMINA: Application of a Concurrent Object Language, Technical Report, KSL-88-33, Heuristic Programming Project, Dept. of Computer Science, Stanford Univ.

Delagi, B.; Saraiya, N.; and Byrd, G. 1986. LAMINA: CARE Applications Interface, Technical Report, KSL-86-67, Heuristic Programming Project, Dept. of Computer Science, Stanford Univ. Also in 1988. Proceedings of the Third International Conference on Supercomputing, 12–21. Boston, Mass.: International Supercomputing Institute.

Delagi, B.; Saraiya, N.; Byrd, G.; and Nishimura, S. 1988. CARE User's Manual, Technical Report, KSL-88-53, Heuristic Programming Project, Dept. of Computer Science, Stanford Univ.

Delagi, B.; Saraiya, N.; Nishimura, S.; and Byrd, G. 1987. Instrumented Architectural Simulation, Technical Report, STAN–CS–87–1189, Heuristic Programming Project, Dept. of Computer Science, Stanford Univ. Also in 1988. Proceedings of the Third International Conference on Supercomputing, 8–11. Boston, Mass.: International Supercomputing Institute.

Delagi, B.; Saraiya, N.; Nishimura, S.; and Byrd, G. 1986. An Instrumented Architectural Simulation System, Technical Report, KSL-86-36, Heuristic Programming Project, Dept. of Computer Science, Stanford Univ. Also in 1988. *Artificial Intelligence and Simulation: The Diversity of Application*. San Diego, Calif.: The Society for Computer Simulation International. Also in 1986. Proceedings of the Defense Advanced Research Projects Agency Expert Systems Workshop, 106–118. Arlington, Va.: Science Applications International Corp.

Delaney, J. 1986. Multi-System Report Integration Using Blackboards, Technical Report, KSL-86-20, Heuristic Programming Project, Dept. of Computer Science, Stanford Univ. Also in 1986. Proceedings of the Defense Advanced Research Projects Agency Expert Systems Workshop, 179–184. Arlington, Va.: Science Applications International Corp.

Hailperin, M. 1988. Load Balancing for Massively Parallel Soft–Real-Time Systems, Technical Report, KSL-88-62, Heuristic Programming Project, Dept. of Computer Science, Stanford Univ.

Nakano, R., and Minami, M. 1987. Experiments with a Knowledge-Based System on a Multiprocessor, Technical Report, KSL-87-61, Heuristic Programming Project, Dept. of Computer Science, Stanford Univ.

Nii, H. P. 1986. CAGE and POLIGON: Two Frameworks for Blackboard-Based Concurrent Problem Solving, Technical Report, KSL-86-41, Knowledge Systems Laboratory, Dept. of Computer Science, Stanford Univ. Also in 1986. Proceedings of the Defense Advanced Research Projects Agency Expert Systems Workshop, 142–145. Arlington, Va.: Science Applications International Corp.

Nii, H. P., and Rice, J. 1989. Signal Understanding and Problem Solving: A Concurrent Approach to Soft-Real-Time Systems, Technical Report, KSL-89-73, Heuristic Programming Project, Dept. of Computer Science, Stanford Univ. Also in 1989. Proceedings of the Twenty-Third Asilomar Conference on Signals, Systems, and Computers. San Jose, Calif.: Maple. Forthcoming.

Nii, H. P.; Aiello, N.; and Rice, J. 1988a. CAGE and Poligon: Two Frameworks for Concurrent Problem Solving, Technical Report, KSL-88-02, Heuristic Programming Project, Dept. of Computer Science, Stanford Univ.

Nii, H. P.; Aiello, N.; and Rice, J. 1988b. Experiments on Cage and Poligon: Measuring the Performance of Parallel Blackboard Systems, Technical Report, KSL-88-66, Heuristic Programming Project, Dept. of Computer Science, Stanford Univ. Also in 1989. *Readings in Distributed Artificial Intelligence,* Volume 2, eds. M. N. Huhns and L. Gasser, San Mateo, Calif.: Morgan Kaufmann. Forthcoming.

Noble, A. 1988. ELMA Programmer's Guide, Technical Report, KSL-88-41, Heuristic Programming Project, Dept. of Computer Science, Stanford Univ.

Noble, A., and Rogers, E. 1988. AIRTRAC Path Association: Development of a Knowledge-Based System for a Multiprocessor, Technical Report, KSL-88-41, Heuristic Programming Project, Dept. of Computer Science, Stanford Univ.

Okuno, H., and Gupta, A. 1987. Parallel Execution of OPS5 in QLISP, Technical Report, KSL-87-43, Heuristic Programming Project, Dept. of Computer Science, Stanford Univ.

Okuno, H.; Osato, N.; and Takeuchi, I. 1987. Firmware Approach to Fast Lisp Interpreter, Technical Report, STAN-CS-87-1184, Heuristic Programming Project, Dept. of Computer Science, Stanford Univ.

Rice, J. 1989. The Design and Implementation of Poligon, a High-Performance, Concurrent Blackboard System Shell, Technical Report, KSL-89-37, Heuristic Programming Project, Dept. of Computer Science, Stanford Univ.

Rice, J. 1988a. The Advanced Architectures Project, Technical Report, KSL-88-71, Heuristic Programming

Project, Dept. of Computer Science, Stanford Univ.

