Final Report on Supercomputer Research

15 November 1983 to 31 May 1988

Ellen P. Douglas, Alan R. Houser, C. Roy Taylor, Editors

June 1989
CMU-CS-89-157

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This research was sponsored by the Defense Advanced Research Projects Agency, DoD, through DARPA order 4864, and monitored by the Space and Naval Warfare Systems Command under contract N00039-85-C-0134. Views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or of the United States Government.

Abstract

This report documents DARPA-supported supercomputer research in Carnegie Mellon University's Computer Science Department during the period 15 November 1983 through 30 September 1987, extended to 31 May 1988. Each chapter discusses one of four major research areas. Sections within each chapter present the area's general context, the specific problems addressed, our contributions and their significance, and an annotated bibliography.

The research areas and their main objectives are:

- *SuperComputer Workbench [SCW]*: Develop a multiprocessor operating system, programming environment, and instrumentation environment to support multiprocessor computing research.
- *Systolic Array Machine [SAM]*: Develop a powerful computational engine using systolic architectures and interconnections tailored to specific tasks.
- *Production System Machine [PSM]*: Explore the use of parallel architectures for production systems and develop a machine especially for production systems.
- *Command Action Team [CAT]*: Continue work on a knowledge-based expert system designed to assess and monitor threats to a carrier group.

Table of Contents

1. INTRODUCTION

This report documents parallel processing research conducted by Carnegie Mellon University's Computer Science Department (CMU-CSD). The Information Processing Techniques Office of the Defense Advanced Research Projects Agency (DARPA) supported this work during the period 15 November 1983 through 30 September 1987, extended to 31 May 1988.

The remainder of this chapter describes our research scope and the CMU-CSD research environment. Chapters 2 through 5 then present in detail our four major research areas: the SuperComputer Workbench, the Systolic Array Machine, the Production System Machine, and the Command Action Team (CAT) project. Sections in each chapter present the area's general research context, the specific problems we addressed, our contributions and their significance, and an annotated bibliography.

The bibliographies present selected references that reflect the scope and significance of CMU's contributions to basic and applied computer science. Wherever possible, particularly for key reports, we have included abstracts. Also, publication dates serve as a reasonable indicator of progress in the various problem areas. CSD Technical Report dates exhibit the closest correlation with temporal progress and the report text frequently reappears later in the more accessible archival literature.

1.1 Research scope

We organize the research reported here under four major headings. These interrelated categories and their major objectives are:

- *SuperComputer Workbench [SCW]*: Develop a multiprocessor operating system, programming environment, and instrumentation environment to support multiprocessor computing research.
- *Systolic Array Machine [SAM]*: Develop a powerful computational engine using systolic architectures and interconnections tailored to specific tasks.
- *Production System Machine [PSM]*: Explore the use of parallel architectures for production systems and develop a machine especially for production systems.
- *Command Action Team [CAT]*: Continue work on a knowledge-based expert system designed to assess and monitor threats to a carrier group.

1.2 The Research environment

Research in the CMU Computer Science environment tends to be organized around specific experimental systems aimed at particular objectives, e.g. the demonstration of a systolic array machine or the design and fabrication of a parallel interpreter. This report describes several such activities. Sometimes the creation and demonstration of a system is itself an appropriate scientific objective. At other times, some level of system

performance constitutes the scientific goal. Thus our work tends to emphasize concept demonstration rather than system engineering. These research systems provide a convenient way to discuss and even to organize the projects at CMU-CSD. They are not always, however, ends in themselves.

A major strength of the Carnegie Mellon University environment lies in the synergy resulting from close cooperation and interdependence among varied research efforts, despite their diverse foci. For example, our basic research in image understanding, supported by DARPA under a separate contract, has an extraordinarily large appetite for computational cycles. Work in low-level vision and applied domains such as road following and obstacle avoidance have put the high computational throughput and novel architecture of the SAM project's Warp machine to good use. Likewise, the SAM project has benefitted from the close relationship with researchers who actually apply the Warp machine to real tasks. This inter-project collaboration significantly influenced Warp, from the conceptual level of program partitioning models to the pragmatic level of rapid feedback regarding performance criteria and bottlenecks.

We have no administrative structure that corresponds to our organization of effort. We consist simply of faculty, research scientists, and graduate students of the Computer Science Department, with the facilities support divided into an Engineering Laboratory and a Facilities Software Group. The rest of the organization is informal. This organizational style minimizes the barriers between efforts and promotes the kind of interactions and synergy reflected in the work distribution shown in Table 1-1.

	Number of Areas	Mach	Warp	PSM/CAT
Roberto Bisiani	2	x	x	
Scott Fahlman	2	x		x
Lanny Forgy	1			x
Thomas Gross	1		x	
Takeo Kanade	2	x	x	
H.T. Kung	3	x	♦	x
John McDermott	1			x
Allen Newell	2	x		♦
Rick Rashid	1	x		
Raj Reddy	3	x	x	x
Zary Segall	1	x		
Albert Spector	1	x		
Daniel Siewiorek	1	♦		
Howard Wactlar	1	♦		

x = Active research in this area
♦ = Responsible for area
Faculty participating, total = 14

Figure 1-1: Distribution of faculty effort

2. SUPERCOMPUTER WORKBENCH

Multiple-processor computer architectures have emerged as a viable response to the challenge of providing sufficient computing power for computationally-intensive applications. When such architectures were first developed, however, they typically suffered from inadequate software support. Early multiprocessor systems normally featured a poor or non-existent programming environment and an operating system that did not take full advantage of the hardware's multiple-processor resources.

The goal of the SuperComputer Workbench project has been to provide software support tools specifically designed for shared-memory, multiprocessor architectures. Our work has produced two such support tools:

- A distributed multiprocessor operating system (Mach)
- A host software development and instrumentation environment (PIE)

These tools support researchers in producing, evaluating, and using multiprocessor computing systems. The Mach operating system permits full utilization of multiprocessor resources, an efficient mechanism for sharing memory, and full UNIX compatibility. Developers can port Mach to a variety of different architectures, as it supports single multiprocessor hosts, distributed computer networks, and individual workstations. The PIE programming and instrumentation environment provides tools for writing and debugging efficient multiprocessor programs and for evaluating them for their ability to fully exploit the underlying hardware and software.

2.1 Challenges in Multiprocessor and Distributed Operating System Research

2.1.1 Previous multiprocessor operating systems

Before Mach, there had been several efforts in developing multiprocessor operating systems. However, each has suffered from limitations in functionality, performance, or usability. None has approached our goal of a general-purpose, multiprocessor, distributed operating system.

Previous multiprocessor operating systems have generally fallen into one of three categories:

- *Simple operating systems providing minimal functionality*—These systems, such as the Cosmic Cube and the Butterfly, do not address operating system issues directly. Typically, they provide only basic functions required to use the hardware. Users must often cross-compile programs on a different machine, then download to execute. Such systems make it possible to use the target machine, but their user environments are less than desirable.
- *Uniprocessor operating systems with simple modifications for use in a multiprocessor environment*—Numerous other multiprocessor operating sys-

tems represent modifications of pre-existing uniprocessor systems. VMS has been extended to run in a dual-processor configuration and several UNIX[1] versions have been modified to run on multiprocessors. These systems usually run in a master/slave configuration and are not realistically extensible to large multiprocessors.

- *Completely new operating systems typically designed to run on a specific type of multiprocessor*—Where completely new multiprocessor operating systems have been built, they were frequently accompanied by inadequate user environments and were difficult to use. Intel's IMax operating system for the 432 exemplifies such a system.

During this contract period, we began to lay the foundations for a general-purpose, multiprocessor, software support environment that does not suffer the limitations common to earlier efforts. Our current Mach operating system, built on the experience of previous research efforts, forms the prototype nucleus of such an environment.

Our previous work produced Accent, a uniprocessor distributed operating system [Rashid 86a]. Mach was conceived as an Accent-like operating system that would provide multiprocessor functionality and complete UNIX compatibility. Mach was designed to better accommodate the kind of general purpose, shared-memory multiprocessors that appear destined to succeed traditional general purpose uniprocessor workstations and timesharing systems.

2.1.2 The Accent distributed operating system

Accent was organized around the notion of a protected, message-based interprocess communication facility integrated with copy-on-write virtual-memory management. Access to all services and resources, including the process and memory management services of the operating system kernel itself, was provided through Accent's communication facility. This design allowed completely uniform access to resources throughout the network. It also provided that access to kernel-provided services was indistinguishable from access to process-provided resources (with the exception of the interprocess communication facility itself).

Accent went beyond demonstrating the feasibility of the message passing approach to building a distributed system. Experience with Accent showed that a message-based network operating system, properly designed, can compete with more traditional operating system organizations. The advantages of this approach are system extensibility, protection and network transparency.

While Accent demonstrated the feasibility of a network operating system, it represented only an early step toward our long-term goal of a distributed, portable, multiprocessor operating system. Accent was a distributed *uniprocessor* operating system.

---

[1] UNIX is a trademark of AT&T Bell Laboratories.

It did not have the necessary process management facilities to take advantage of multiprocessor architectures. Accent was largely *architecture dependent*, running on a network of 150 PERQ workstations.[2] Portability to other architectures would have required extensive modifications to the Accent kernel. Finally, Accent's slow "UNIX compatibility" package was ineffective in absorbing the ever-burgeoning body of UNIX-developed software.

2.2 A UNIX-Compatible Distributed Multiprocessor Operating System

A major reason that Accent never achieved widespread acceptance was its lack of true UNIX compatibility. For Mach to survive, UNIX compatibility was essential. To insure UNIX compatibility, we evolved Mach directly from the 4.2 BSD UNIX kernel. As we developed Mach features, we replaced existing UNIX features with our Mach implementations. This strategy had several advantages. It allowed us to maintain a working kernel throughout the Mach development process. It simplified the debugging of new kernel features. It also allowed us to incorporate into Mach new UNIX features developed outside CMU, such as the 4.3 BSD UNIX distribution and MIT's **X** window manager.

Our Mach design combines several low-level kernel abstractions with unique approaches in virtual memory implementation and interprocess communication. After presenting an overview of the current Mach operating system, we will discuss the building-block abstractions that form the basis of the Mach kernel design. We will then discuss Mach's virtual memory and interprocess communication facilities, both separately and as they together provide such Mach features as copy-on-write message passing and flexible memory sharing.

2.2.1 Mach system overview

Mach currently runs on a variety of architectures, including the entire VAX family of uniprocessors and multiprocessors, the IBM RT PC, the Sun 3, the Encore MultiMax, and the Sequent Balance 21000. Mach provides key functionality for parallel system software development, including

- Ability to allocate and manage large, sparse virtual memories
- A parallel multiprocessor scheduler with the ability to spawn new control threads cheaply within an address space
- Mechanisms for flexible memory-sharing among multiprocessor tasks
- Support for fine granularity synchronization
- Transparent communication between tasks running on both tightly- and loosely-coupled processor nodes

Our long-term goal is to have user-state server programs that reside outside the Mach

[2]PERQ is a trademark of PERQ System Corporation.

kernel perform traditional operating system functions. During this contract period, we set the foundation for such an implementation. Our approach results in increased modularity and protective isolation among unrelated operating system functions. It also provides for a natural function decomposition in a multiprocessor system.

2.2.2 Building-block abstractions

The primary purpose of the Mach kernel is to provide an execution environment for user tasks and an interprocess communication (IPC) facility that allows user tasks to share data and resources. Our kernel design provides a minimal system abstraction set, extended from Accent, that forms the basic building blocks for a distributed multiprocessor computing environment:

- A *message* is a typed collection of data objects and consists of a fixed size header and a variable length body. Messages may be any size and may contain typed pointers to data outside the contiguous portion of the message body.
- A *port* is a kernel-protected queue for messages. At any given time, the maximum length of a port is fixed, although that fixed length can be changed. Tasks refer to ports through port capabilities. There are three kinds of port capabilities: send access, receive access, and ownership. Tasks obtain capabilities to ports only by receiving such capabilities in messages.
- A *task* represents the basic resource allocation unit, comprising a paged address space and access to system resources. A task may contain a single thread or multiple threads executing in parallel.
- A *thread* is the basic unit of computation, executing within a task. Threads may send and receive messages according to their access rights. When creating a thread, the kernel also creates a port, the *thread port*, to represent the thread. Messages sent to a thread port can alter the associated thread's state.
- A *process* is a thread operating within a task context. A standard UNIX process is equivalent to a Mach task with a single control thread.
- A *memory object* is a kernel-managed data repository. Memory objects can be created, destroyed, read or written. Backing storage for a memory object is determined by its type: permanent disk, temporary disk, physical memory, or port. Permanent disk memory objects are used to manage files. Temporary disk objects are used to back newly created virtual storage on disk and to shadow copy-on-write data. Physical memory objects are used to manage devices that operate on physical memory. Port memory objects provide copy-on-reference network access to data and any other on-demand creation or control of information.

A thread executes in the context of exactly one task; however, any number of threads may execute within a single task. Theoretically, all threads execute in parallel. This ability to execute multiple threads simultaneously within a task is the key feature in

Mach's multiprocessor capability. Our multiprocessor scheduler and parallel thread execution capabilities allow Mach to take full advantage of the computing capacity of multiprocessor architectures.

The Mach kernel can be viewed as a task with its own 2^{32} byte paged virtual address space and port access rights. The Mach network operating system is implemented as a collection of tasks running above the Mach kernel using the Mach IPC facility to communicate. Port capabilities are used to represent task-provided services, resources and data structures. As such, port capabilities serve a role in Mach similar to object capabilities in systems such as Hydra or StarOS. Interprocess interfaces in Mach are defined using MatchMaker, an object-oriented interface definition language developed for Accent [Jones and Rashid 86]. These interfaces are compiled into remote procedure call (RPC) stubs that use the Mach message passing primitives for communication and control.

2.2.3 A portable virtual memory management system

Proliferating hardware memory structures, with their varying requirements for virtual memory management, have hindered operating system portability. UNIX systems traditionally address the problem of virtual memory management portability by restricting the facilities provided and basing implementations for new memory management architectures on versions already done for previous systems. As a result, existing versions of UNIX, such as Berkeley 4.3 BSD, offer little in the way of virtual memory management other than simple paging support. Versions of Berkely UNIX on non-VAX hardware, such as SunOS on the Sun 3 and ACIS 4.2 on the IBM RT PC, actually simulate internally the VAX memory mapping architecture—in effect treating it as a machine-independent memory management specification.

Our goal was to implement a memory management system that would be readily portable to multiprocessor computing engines as well as to traditional uniprocessors. We designed our system by dividing Mach's virtual memory management code into machine *dependent* and machine *independent* sections [Rashid et al. 87]. Machine dependent code implements only those operations necessary to create, update and manage the hardware required for data structure mapping. All important virtual memory information is maintained by machine independent code. By clearly defining and organizing the machine dependent portion of the kernel, we greatly decrease the amount of time and effort required to port Mach to other architectures.

2.2.4 Interprocess communication

UNIX interprocess communication has never been flexible enough to easily build distributed systems. While advanced versions of UNIX, such as 4.3 BSD, continue to add communication mechanisms, the problems that distributed systems must address are glossed over. For example, internet domain sockets use a global machine-specific naming convention based on IP address, with a lack of location-independence and protection.

To address the problems associated with building distributed systems, we designed Mach to provide a flexible interprocess communication facility through:

- A capability-based interprocess communication paradigm
- Typed message data
- Transparent extension of local communication into a network through message servers
- An interface language, Matchmaker, that generates client/server interfaces
- Integration with virtual memory management for efficient transfer of large messages

The Mach kernel itself has no knowledge of networks. The kernel doesn't have to distinguish between messages passed between tasks on the same host and messages passed over a network. Network message servers transparently extend communication over a network. A message sent to a port on a remote machine actually is sent to a network server on the sending host which then forwards the message over the network. The forwarding operation is transparent (and undetectible) to both the sender and the receiver.

In addition to simply extending the IPC paradigm to the network, network servers may participate in data type conversion and provide secure network transmission. By providing this functionality outside the kernel, Mach allows a host more flexibility in choosing data type representations, the amount or type of security to be used on a network and even the protocols to use for network transmission.

Matchmaker, our interprocess specification language, handles details of interprocess communication between different machine architectures and languages [Jones and Rashid 86]. Developed for Accent, Matchmaker enables a program to specify an interface between a client and server. Matchmaker allows a programmer to create a distributed program without worrying about the details of sending messages or type conversion between different machines.

Finally, the IPC mechanism makes use of the virtual memory system to make virtual, rather than physical, copies of large messages. This mechanism allows large amounts of data to be sent copy-on-write. Data is not copied from its original location unless a task writes to it. Our IPC facility is an especially important feature since a task usually only reads data, making data copying unnecessary.

2.2.5 Integrating memory and communication

Mach combines virtual memory management and interprocess communication so that data may be transferred by memory mapping rather than data copying. Initially employed by Accent, by-value data transfer semantics are obtained by transferring message data with copy-on-write memory mapping, allowing multiple processes to access the same area of memory. Memory is not physically copied unless a process attempts to write to that memory space. Copy-on-write memory mapping saves the com-

putational expense of making a physical copy of a memory region every time that region is accessed.

Figure 2-1 details schematically a copy-on-write data transfer between two Mach processes. At time *t0* process *A* sends a message containing a large amount of data (for example, an eight Mbyte pixel array gathered by a video camera) to communication port *P1*. When *A* sends the message, Mach marks corresponding memory areas in the address spaces of both *A* and the kernel (indicated by cross-hatched areas in their memory maps) "copy-on-write". At time *t1*, process *B* retrieves the message and Mach then moves the image data, again copy-on-write, into *B*'s address space. At no time is the data actually copied during these operations. A page-by-page copy would be performed only if *A* or *B* attempted to change parts of the transferred image.

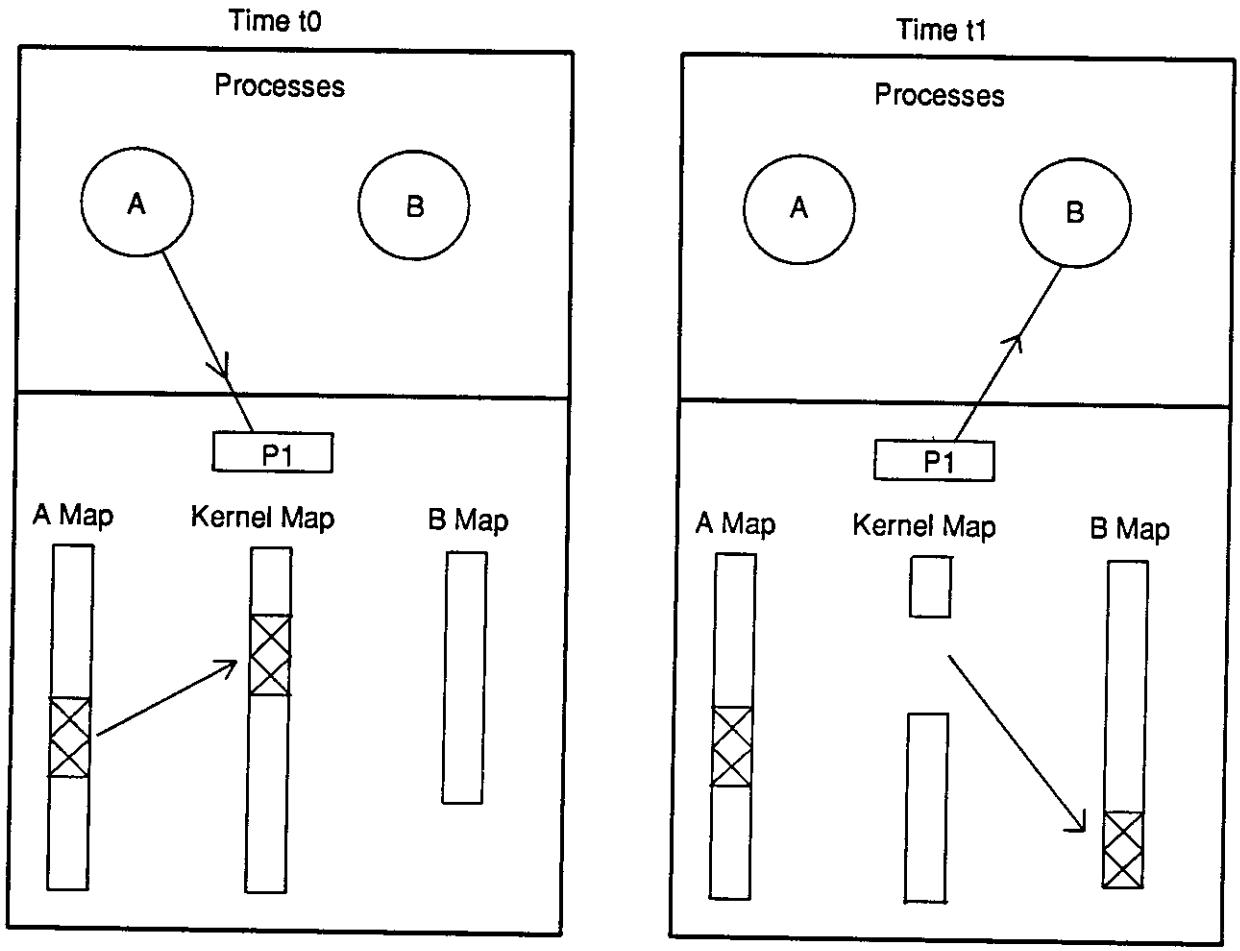

Figure 2-1: Transferring data copy-on-write in Mach

By using copy-on-write mapping to transfer large data objects, Mach provides:

- Ability to transfer data objects in their natural size, up to and including the size of a process address space (as much as 2^{31} bytes), unhindered by artificial message size limits
- Simple, mapped access to data such as files by making data objects directly addressable as regions of process address space, as in traditional P-MAP style file mapping
- Better utilization of both physical memory and backing storage through greater sharing between processes.

In addition to the ease of sharing memory, the integration of interprocess communication and virtual memory management provides another key Mach feature. Passing data by value in messages, Mach gains the advantages of simple communication semantics, including the ability to transparently extend communication in a large multiprocessor or onto a network with an absence of unintended side effects.