Rice, J. 1988b. The Elint Application on Poligon: The Architecture and Performance of a Concurrent Blackboard System, Technical Report, KSL-88-69, Heuristic Programming Project, Dept. of Computer Science, Stanford Univ. Also in 1989. Proceedings of the Eleventh International Joint Conference on Artificial Intelligence, 212–217. Menlo Park, Calif.: International Joint Conferences on Artificial Intelligence, Inc..

Rice, J. 1988c. Problems with Problem Solving in Parallel: The Poligon System, Technical Report, KSL-88-04, Heuristic Programming Project, Dept. of Computer Science, Stanford Univ. Also in 1988. Proceedings of the Third International Conference on Supercomputing, 25–34. Boston, Mass.: International Supercomputing Institute. Also in 1989. *Artificial Intelligence, Simulation, and Modelling,* ed. Lawrence Widman, 231–253. New York: Wiley.

Rice, J. 1986a. Poligon, A System for Parallel Problem Solving, Technical Report, KSL-86-19, Heuristic Programming Project, Dept. of Computer Science, Stanford Univ. Also in 1986. Proceedings of the Defense Advanced Research Projects Agency Expert Systems Workshop, 152–159. Arlington, Va.: Science Applications International Corp.

Rice, J. 1986b. The Poligon User's Manual, Technical Report, KSL-86-10, Heuristic Programming Project, Dept. of Computer Science, Stanford Univ.

Rice, J., and Aiello, N. 1989. See How They Run: The Architecture and Performance of Two Concurrent Blackboard Systems, Technical Report, KSL-89-08, Heuristic Programming Project, Dept. of Computer Science, Stanford Univ. Also in 1989. *Blackboard Architectures and Applications: Current Trends,* eds. V. Jagannathan and R. Dodhiawala, 153–178. San Diego, Calif.: Academic.

Saraiya, N. 1989. Design and Performance Evaluation of a Parallel Report Integration System, Technical Report, KSL-89-16, Heuristic Programming Project, Dept. of Computer Science, Stanford Univ.

Saraiya, N. 1988. A Shared-Memory Lisp Package for CARE, Technical Report, KSL-88-85, Heuristic Programming Project, Dept. of Computer Science, Stanford Univ.

Saraiya, N. 1986. AIDE: A Distributed Environment for Design and Simulation, Technical Report, KSL-86-56, Heuristic Programming Project, Dept. of Computer Science, Stanford Univ. Also in 1986. Proceedings of the Defense Advanced Research Projects Agency Expert Systems Workshop, 185–190. Arlington, Va.: Science Applications International Corp.

Schoen, E. 1986. The CAOS System, Technical Report, KSL-86-22, Heuristic Programming Project, Dept. of Computer Science, Stanford Univ. Also in 1986. Proceedings of the Defense Advanced Research Projects Agency Expert Systems Workshop, 160–170. Arlington, Va.: Science Applications International Corp.

Stanford University. 1988. Expert Systems on Multiprocessor Architectures: Phase One Final Report, RADC-TR-88-187, Rome Air Development Center.

Other Publications

Abelson, H., and Sussman, G. 1983. *Structure and Interpretation of Computer Programs*. Cambridge, Mass.: MIT Press.

Engelmore, R., and Morgan, T. 1988. *Blackboard Systems*. Reading, Mass.: Addison-Wesley.

Gabriel, R., and McCarthy, J. 1984. Queue-Based Multi-processing Lisp. In Proceedings of the ACM Symposium on Lisp and Functional Programming, 25–44. New York: ACM Press.

Gupta, A. 1986. Parallelism in Production Systems. Ph.D diss., Dept. of Computer Science, Carnegie-Mellon Univ.

Nii, H. P., and Aiello, N. 1979. AGE: A Knowledge-Based Program for Building Knowledge-Based Programs, Technical Report, HPP-79-4, Heuristic Programming Project, Dept. of Computer Science, Stanford Univ. Also in 1979. Proceedings of the Sixth International Joint Conference on Artificial Intelligence, 645–655. Menlo Park, Calif.: International Joint Conferences on Artificial Intelligence, Inc.

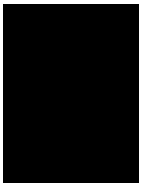
Nii, H. P.; Feigenbaum, E.; Anton, J.; and Rockmore, A. 1982. Signal-to-Symbol Transformation: HASP/SIAP Case Study, Technical Report, HPP-82-6, Heuristic Programming Project, Dept. of Computer Science, Stanford Univ. Also in *AI Magazine* 3(2): 23–35.

Selig, L. 1987. An Expert System Using Numerical Simulation and Optimization to Find Particle Beam Line Errors, Technical Report, KSL-87-36, Heuristic Programming Project, Dept. of Computer Science, Stanford Univ.

Williams, M.; Brown, H.; and Barnes, T. 1984. TRICERO Design Description, ESL Inc.

Acknowledgments

The author gratefully acknowledges the support of the following funding agencies for this project: The Defense Advanced Research Projects Agency/Rome Air Development Center under contract F30602-85-C-0012, the National Aeronautics and Space Administration under contract NCC 2-220, and Boeing Computer Services under contract W-266875.



James Rice has been a researcher in Stanford University's Knowledge Systems Laboratory since 1985, concentrating on concurrent problem-solving models, particularly the Poligon system. He received his B.Sc. in Cybernetics at the University of Reading in England in 1981. He worked for four years at SPL Int., a software house, on expert system tools and the United Kingdom's first blackboard system, MXA.