2.2.6 Sharing memory

Mach's memory sharing, through the copy-on-write message sharing provided by the IPC facility, works well for communicating tasks that require the protection of by-value message passing. It is also appropriate for applications with components intended for distribution over a local area network or loosely-coupled multiprocessor. On large, shared-memory multiprocessors, however, Mach provides two ways that processes can communicate more directly and efficiently:

- Many threads may directly share a single task address space
- A task may specify regions of its virtual address space as read/write inheritable to tasks it creates.

The ability to share memory between tasks allows for the sharing of global data structures at a page (4-8 Kbyte) level without incurring large performance penalties. It also provides access to the parallelism of memory access between tasks that require separate protection domains for other reasons. By including constructs for message passing, structured memory sharing between tasks and unrestricted sharing between threads, Mach can accommodate a range of multiprocessor architectures from loosely-coupled multiprocessors and networks to tightly-coupled machines with low latency memory access.

Associated with each region of memory in a task's address space are a *current* and *maximum* protection. The current protection specifies what rights a task has to that memory in the form of a combination of read, write, and execute rights. The maximum protection describes the greatest set of protection rights a task may have. When a region of memory is allocated, a task has a maximum and current protection allowing all privileges. A task's current protection rights may be changed up to those specified by its maximum protection, or its maximum protection rights may be decreased. Separating maximum protection from current protection enables a parent task to provide protected (e.g. read-only) access to a portion of its memory.

2.2.7 Extending the kernel

Mach forms a single-machine operating system kernel, with all of its operations executing on a single processor or a single, shared-memory multiprocessor. The kernel's IPC facility, for example, supports only communication between tasks on the same machine. Our strategy for extending Mach to serve as a network operating system kernel was to design its abstractions, IPC facility, and virtual memory support to be transparently extensible by user-state tasks. This strategy permits server tasks, which are typically easier to prototype and develop than an operating system kernel, to provide traditional operating system functions.

Several examples illustrate the flexibility of Mach's primitives in providing traditional operating system functions. Rather than provide kernel support for network communication, we have implemented network server tasks that transparently extend Mach's IPC facilities between machines. We have developed an experimental file system that relies only on the kernel's memory object facilities for support. A file server task builds user file abstractions, such as directories, on top of the memory object. The file system server also uses network-transparent IPC to cooperate with other file servers in providing network-transparent remote file access. Finally, CMU's Camelot distributed transaction processing system is based on Mach's network interprocess communication facilities.

2.3 A Multiprocessor Programming Environment

2.3.1 Programming multiprocessors for performance

Historically, programming a parallel processor application has required a detailed knowledge of multiprocessor architecture and operating systems. The programmer had only rudimentary tools for creating parallel applications. Moreover, creating a correct parallel program has not been the end of the task. Often the only reason for developing a parallel program is for real-time performance. The difficult task of performance debugging and interpreting feedback in the context of a rudimentary program environment required an even more specialized and highly knowledgeable programmer.

A key element in a parallel debugging environment is an ability to collect data, through instrumentation support, on both the parallel hardware utilization (e.g., caches, buses, memories) and on system and application software performance (e.g., scheduling, resource management, virtual memory). In previous parallel programming research, such as the Cm* project, we found extensive instrumentation support critical to both the programming itself and to producing reliable, supportable, and maintainable software. There are two important reasons to integrate instrumentation with a parallel programming environment:

- In a distributed parallel system, it is absolutely essential to assure target performance levels. System developers therefore need a programming environment that can support them in making performance-related decisions

regarding program structure. Such support involves obtaining estimated performance data at program development time. An environment that provides this capability to program for performance must build on an instrumented base, that is, on a highly observable virtual machine (hardware+OS).

- Debugging a distributed or parallel program is far more complex than debugging a sequential program. The parallel application developer faces numerous hurdles, including a need to comprehend in detail the multiprocessor architecture and operating system, as well as a need to manually map intricate parallel algorithms onto parallel machines.

During the contract period we developed hardware instrumentation facilities for shared-bus, shared-memory multiprocessors, such as the Encore MultiMax and Sequent Balance. The facilities include several special-purpose hardware monitors under program control. These mechanisms provide the hardware monitoring basis for PIE, our Programming and Instrumentation Environment, and allow us to improve parallel program performance through detailed analysis of their CPU, cache, bus, and memory requirements.

2.3.2 A Programming and instrumentation environment

Our research in generating efficient parallel programs emphasizes two strategies, both embodied in our programming and instrumentation environment. The first approach involves avoiding performance bottlenecks through combining a coding methodology and performance prediction models to detect potential problems before undertaking extensive coding. The second approach, performance debugging, applies the concept of programming-for-observability.

During the contract period we continued to develop a programming and instrumentation environment (PIE) specifically tailored to parallel programming needs [Segall and Rudolph 85]. Our environment includes tools for constructing, instrumenting, and measuring parallel programs. PIE comprises a multilevel program development environment that assists the user in organizing, writing, and managing efficient parallel software and a tool set geared toward instrumenting such software for performance debugging.

The PIE environment consists of the following set of tools:

- MPC—a multiprocessor **C** language: MPC is a **C** preprocessor that converts special language constructs into **C** program systems. MPC resolves data consistency problems and handles physical synchronization and communication demands of multiprocessor code.
- PIEMACS is a syntax and semantics-based editor that automatically extracts the development time data about the target program and assists in instrumenting it for the run-time monitoring process.
- PIEMON (the PIE monitor) supports collection and storage of run-time event data via hardware instrumentation sensors.

- PIEMAN (PIE manager) is a relational database which intelligently integrates development-time with run-time information.
- PIESCOPE is a graphical user interface which allows the programmer to view the development and execution of an MPC program.

PIE addresses the issues of performance debugging and programming for observability. The PIE environment tools aid the parallel programmer in both generating efficient multiprocessor programs and observing the execution of those programs for debugging and improvement of program efficiency. Additionally, PIE allows the programmer to write parallel programs without having to worry about the details of low-level process synchronization and communication. The PIE environment is designed to take the burden of these details off the user. The user can then concentrate on algorithm design and implementation to a greater degree than previously possible.

2.4 Bibliography

[Baron et al. 85] Baron, R., R. Rashid, E. Siegel, A. Tevanian, and M. Young.
Mach 1: an operating system environment for large scale multiprocessor applications.
IEEE Software Special Issue, July, 1985.

[Fitzgerald and Rashid 85]
Fitzgerald, R. and R. Rashid.
The integration of virtual memory management and interprocess communication in Accent.
Technical Report CMU-CSD-85-164, Computer Science Department, Carnegie Mellon University,
September, 1985.
The integration of virtual memory management and interprocess communication in the Accent network operating system kernel is examined. The design and implementation of the Accent memory management system is discussed and its performance, both on a series of message-oriented benchmarks and in normal operation, is analyzed in detail.

[Gregoretti and Segall 86]
Gregoretti, F. and Z. Segall.
Programming for observability support in a parallel programming environment.
In *14th Annual Computer Science Conference*, ACM, February, 1986.
The programming for observability concept for performance/correctness debugging in a parallel programming environment is introduced. The design, first implementation, and evaluation of the required language and system support is presented. A two dimensional, multilevel, integrated monitoring system is described. Distinction is made between the monitoring mechanism, monitoring policies and their implementation tradeoffs. PIEMON-1, an initial implementation of the presented design is outlined and evaluated.

[Jones and Rashid 86]
Jones, M. and R. Rashid.
Mach and Matchmaker: kernel and language support for object-oriented distributed systems .
In *1st Annual OOPSLA Conference*, ACM, October, 1986.
Mach, a multiprocessor operating system kernel providing capability-based interprocess communication, and Matchmaker, a language for specifying and automating the generation of multi-lingual interprocess communication interfaces, are presented. Their usage together providing a heterogeneous, distributed, object-oriented programming environment is described. Performance and usage statis-

tics are presented. Comparisons are made between the Mach/Matchmaker environment and other systems. Possible future directions are examined.

[Rashid 86a] Rashid, R.
From RIG to Accent to Mach: the evolution of a network operating system.
In *Proceedings of the Fall Joint Computer Conference*, ACM/IEEE, November, 1986.
This paper describes experiences gained during the design, implementation and use of the CMU Accent Network Operating System, its predecessor, the University of Rochester RIG system and its successor CMU's Mach multiprocessor operating system. It outlines the major design decisions on which the Accent kernel was based, how those decisions evolved from the RIG experiences and how they had to be modified to properly handle general purpose multiprocessors in Mach. Also discussed are some of the major issues in the implementation of message-based systems, the usage patterns observed with Accent over a three year period of extensive use at CMU and a timing analysis of various Accent functions.

[Rashid 86b] Rashid, R.
Threads of a new system.
Unix Review 4(8):37-49, 1986.
The Department of Defense, anxious for better multithreaded application support, has funded the development of Mach, a multiprocessor operating system for UNIX applications.

[Rashid et al. 87] Rashid, R.F., A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew.
Machine independent virtual memory management for paged uniprocessor and multiprocessor architectures.
In *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems*, ACM, February, 1987.
Also appeared as tech-report CMU-CSD-87-140.
This paper describes the design and implementation of virtual memory management within the CMU Mach Operating System and the experiences gained by the Mach kernel group in porting that system to a variety of architectures. As of this writing, Mach runs on more than half a dozen uniprocessors and multiprocessors including the VAX family of uniprocessors and multiprocessors, the IBM RT PC, the SUN 3, the Encore Multimax, the Sequent Balance 21000 and several experimental computers. Although these systems vary considerably in the kind of hardware support for memory management they provide, the machine-dependent portion of Mach virtual memory con-

sists of a single code module and its related header file. This separation of software memory management from hardware support has been accomplished without sacrificing system performance. In addition to improving portability, it makes possible a relatively unbiased examination of the pros and cons of various hardware memory management schemes, especially as they apply to the support of multiprocessors.

[Segall and Rudolph 85]
Segall, Z. and L. Rudolph.
PIE- a programming and instrumentation environment for parallel processing.
Technical Report CMU-CSD-85-128, Computer Science Department, Carnegie Mellon University,
April, 1985.

The issues of efficient development of performance efficient parallel programs is explored. Programming and Instrumentation Environment (PIE) for Parallel Processing system's concepts, designs, and preliminary implementation results are presented. The key goal in PIE is semi-automatic generation of performance efficient parallel programs. In PIE, a system intensive rather that a programmer intensive programming environment is promoted for supporting users with different experience in parallel programming. Three levels of such support are provided, namely the Modular Programming Metalanguage, the Program Constructor, and the Implementation Assistant. In order to facilitate the task of parallel programming, each component employs a set of new concepts and approaches to integrate functionality with performance concerns. This paper presents the results of PIE 1, the first of a three phase project.

[Tevanian and Rashid 87]
Tevanian Jr., A. and R.F. Rashid.
Mach: A basis for future UNIX development.
Technical Report CMU-CS-87-139, Computer Science Department, Carnegie Mellon University,
June, 1987.

Computing in the future will be supported by distributed computing environments. These environments will consist of a wide range of hardware architectures in both the uniprocessor and multiprocessor domain. This paper discusses Mach, an operating system under development at Carnegie Mellon University, that has been designed with the intent to integrate both distributed and multiprocessor functionality. In addition, Mach provides the foundation upon which future UNIX development may take place in these new environments.

[Tevanian et al. 87a]
Tevanian Jr., A., R. Rashid, M.W. Young, D.B. Golub, M.R. Thompson, W. Bolosky, and R. Sanzi.
A Unix interface for shared memory and memory mapped filed under Mach.
In *Proceedings of the Summer USENIX Technical Exhibition*, USENIX, June, 1987.
This paper describes an approach to UNIX shared memory and memory mapped files currently in use at CMU under the Mach operating system. It describes the rationale for Mach's memory sharing and file mapping primitives as well as their impact on other system components and on overall performance.

[Tevanian et al. 87b]
Tevanian Jr., A., R.F. Rashid, D.B. Golub, D.L. Black, E. Cooper, and M.W. Young.
Mach threads and the Unix Kernel: the battle for control.
In *Proceedings of the Summer USENIX Technical Exhibition*, USENIX, June, 1987.
This paper examines a kernel implemented lightweight process mechanism built for the Mach operating system. The pros and cons of such a mechanism are discussed along with the problems encountered during its implementation.

[Young et al. 87] Young, M., A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron.
The duality of memory and communication in the implementation of a multiprocessor operating system.
In *Proceedings of the 11th Symposium on Operating System Principles*, ACM, November, 1987.
Mach is a multiprocessor operating system being implemented at Carnegie Mellon University. An important component of the Mach design is the use of memory objects which can be managed either by the kernel or by user programs through a message interface. This feature allows applications such as transaction management systems to participate in decisions regarding secondary storage management and page replacement.
This paper explores the goals, design and implementation of Mach and its external memory management facility. The relationship between memory and communication in Mach is examined as it relates to overall performance, applicability of Mach to new multiprocessor architectures, and the structure of application programs.

3. SYSTOLIC ARRAY MACHINE

Our objective in the Systolic Array Machine (SAM) project has been to demonstrate that we can build a useful supercomputer within both a short time period and a modest budget. In meeting this goal, we have developed the Warp machine [Annaratone et al. 87a]. Warp incorporates a systolic array of powerful, programmable cells, each capable of a 10 MFLOPS peak computing rate. In a typical configuration, the array comprises ten cells, thus offering a 100 MFLOPS aggregate computational bandwidth.

Warp's effectiveness results from a synergetic research strategy that simultaneously considers architecture, software, and applications. The Warp array's simple, linear topology supports several useful program partitioning models. In addition, each cell is highly programmable and has a large local memory. Together these features eliminate a need for the higher-dimensional connections that simpler systolic processors must employ to achieve equivalent power. With powerful cells, we need fewer of them to realize our performance goal. Warp complements its high cellular computation rate with correspondingly fast communication. The array's design provides high inter-cell bandwidths, while the host system provides high-speed external I/O. To deliver Warp's power into the programmer's hands, we developed a high-level language (W2) that provides detailed control down to cell-level parallelism, and an optimizing compiler that maps programs directly from W2 code to efficient machine instructions. Finally, we have facilitated user access to Warp's power by integrating the machine within UNIX as an attached processor, implementing a sizable application library that supports vision systems research, and developing general methods for mapping application problems onto the Warp array.

Our research has demonstrated the practicality of designing and building versatile, high-performance, systolic array computers. Warp's powerful array cells, fast communication, and user-accessible parallelism have extended its application domain substantially beyond that of previous designs. Programmability requires merely a physically larger machine and, given appropriate architectural support, does not degrade performance. We have, in fact, programmed Warp to execute well-known systolic algorithms—including matrix multiplication and convolution—as fast as special-purpose arrays employing comparable technology. Warp has also demonstrated high performance in diverse application areas, including low-level vision, signal processing, and scientific computing. As currently produced by our industrial partner, General Electric Corporation, Warp provides considerably more power and programmability than other machines of comparable cost.

3.0.1 System components

The Warp system, illustrated in Figure 3-1, has three major subsystems: processor array, interface unit, and host. The processor array performs the computation-intensive routines such as low-level vision routines or matrix operations. The interface unit (IU) handles input/output between array and host, and can generate memory addresses and control signals for the array. The host supplies data to and receives results from the

array. In addition, it executes those parts of the application programs that are not mapped onto the Warp array.

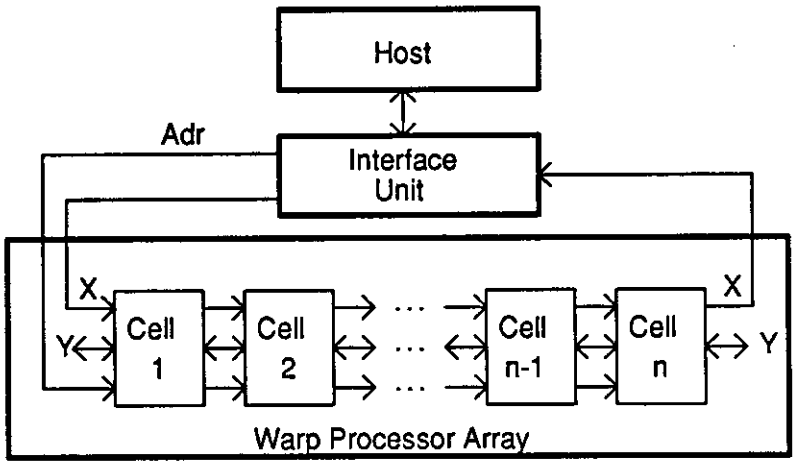

Figure 3-1: Warp system overview

The processor array is a linear systolic structure of identical Warp cells. Data flow through the cells on two communication channels (X and Y), and each cell's I/O bandwidth totals 20 Mword/s. The Y channel's direction is statically configurable, thus allowing bidirectional data flow. IU-generated control signals and local memory addresses propagate down the Adr channel.

Each cell is implemented as a programmable, horizontal micro-engine, with its own microsequencer and program memory for 8K 272-bit instructions. The cell data path, shown in Figure 3-2, includes a 32-bit floating-point multiplier (Mpy), a 32-bit floating-point adder (Add), two local memory banks for resident and temporary data (Mem), a queue for each inter-cell communication channel (XQ, YQ, and AdrQ), and a register file to buffer data for each floating-point unit (AReg and MReg). All these components are connected through a crossbar switch. Addresses for memory access can be computed locally by the address generation unit (AGU), or taken from the address queue (AdrQ).

The Warp host system, detailed in Figure 3-3, comprises a standard Sun-3 *workstation* that serves as master system controller and a VME-based *external host* multiprocessor, so named because it lies outside the workstation. The workstation provides a UNIX environment for application programs. The external host controls peripherals and provides a large memory for data the Warp array will process. It also transfers data to and from the Warp array and can perform certain data operations — corner-turning or scaling, for example— without the higher overhead that a complete operating system would entail.

Both the Warp cell and IU use off-the-shelf, TTL-compatible parts, and are each im-

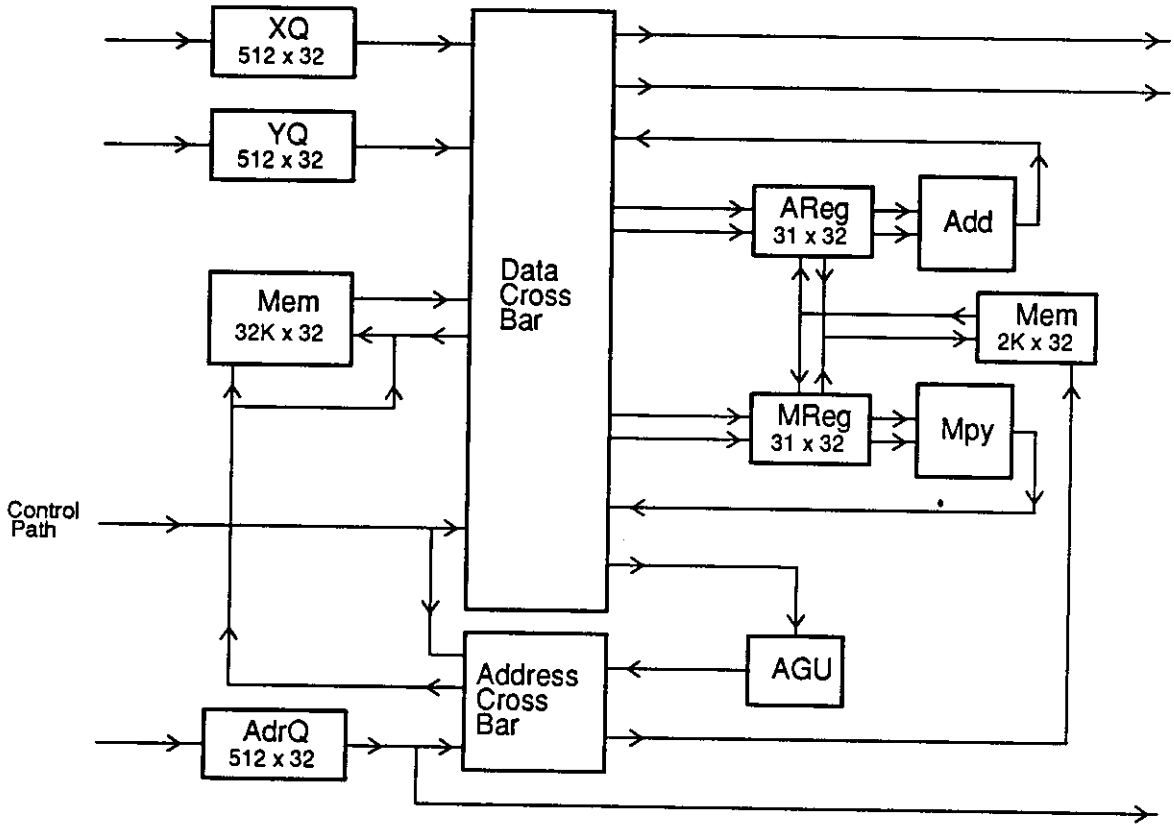

Figure 3-2: Warp cell data path

plemented on a 15"×17" board. The entire Warp machine, with the exception of the Sun-3, is housed in a single 19" rack, which also contains power supplies and cooling fans. The machine typically consumes about 1800W.

3.0.2 Chronology

We completed a two-cell Warp system at CMU in June, 1985, and then contracted two industrial partners to construct identical, 10-cell, wire-wrapped prototypes. GE delivered the first machine in February, 1986, and the Honeywell prototype arrived in June of that year. We next revised the design and reimplemented it with printed circuit (PC) technology to allow faster and more efficient production. Our revision also incorporated several architectural improvements. GE developed the PC version as a commercial product and delivered the first PC Warp machine to CMU in April, 1987. Design work for a single-chip Warp cell implementation began in 1986 with the collaboration of Intel Corporation.

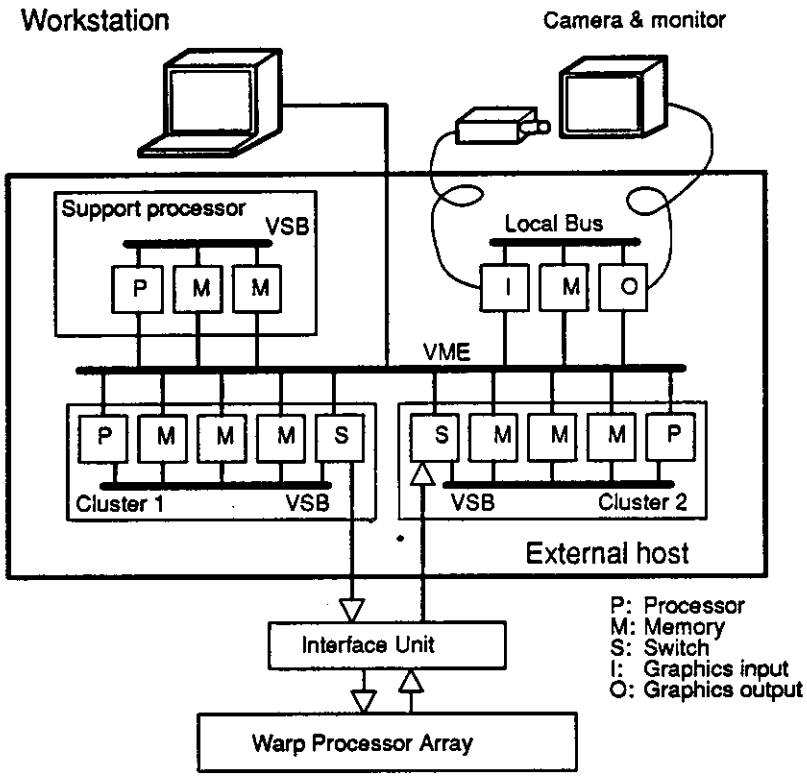

Figure 3-3: The Warp machine host system

3.0.3 Evaluation

We have evaluated Warp's architecture and compiler extensively, measuring several overall system performance factors and comparing them to other machines [Lam 87, Annaratone et al. 87a]. For applications in robot navigation, signal processing, scientific computation, and computer vision research, Warp is typically several hundred times faster than a VAX-11/780 class computer.

In the typical case with unidirectional data flow, the array's composite computational rate is roughly the cell count times each cell's throughput. For balanced computations, where the compiler can fully occupy both adders and multipliers, we can thus expect 100 MFLOPS total. However, because each Warp cell has multiple parallel functional units, an underutilized resource will degrade bandwidth, and so a cell's actual performance will depend upon the program's operation mix. For instance, in a computation containing only additions and no multiplications, the maximum achievable performance falls to 50 MFLOPS. Our studies of scheduling efficiency have shown that the compiler exploits the parallel and pipelined functional units quite effectively. In a sample of 72

programs, Warp achieved a mean computational speed of 28 MFLOPS with an 18 MFLOPS standard deviation.

3.1 Developing the Architecture

The Warp machine represents Carnegie Mellon's second-generation effort in systolic systems. We designed its predecessor, the programmable systolic chip (PSC), as a single-chip microprocessor building block for constructing large systolic arrays. While the PSC successfully demonstrated the idea's feasibility, we soon recognized its limitations: It was difficult to program, slow compared to special-purpose arrays, and capable of only integer arithmetic. Fortunately, a major advance in commercial chip technology occurred just as we commenced work on Warp. Weitek's floating-point chips, first of their kind, significantly eased the task of building systolic systems with truly powerful cells.

Running as an attached processor, Warp forms the high-performance heart of an integrated, general-purpose system. Warp offers parallelism both across the processor array and within individual cells. Each array is a VLIW (very long instruction word) machine with multiple pipelined functional units, all independently controllable. Users can access array-level parallelism directly, and through the W2 compiler, can exploit cell-level parallelism. This flexibile control represents a key to Warp's power.

3.1.1 Powerful systolic cells

Previous systolic systems have typically employed numerous small cells. For Warp, we chose to pursue a design that uses a few, powerful cells in a simple linear array. Our work demonstrates the concept's feasibility: Warp efficiently supports several types of parallel computation.

Coarse- and fine-grain parallelism

Warp's powerful cells support coarse-grain parallelism efficiently. With its own program memory, program sequencer, and data memory, each cell can operate independently. The data memories (4K words in the prototype and 32K words in the PC version) are relatively large for systolic array designs. Big data memories allow individual cells to sustain high computing rates without imposing increased demand on available I/O bandwidth [Kung 86].

Fast communication between cells also makes Warp efficient for the fine-grain parallelism typically found in systolic processing. At 20 Mword/s, Warp's inter-cell I/O bandwidth exceeds that of other processors offering similar computational power and allows neighboring cells to exchange large volumes of intermediate data.

Local and global operations

Systolic arrays are known to be effective for *local* operations, where each output depends only on a small corresponding area of the input. Warp's large memory and high I/O bandwidth also enable it to perform *global* operations, where each output may

depend on a large portion of the input [Kung and Webb 85a]. Computations involving such global operations include FFTs, component labeling, Hough transforms, image warping, and computations such as matrix multiplication or singular value decomposition. Warp's ability to perform global operations significantly extends its computational domain.

3.1.2 Systolic communication support

In systolic computing, unlike other forms of interprocessor cooperation, data passes directly from one cell's data path to its neighbor's, without going through memory. Such communication, transferring individual words, is inherently fine-grained and must be fast and inexpensive. When we began the Warp project, architectural support for this kind of communication was not well understood. Our goal was to provide an efficient communication mechanism suitable for a programmable, general-purpose machine.

One of our initial objectives was a machine that could implement existing systolic algorithms. We began by studying previous designs and identifying the dataflow mechanisms they employ. Many such algorithms use programmable delays to synchronize data streams, and we considered adopting this strategy. A high-performance, programmable processor, however, requires more flexible buffering and, even before building our two-cell prototype, we shifted to a queue-based mechanism.

We implemented the prototype's queues with compile-time flow control. For a substantial set of problems in our application domain, this strategy serves adequately. Applications that permit compile-time flow control include both homogeneous and heterogeneous programs, but not those incorporating WHILE or FOR loops with dynamic bounds. Runtime flow control, while more versatile, can be difficult to design, implement, and debug, so we postponed that refinement. Our redesign for the PC Warp provides run-time flow control and supports the full range of dynamic control flow requirements.

In our first, two-cell machine, receiving cells controlled data latching into the queues. The strategy required close cooperation between sender and receiver and the tight coupling resulted in tremendously increased code size. We improved the situation in the ten-cell prototype. There the sender signals the receiver's queue to latch the incoming datum.

We implemented the prototypes' queues with RAM chips, intending to support both FIFO and random access disciplines. However, there was only a single pair of hardware pointers associated with each queue, and the pointers could not be read under program control. Because the pointers had to be changed when the queue was accessed randomly, it was impossible to use the buffer both for communication and as a local storage element. To improve the array's efficiency, we employed FIFO chips for the PC Warp's queues. The change permits larger queues and relaxes execution coupling between communicating cells by allowing them to send and receive data in larger bursts and at different times.

3.1.3 Inter-cell control coupling

Localized control

Warp's long (272-bit) instructions make it awkward to broadcast instructions to all cells or to propagate them between cells. Moreover, even if cells execute the same program, their computations must often be skewed to delay each cell with respect to its predecessor. To resolve these problems, we chose a MIMD strategy, where each processor has its own control path. Independent control supports both homogeneous computing, the prototypes' computational model, and heterogeneous computing [Annaratone et al. 87a].

The local sequencer also supports conditional branching efficiently. SIMD machines achieve branching by masking, and execution time is the sum of times for branch's THEN and ELSE clauses. With Warp's local program control, an individual cell's data can determine which branch to follow and a conditional statement's execution time reduces to that for the clause selected.

Address generation

For the prototypes, we lacked the VLSI address-generating units (AGUs) that later became available. Thus we chose to generate all common code, including addresses, on the IU and to produce data-dependent code on-cell using the floating-point arithmetic units. This strategy allowed us to handle homogeneous programs—with some restrictions—by paying an execution-time price.

Each PC Warp cell, however, contains an AGU that enables it to support more diverse applications [Annaratone et al. 87b]. With its own AGU, each cell gains both independence and efficiency. Hardware flow control of queues and independent functional units allows individual cells to execute different programs with arbitrary, data-dependent control flow.

3.1.4 Gaining programmability without sacrificing efficiency

One goal in designing the Warp processor was to make the achievable bandwidth as near the Weitek 10 MFLOPS peak as possible. Our strategy was to support direct user/compiler access to datapath parallelism and to make this parallelism easy to exploit [Annaratone et al. 87a]. Warp's wide instruction format provides the key link between the architectural level and datapath parallelism. A dedicated instruction field controls each datapath component and all functional units can be programmed to execute in parallel. Such orthogonal structure in the microinstruction word facilitates scheduling, since schedules for different components do not interfere.

In designing the data path, we ensured that scheduling a resource depends only on the functional unit's availability, and not on other resource schedules. Our approach was threefold. We first provided sufficient internal data bandwidth by connecting all functional units through a crossbar. This approach simplifies scheduling since, unlike a bus-based system, the process need not await an available shared channel. Secondly,

we incorporated internal storage to support the two floating-point processors. These high-speed units can consume up to four data items and generate two results per cycle. Our design uses a five-port, 32-word register file to buffer operands and intermediate results for each processor. Finally, we provided three main datapath sources and drains —two queues and a local memory— and a large backup memory for the register files. When functional units can operate directly on data arriving at the queues, the main drains/sources together offer a data flow rate that matches datapath processing. The register backup increases memory bandwidth and improves throughput for those programs operating mainly on local data.

3.1.5 An integrated, general-purpose host

The Warp array can consume up to five million words each second and generate an equivalent output volume. Designing a host system whose capabilities match Warp's I/O bandwidth posed a significant challenge. We also wanted an open system that we could easily extend as better technology became commercially available.

Two design features contribute to high-speed performance in our host/Warp interface:

- Two clusters within the external host system, as Figure 3-3 illustrates, handle Warp's input and output [Annaratone et al. 87a]. One supplies data to Warp and the other receives results. Each cluster consists of a Motorola 68020 microprocessor and a large local memory. In the PC Warp machine, each cluster also has direct memory access (DMA) capability. For sequential data transfer, DMA permits a transfer in less than 500 ns/word. With block transfer mode, transfer time reduces to about 350 ns/word. Non-sequential transfer speed depends on the complexity of the address computation. For simple address patterns, one 32-bit word is transferred in about 900 ns.
- Data packing and unpacking reduce the host/IU bandwidth requirement by a factor of two to four. In signal, image, and low-level vision applications, input and output data are usually 16- or 8-bit integers. These data can be packed into 32-bit words before transferral to the IU, which then unpacks the data into two or four 32-bit floating-point numbers and sends them on to the Warp array. The reverse operation takes place with the array's floating-point outputs.

We have achieved an open system design by using industry standard VME/VSB protocols. This strategy enables us to employ off-the-shelf components for all external host boards except the crossbar switch. Using standard boards allows us to take advantage of commercial processors, I/O boards, memory, and software. Moreover, standard boards provide a growth path for future system improvements with a minimal investment of time and resources. During the transition from prototype to production machine, for example, we introduced faster processor boards (16 vs. 12 MHz) and larger memories and incorporated both into the host with minimal effort.

Our standard-parts approach to building the host produced two other benefits. It al-

lowed us to concentrate on the array's architecture and sped our implementation of the prototype. Having a prototype early aided development by giving system designers realistic feedback about constraints in the hardware implementation and provided a base for software and application developers to test out their ideas.

3.2 Software system

Although we originally intended to provide only minimal software support for Warp, it quickly became obvious that we needed a high-level language and compiler to make the machine truly usable. In addition to increasing the Warp machine's utility, our work on Warp's W2 language and compiler provided a critical tool for evaluating alternative design strategies and significantly influenced Warp's architectural evolution. Designing and implementing a compiler requires a thorough study of the target machine's functionality. The systematic analysis we undertook in developing W2 allowed us to uncover problems that might have otherwise gone undetected.

3.2.1 Language design

To achieve both generality and efficiency, the user must retain control of how a computation maps across the array. At the cellular level, however, automatic tools can do better. Parallelism available within Warp cells makes hand coding impractical.

We set out to develop a general systolic language (W2) that would allow the user to specify each cell's actions individually while still permitting access to array-level parallelism. Since the user may sometimes need to restructure a sequential algorithm to exploit systolic cells, we also wanted to provide appropriate high-level constructs [Lam 87]. Previous systolic array notations were unsuitable because they typically assumed a simple, repetitive problem domain and dedicated, custom hardware. Our design goals for W2 were:

- Generality sufficient to enable a user to express all programs that the flexible, programmable hardware can support, such as those employing general and data-dependent control flow
- A language design allowing us to build a complier that can generate efficient execution code.

For communication between cells, W2 employs an asynchronous communication model. We chose this strategy because it offers programmability and allows compiler optimization. With appropriate techniques, we can compile unidirectional systolic array programs that use asynchronous communication into highly efficient code. The high-level semantics of asynchronous communication permit "code motion" whereby the compiler can redistribute instructions among basic code blocks and more effectively utilize Warp's intra-cell parallelism [Lam 88a]. W2's asynchronous communication model is even applicable in simple implementations, such as the Warp prototype machines, that have no dynamic flow control hardware. This flexibility derives from W2's efficient compile-time control flow algorithm [Gross and Lam 86].

3.2.2 An optimizing compiler

Code optimization

To exploit the parallelism a VLIW machine offers, "global scheduling" techniques are essential. These techniques, which overlap operations from different basic code blocks, are vital for heavily pipelined and horizontal processors because the basic block structure alone offers very little parallelism. Global scheduling, in turn, relies on accurate global data dependency information. In developing Warp's programming language, we have addressed both data dependency issues and methods for scheduling VLIW machines.

We have implemented a sophisticated global flow analyzer that generates flow information accurate up to the level of individual array elements. It analyzes data dependencies between array accesses throughout a program, within basic blocks and different iterations of a single loop, and across different loops. Labeled arcs in the flow graph capture the derived information, which is then readily available for various code optimizations.

For global scheduling, we concentrated on two techniques: *software pipelining* and *hierarchical reduction* [Lam 88b, Lam 87]. Software pipelining exploits the repetitive nature of innermost loops to generate highly efficient code for processors with parallel, pipelined functional units. We showed that software pipelining is a practical, efficient, and general technique for scheduling the parallelism in a VLIW machine. We have extended previous software pipelining work in two ways. First, we demonstrated that, using scheduling heuristics, we can obtain near-optimal results for all loops. We have improved and extended previous heuristics and introduced a new optimization technique, "modulo variable expansion." Our approach has part of the functionality of the specialized hardware proposed for the polycyclic machine, and thus allows us to achieve similar performance.

Our hierarchical reduction scheme allows us to reduce an entire control construct to an object resembling an operation within a basic block. Previously, software pipelining has been applied only to loops whose bodies are straight-line code segments. Heirarchical reduction allows us to apply software pipelining to arbitrarily complex loops. The significance is threefold: All innermost loops, including those containing conditional statements, can be pipelined. Secondly, if the number of iterations in the innermost loop is small, we can pipeline the second level loop as well. Lastly, hierarchical reduction diminishes the start-up cost penalty for short vectors.

Multiple code generators

Since Warp cell computations are tightly coupled, the compiler must extract data address computation and host communication from the user's program and implement them on the IU and the host. Our design achieves this parallelism by decomposing a program's flow graph into three subgraphs for the cell, IU, and cluster code generators. From the cell code, the compiler extracts timing and sequencing information for the input to and output from the array (including addresses on the address queue). The IU and the host code generators then use this information [Gross and Lam 86].

Retargetability

We have structured the W2 compiler so that we can easily retarget it to handle architectural revisions [Gross and Lam 86]. The compiler was first built for the wirewrapped Warp prototypes, but has since been retargeted for both the PC Warp and iWarp. Many compiler parts can be reused without modification on different architectures. The flow graph representation, for example, is machine-independent, as are the modules that operate on the flow graph: the parser and the local and global dataflow analyzers. The Warp machine's simple, orthogonal instruction set also makes backend modules reusable: scheduler, register allocator, and code emission units in the cell code generator.

3.2.3 Programming environment

The primary objective of the Warp Programming Environment (WPE) is to simplify the use of the Warp machine. Our design provides a uniform environment to edit, compile, debug, and execute W2 programs, supports efficient multiple user access, allows users to access multiple Warp machines, and provides network transparency [Bruegge et al. 87].

The WPE achieves efficiency and convenience by supporting two modes of accessing Warp. Users may opt for convenience by using the Warp shell, a command interpreter that interfaces the user to the components to the WPE. Or he may choose efficiency by programming the machine in "standalone" mode, calling run-time system procedures directly.

The runtime system supports multi-user access through two kinds of server processes. The *Warp server* manages machine access through functions that lock and unlock Warp for different users. Multiple *user servers* provide the primary location for variables an individual user creates in his own Warp shell. Memory is copied back and forth between user server and Warp machine each time a user accesses the machine. This feature aids the efficient use of the machine by permitting the environment to maintain user-specific state information across several locks/unlocks of the Warp machine and by making it possible to initialize shell variables without monopolizing the Warp array.

3.2.4 Debugger

We have also developed a symbolic debugger for W2 programs that allows the user to set source line breakpoints and inspect symbolic variables [Bruegge et al. 87]. The prototype provided only a post-mortem debugging mode because the machine's hardware pipelines made restarting impossible. The PC Warp has better support for interactive debugging and can resume execution after the user inspects the machine's internal state.

3.3 Applications

Applications played a multifaceted role throughout the system development process. A systolic array with genuinely powerful cells represents a new machine organization and, as we developed applications, we also evolved a set of general machine models. Our initial study of potential applications provided critical guidance on system requirements, such as memory size and I/O bandwidth. We chose the vision domain because CMU researchers have both interest and expertise there and because its tasks require intensive computation. By focusing on one area, we enhanced the chances that Warp could actually provide a useful resource for real research problems. The continuing, independent pursuit of Warp applications also provided feedback for system development. With real users we could more rapidly locate problem areas and bottlenecks and make appropriate improvements. Finally, application software provided valuable benchmark performance data. The following sections describe work in several application areas and the general algorithm mapping methods we developed.

3.3.1 Application areas

Warp machines have proven useful in several task areas [Gross et al. 85, Kung and Webb 86, Annaratone et al 86] [Annaratone et al. 87c, Annaratone et al. 87d] [Clune et al. 87]:

- *General image processing*—We have implemented an extensive subroutine library for image processing. With this resource, researchers are now using the Warp machine in vision work as well as more applied domains. Other investigators are seeking ways to exploit Warp for processing medical images, particularly nuclear magnetic resonance (NMR) data.
- *Image processing for robot navigation*—Algorithms and systems implemented include road following, obstacle avoidance using stereo vision, and obstacle avoidance using the Environmental Research Institute of Michigan (ERIM) laser range scanner.
- *Signal processing*—We have developed several different algorithms in this area, including singular value decomposition (SVD) for adaptive beam forming.
- *Scientific computing*—Algorithms include successive over-relaxation (SOR) for solution of systems of partial differential equations.

In addition, CMU has been providing help to several DARPA contractors in their applications of Warp, including Martin Marietta Corporation in Autonomous Land Vehicles and Hughes Aircraft Corporation in image analysis.

3.3.2 Program partitioning methods

We have identified three general program partitioning methods: *input partitioning*, *output partitioning*, and *pipelining* [Annaratone et al. 87d, Kung and Webb 86].

Input partitioning

Input partitioning is useful, for example, in image processing where the result at each point of the output image depends only on a small neighborhood of the corresponding point of the input image. In this model, the input data are partitioned among the Warp cells. Each cell computes on its portion of the input data to produce a corresponding portion of the output data.

This model provides a simple and powerful method for exploiting parallelism—most parallel machines support it in one form or another. Many Warp algorithms use it, including most of the low-level vision programs, the discrete cosine transform (DCT), singular value decomposition [Annaratone et al 86], connected component labeling [Kung and Webb 86], border following, and the convex hull procedure.

Output partitioning

Output partitioning is useful when the input to output mapping is not regular, or when any input can influence any output. Histogram and image warping are examples of such computations. This model usually requires extensive memory because either the required input data set must be stored and then processed later, or the output must be stored in memory while the input is processed. For output partitioning, each Warp cell processes the entire input data set or a large part of it, but produces only part of the output. Each cell has 32K words of local memory to support efficient use of this model.

Pipelining

For some algorithms, pipelining represents the only possible means to parallel computation. In this model, typical of systolic computation, the algorithm is partitioned among the cells in the array and each cell performs one stage of the processing. Warp's high inter-cell communication bandwidth and its effectiveness in handling fine-grain parallelism make pipelining possible.

A simple example is the solution of elliptic partial differential equations using successive over-relaxation. Each cell is responsible for one relaxation. In raster order, each cell receives inputs from the preceding cell, performs its relaxation step, and outputs the results to the next cell. While a cell is performing the k^{th} relaxation step on row i, the preceding and subsequent cells perform the $k-1^{st}$ and $k+1^{st}$ relaxation steps on rows $i+2$ and $i-2$, respectively. Thus, in one pass of the u values through the 10-cell Warp array, the relaxation steps are performed ten times. This process is repeated, under control of the external host, until convergence is achieved.

3.4 Bibliography

[Annaratone et al 86]
Annaratone, M., E. Arnould, H.T. Kung, and O. Menzilcioglu.
Using Warp as a supercomputer in signal processing.
In *Proceedings of ICASSP*, Pages 2895-2898. IEEE, April, 1986.

Warp is a programmable systolic array machine designed by CMU and built together with its industrial partners, GE and Honeywell. The first large scale version of the machine with an array of 10 linearly connected cells will become operational in January 1986. Each cell in the array is capable of performing 10 million 32-bit floating-point operations per second (10 MFLOPS). The 10-cell array can achieve a performance of 50 to 100 MFLOPS for a large variety of signal processing operations such as digital filtering, image compression, and spectral decomposition. The machine, augmented by a boundary processor, is particularly effective for computationally expensive matrix algorithms such as solution of linear systems, QR-decomposition and singular value decomposition, that are crucial to many real-time signal processing tasks. This paper outlines the Warp implementation of the 2-dimensional Discrete Cosine Transform and singular value decomposition.

[Annaratone et al. 85]
Annaratone, M., E. Arnould, P.K. Hsiung, and H.T. Kung.
Extending the CMU warp machine with a boundary processor.
In *Proceedings of the International Society for Optical Engineers*, SPIE, January, 1985.

A high-performance systolic array computer called *Warp* has been designed by CMU and is currently under construction. The full scale machine has a systolic array of 10 or more linearly connected cells, each of which is a programmable processor capable of performing 10 million floating-point operations per second (10 MFLOPS). By the end of 1985 the first full scale machine will be operational. Low-level vision processing for robots and autonomous vehicles are among the first applications of the machine.

This paper describes a new *boundary processor* to be attached to an end of the linear systolic array in Warp. Extending Warp with this boundary processor will substantially enhance the performance and applicability of the machine. The extended machine will be efficient for new application areas such as solution of linear systems of equations and adaptive signal processing.

[Annaratone et al. 86]
Annaratone, M., E. Arnould, T. Gross, H.T. Kung, M.S. Lam,
O. Menzilcioglu, K. Sarocky, and J.A. Webb.
Warp architecture and implementation.
In *13th Annual International Symposium on Computer Architecture*,
IEEE, June, 1986.

A high-performance systolic array computer called *Warp* has been designed and constructed. The machine has a systolic array of 10 or more linearly connected cells, each of which is a programmable processor capable of performing 10 million floating-point operations per second (10 MFLOPS). A 10-cell machine therefore has a peak performance of 100 MFLOPS. Warp is integrated into a UNIX host system. Program development is supported by a compiler.

The first 10-cell machine became operational in 1986. Low-level vision processing for robot vehicles is one of the first applications of the machine.

This paper describes the architecture and implementation of the Warp machine, and justifies and evaluates some of the architectural features with the system, software and applications considerations.

[Annaratone et al. 87a]
Annaratone, M., E. Arnould, T. Gross, H.T. Kung, M. Lam,
O. Menzilcioglu, and J.A. Webb.
The Warp computer: architecture, implementation, and performance.
IEEE Transactions on Computers C-36(12), December, 1987.
Also appeared as tech report CMU-CS-87-166.

The Warp machine is a systolic array computer of linearly connected cells, each of which is a programmable processor capable of performing 10 million floating-point operations per second (10 MFLOPS). A typical Warp array comprises 10 cells, thus having a peak computation rate of 100 MFLOPS. Warp is integrated as an attached processor into a UNIX host system. Programs for Warp are written in a high-level language supported by an optimizing compiler.

The first 10-cell machine became operational in February 1986. Five machines have been built as of June 1987, and more are under construction. Warp has been demonstrated to be effective in the application domain of low-level vision processing for robot navigation, as well as other fields such as signal processing, scientific computation, and texture image analysis. The average performance of Warp for a large sample of programs in these application areas is 28 MFLOPS.

This paper describes the architecture, implementation and performance of the Warp machine. Each major architectural decision is discussed and evaluated with system, software,

and application considerations. This paper also describes the programming model and support developed to allow us to use the machine effectively. The paper concludes with performance data measured for a large number of applications.

[Annaratone et al. 87b]
Annaratone, M., E. Arnould, R. Cohn, T. Gross, H.T. Kung, M. Lam, O. Menzilcioglu, K. Sarocky, J. Senko, and J.Webb.
Warp architecture: from prototype to production.
In *Proceedings of the 1987 National Computer Conference*, June, 1987.

The Warp machine is a high-performance systolic array computer with a linear array of 10 or more cells, each of which is a programmable processor capable of performing 10 million floating-point operations per second (10 MFLOPS). Warp is integrated into a UNIX host system, and program development is supported by a compiler.

Two copies of a 10-cell prototype of the Warp machine became operational in 1986 and are in use at Carnegie Mellon for a wide range of applications, including low-level vision processing for robot vehicle navigation and signal processing. The success of the prototypes led to the development of a production version of the Warp machine that is implemented with printed circuit boards. At least eight copies of this machine are being built by General Electric in 1987. The first copy was delivered to Carnegie Mellon in April 1987. This paper describes the architecture of the production Warp machine and explains the changes that turned the prototype system into a mature high-performance computing engine.

[Annaratone et al. 87c]
Annaratone, M., F. Bitz, E. Clune, H.T. Kung, P. Maulik, H. Ribas, P. Tseng, and J. Webb.
Applications and algorithm partitioning on Warp.
In *Proceedings of Compcon Spring '87*, IEEE, February, 1987.

The prototype Warp machines at Carnegie Mellon have been used in a diverse range of applications, including robot vehicle control, scientific computing, and medical image processing, and as a tool for vision research. A small number of algorithm partitioning methods have allowed efficient use of the Warp machine in all of these areas. Large applications that use Warp as part of a system are efficiently supported by Warp's flexible host.

[Annaratone et al. 87d]
Annaratone, M., F. Bitz, J. Deutch, H.T. Kung, L. Hamey, P. Maulik, P. Tseng, and J. Webb.
Applications experience on Warp.
In *Proceedings of the 1987 National Computer Conference*, Pages 149-158. AFIPS, June, 1987.
Also appeared in *Proceedings of Compcon Spring 1987.*

The prototype Warp machine at Carnegie Mellon is being used to develop new applications in magnetic resonance image processing, as a research tool in image texture analysis, and for scientific computing. In these areas, orders of magnitude speedup over conventional computers are being observed. These new applications build on our use of Warp for low level vision, which is the area for which the machine was originally designed.

Experience with the prototype Warp machine has led to rules that programmers should follow to achieve best performance in their application. These rules concern all levels of the Warp system, from input and output ordering to programming each individual Warp cell to memory use in Warp's host. The new printed circuit board version of Warp incorporates several architectural improvements, which lead to better support of a wider class of applications.

An ambitious design for implementation of Warp in custom VLSI is underway, which promises an increase of at least ten in cost-performance over the current version of Warp, together with the opportunity to build much more powerful systolic arrays delivering GigaFLOPS performance.

[Annaratone et al. 87e]
Annaratone, M., E. Arnould, R. Cohn, T. Gross, H.T. Kung, M. Lam, O. Menzilcioglu, K. Sarocky, J. Senko, and J. Webb.
Architecture of Warp.
In *Proceedings of Compcon Spring '87*, IEEE Computer Society, February, 1987.

Warp is a high-performance systolic array computer. A linear array of cells is connected to a host computer operating under the UNIX operating system. Each cell of the array is a programmable processor capable of performing 10 million floating-point operations per second. To date, two 10-cell prototype systems have been built and are in use; eight more systems are under construction.

This paper describes the architecture and implementation of the Warp cells, the array configuration, and the organization of the host system.

[Arnould et al. 85] Arnould, E., H.T. Kung, O. Menzilcioglu, and K. Sarocky.
A systolic array computer.
In *Proceedings of the International Conference On Acoustics, Speech, and Signal Processing*, IEEE, March, 1985.

A high-performance systolic array computer has been designed at CMU and is currently under construction. The first copy of the machine, to be built by CMU together with its industrial partners before the end of 1985, will incorporate a programmable systolic array of ten linearly connected cells. Each cell in the systolic array is capable of performing 10 million floating-point operations per second (10 MFLOPS), giving the total machine a peak performance of 100 MFLOPS, or higher if additional cells are used. This particular systolic array computer is called *Warp*, suggesting that it can perform computations at a very high speed. The 10-cell systolic array, with one cell implemented on one board, can process 1024-point complex FFTs at a rate of one FFT every 600 μs. Under program control, the same array can perform many other primitive computations in signal, image, and vision processing, including two-dimensional convolution, dynamic programming, and real or complex matrix multiplication, at a rate of 100 million floating-point operations per second. Users may view the systolic array as an array of conventional "array processors," which can efficiently implement not only systolic algorithms where communication between intensive cells is intensive, but also non-systolic algorithms where each cell operates on its own cell data independently from the rest. This paper describes the hardware organization of the Warp machine.

[Bruegge et al. 87] Bruegge, B., C. Chang, R. Cohn, T. Gross, M. Lam, P. Lieu, A. Noaman, and D. Yam.
The Warp programming environment.
In *Proceedings of the 1987 National Computer Conference*, AFIPS, June, 1987.

This paper describes the environment for developing and executing Warp programs. The center of the program development environment is a customized shell that ties together a compiler for the Warp array, the Warp run-rime system, and a debugger. The compiler translates high-level language programs to microcode for the Warp machine. It achieves a high utilization of the computation power of the processor. The run-time system supports remote execution of Warp programs across a network and makes the Warp machine available as a sharable resource. The debugger permits symbolic debugging of Warp programs. The Warp programming environment makes the Warp machine an easily programmable and accessible attached processor in a UNIX environment.

[Clune et al. 87] Clune, E., J.D. Crisman, G.J. Klinker, and J.A. Webb.
Implementation and performance of a complex vision system on a systolic array machine.
Technical Report CMU-RI-TR-87-16, The Robotics Institute, Carnegie Mellon University,
June, 1987.

Complex vision systems are usually quite slow, requiring tens of seconds or minutes of computer time for each image. As the complexity and experimental nature of the system increases, the speed is especially low, since all components of the system must be optimized if the system is to show good performance. The FIDO system, a stereo vision system for controlling a robot vehicle, has existed for a number of years and has been implemented on a number of different computers. These computers have ranged from a DEC KL10 to the current implementation on the Warp machine, a 100 Million Floating Point Operations Per Second (MFLOPS) systolic array machine. FIDO has shown enormous range in speed; its ancestor took 15 minutes per step, while the Warp implementation takes less than 5 seconds per step. Moreover, while early versions of FIDO moved in slow, start-and-stop steps, FIDO now runs continuously at 100 mm/second. We review the history of the FIDO system, discuss its implementation on different computers, and concentrate on its current Warp implementation.

[Deutch et. al. 87] Deutch, J., P.C. Maulik, R. Mosur, H. Printz, H. Ribas, J. Senko, P.S. Tseng, J.A. Webb, and I.C. Wu.
Performance of Warp on the DARPA architecture benchmarks.
Technical Report 87-148, Computer Science Department, Carnegie Mellon University,
September, 1987.

Warp was a participant in the DARPA Architecture Workshop Benchmark Study, which compared performance of a variety of architectures for image processing on image processing tasks from low-level and mild-level vision. We present algorithms and performance figures resulting from this study. These algorithms and performance numbers can be used as a guide to Warp programming at the time of this study. Based on these performance figures, we can evaluate the architectural decisions made in the Warp design.

[Fisher et al. 84] Fisher, A.L., H.T. Kung, and K. Sarocky.
Experience with the CMU programmable systolic chip.
In *Proceedings of SPIE Symposium, Vol. 495, Real-Time Signal Processing VII*, Society of Photo-Optical Instrumentation Engineers, August, 1984.

The CMU *programmable systolic chip* (PSC) is an experimental, microprogrammable chip designed for the efficient implementation of a variety of systolic arrays. The PSC has been designed, fabricated, and tested. The chip has about 25,000 transistors, uses 74 pins, and was fabricated through MOSIS, the DARPA silicon broker, using a 4 micron nMOS process. A modest demonstration system involving nine PSCs is currently running. Larger demonstrations are ready to be brought up when additional working chips are acquired.

The development of the PSC, from initial concept to a silicon layout, took slightly less than a year, but testing, fabrication, and system demonstration took an additional year. This paper reviews the PSC, describes the PSC demonstration system, and discusses some of the lessons learned from the PSC project.

[Gross and Lam 86]
Gross, T. and M.S. Lam.
Compilation for a high-performance systolic array.
In *Proceedings of the SIGPLAN 86 Symposium on Compiler Construction*, ACM SigPlan, June, 1986.

We report on a compiler for Warp, a high-performance systolic array developed at Carnegie Mellon. This compiler enhances the usefulness of Warp significantly and allows application programmers to code substantial algorithms.

The compiler combines a novel programming model, which is based on a model of skewed computation for the array, with powerful optimization techniques. Programming in W2(the language accepted by the compiler) is orders of magnitude easier than coding in microcode, the only alternative available previously.

[Gross et al. 85] Gross, T., H.T. Kung, M. Lam, and J. Webb.
Warp as a machine for low-level vision .
In *Proceedings of the International Conference on Robotics and Automation*, IEEE, March, 1985.

Warp is a programmable systolic array processor. One of its objectives is to support computer vision research. This paper shows how the Warp architecture can be used to fulfill the computational needs of low-level vision.

We study the characteristics of low-level vision algorithms and show they lead to requirements for computer architecture. The requirements are met by Warp. We then describe how the Warp system can be used. Warp programs can be classified in two ways: chained versus severed, and heterogeneous versus homogeneous. Chained and severed characterize the degree of interprocessor dependency, while heterogeneous and homogeneous charac-

terize the degree of similarity between programs on individual processors. Taken in combination, these classes give four user models. Sophisticated programming tools are needed to support these user models.

[Hsu et al. 85] Hsu, F.H., H.T. Kung, T. Nishizawa, and A. Sussman.
Architecture of the link and interconnection chip.
In *Proceedings of the 1985 Chapel Hill Conference on VLSI*, Computer Science Press, 1985.

The link and interconnection chip (LINC) is a custom chip whose function it is to serve an efficient link between system functional modules, such as arithmetic units, register files and I/O ports. This paper describes the architecture of LINC, and justifies it with several application examples.

LINC has 4-bit datapaths consisting of an 8x8 crossbar interconnection, a FIFO or programmable delay for each of its inputs, and a pipeline register file for each of its outputs. Using pre-stored control patterns LINC can configure an interconnection and delays on-the-fly. Therefore the usual functions of busses and register files can be realized with this single chip.

LINC can be used in a bit-sliced fashion to form interconnections with datapaths wider than 4 bits. Moreover, by tristating the proper data output pins, multiple copies of LINC can be used for crossbar interconnections larger than 8x8.

Operating at the target cycle time of 100ns, LINC makes it possible to implement a variety of high-performance processing elements with much reduced package counts.

[Kanade and Webb 87]
Kanade, T., and J.A. Webb.
End of year report for parallel vision algorithm design and implementation.
Technical Report CMU-RI-TR-87-15, The Robotics Institute, Carnegie Mellon University,
June, 1987.

The parallel vision algorithm design and implementation project was established to facilitate vision programming on parallel architectures, particularly low-level vision and robot vehicle control algorithms on the Carnegie Mellon Warp machine. To this end, we have (1) demonstrated the use of the Warp machine in several different algorithms; (2) developed a specialized programming language, called Apply, for low-level vision programming on parallel architectures in general, and Warp in particular; (3) used Warp as a research tool in vision, as opposed to using it only for research in parallel vision; (4) developed a significant library of low-level vision programs for use on Warp.

[Kung 86] Kung, H.T.
Memory requirements for balanced computer architectures.
In *13th Annual International Symposium on Computer Architecture*, IEEE, June, 1986.
Also appeared in *Journal of Complexity*, Vol. 1, No. 1., 1985.

A processing element (PE) can be characterized by its computational bandwidth, I/O bandwidth, and the size of its local bandwidth. In carrying out a computation, a PE is said to be *balanced* if the computational time equals the I/O time. Consider a balanced PE for some computation. Suppose that the computational bandwidth of the PE is increased by a factor of α relative to its I/O bandwidth. Then when carrying out the same computation the PE will be imbalanced; i.e. it will have to wait for I/O. A standard method for avoiding this I/O bottleneck is to reduce the overall I/O requirements of the PE by increasing the size of its local memory. This paper addresses the question of by how much the PE's local memory must be enlarged in order to restore balance.

The following results are shown: For matrix computations such as matrix multiplication and Gaussian elimination, the size of the local memory must be increased by a factor of α^2. For computations such as relaxation on a d-dimensional grid, the local memory must be increased by a factor of α^d. For some other computations such as fast Fourier transform and sorting, the increase is exponential; i.e., the size of the new memory must be the same as the old memory to the α-th power. All these results indicate that the size of a PE's local memory should be increased much more rapidly that the PE's computational bandwidth. This phenomenon seems to be common for many computations where an output may depend on a large subset of the inputs.

Implications of these results for some parallel computer architectures are discussed. One particular result is that to balance an array of p linearly connected PEs for performing matrix computations such as matrix multiplication and matrix triangularization, the size of each PE's local memory must grow linearly with p. Thus, the larger the array is, the larger each PE's local memory must be.

[Kung and Lam 84]
Kung, H.T. and M.S. Lam.
Wafer-scale integration and two-level pipelined implementations of systolic arrays .
*Journal of Parallel and Distributed Computing*1:32-63, 1984.
A preliminary version appears in *Proc. Conference on Advanced Research in VLSI*, MIT, January 1984.

[Kung and Webb 85a]
Kung, H.T. and J.A. Webb.
Global operations on the CMU Warp machine.
In *Proc. 1985 AIAA Computers in Aerospace V Conference*, Pages 209-218. American Institute of Aeronautics and Astronautics, October, 1985.

CMU is developing a high-performance machine, called Warp, for image and signal processing. The machine has a programmable systolic array of linearly connected cells, each capable of performing 10 million floating-point operations per second. It is not surprising that the array can efficiently perform local operations, in which each output depends on a small corresponding area of the input, since the connections between the cells are neighbor connections. However, Warp is also suited to global image operations, in which each output can depend on any or a large portion of the inputs. In this paper we show this, and discuss the reasons why.

As example global operations we take the fast Fourier transform (FFT), component labeling, Hough transform, and image warping. The FFT is am important computation in signal processing. Component labeling is a basic operation in image processing, often the last operation done before symbolic processing takes over. Hough transform, a technique used to match curve templates in images, is finding wide use in image processing these days, because of its robust performance in the presence of noise. Image warping is used to correct for lens distortions or to normalize images to make later processing easier. It is a time-consuming step not readily implementable on most parallel machines.

We describe how Warp can efficiently implement these global operations. In particular, an efficient parallel algorithm for component labeling is proposed.

[Kung and Webb 85b]
Kung, H.T. and J.A. Webb.
Global operations on a systolic array machine.
In *Proceedings of the International Conference on Computer Design: VLSI in Computer*, IEEE, October, 1985.

CMU is developing a high-performance machine, called Warp, for image and signal processing. The machine has a programmable systolic array of linearly connected cells, each capable of performing 10 million floating-point operations per second. It is not surprising that the array can efficiently perform local operations, in which each output depends on a small corresponding area of the input, since the connections between the cells are neighbor connections. However, Warp is also suited to global image operations, in which each output can depend on any or a large

portions of the inputs. In this paper we show this, and discuss the reasons why.

As example global operations we take the fast Fourier transform (FFT), component labeling, and Hough transform. The FFT is an important computation on signal processing. Component labeling is a basic operation in image processing, often the last operation done before symbolic processing takes over. Hough transform, a technique used to match curve templates in images, is finding wide use in image processing these days, because of its robust performance in the presence of noise.

We describe how Warp can efficiently implement these global operations. In particular, a component labeling algorithm suitable for Warp is proposed. This algorithm appears to be simpler and faster than previously known algorithms, even for a conventional sequential machine.

[Kung and Webb 86]
Kung, H.T. and J.A. Webb.
Mapping image processing operations onto a linear systolic machine.
Distributed Computing 1(4):246-257, 1986.

A high-performance systolic machine, called Warp, is operational at Carnegie Mellon. The machine has a programmable systolic array of linearly connected cells, each capable of performing 10 million floating point operations per second. Many image processing operations have been programmed on the machine. This programming experience has yielded new insights in the mapping of image processing operations onto a parallel computer. This paper identifies three major mapping methods that are particularly suited to a Warp-like parallel machine using a linear array of processing elements. These mapping methods correspond to partitioning of input dataset, partitioning of output dataset, and partitioning of computation along the time domain (pipelining). Parallel implementations of several important image processing operations are presented to illustrate the mapping methods. These operations include the Fast Fourier Transform (FFT), connected component labeling, Hough transform, image warping, and relaxation.

[Lam 87] Lam, M.
An optimizing systolic array compiler.
PhD thesis, Computer Science Department, Carnegie Mellon University, May, 1987.

The Warp machine is a linear array of ten programmable processors and is capable of executing 100 million floating-point operations per second (100 MFLOPS). The individual processors, or cells, derive their performance from

a wide instruction set and a high degree of internal pipelining and parallelism.

My thesis is that systolic arrays of high-performance cells can be programmed effectively using a high-level language. The solution has two components: a machine abstraction and compiler optimizations for systolic arrays, and code scheduling techniques for horizontally microcoded or VLIW processors.

In the proposed machine abstraction, individual cells are programmed in a high-level programming language; inter-cell communication is explicitly specified by asynchronous primitives: receive and send operations. This machine abstraction offers both efficiency and generality. Unidirectional systolic array programs can be compiled into highly efficient code by compiler optimizations that exploit the high-level semantics of asynchronous communication. This abstraction is applicable even for simple implementations with no dynamic flow control hardware by using an efficient compile-time control flow algorithm.

This thesis shows that software pipelining is a practical and efficient code scheduling technique for highly parallel and pipelined processors. We have extended the previous scheduling algorithm and introduced a new optimization called modulo variable expansion. We show that near-optimal results can be obtained using software heuristics. This thesis also proposes a unified approach to scheduling both within and across basic blocks called hierarchical reduction. This technique makes software pipelining applicable to all innermost loops, including those containing conditional statements. A consistent performance improvement can thus be obtained for all programs.

[Lam 88a] Lam, M.
Compiler optimizations for asynchronous systolic array programs.
In *Proc. Fifteenth Annual ACM Symposium on Principles of Programming Languages*, Jan., 1988.

A programmable systolic array of high-performance cells is an attractive computation engine if it attains the same utilization of dedicated arrays of simple cells. However, typical implementation techniques used in high-performance processors, such as pipelining and parallel functional units, further complicate the already difficult task of systolic algorithm design. This paper shows that high-performance systolic arrays can be used effectively by presenting the machine to the user as an array of conventional processors communicating asynchronously. This abstraction allows the user to focus on the higher level problem of partitioning a computation across cells in the array. Efficient fine-grain parallelism can be achieved by code motion of

communication operations made possible by the asynchronous communication model. This asynchronous communication model is recommended even for programming algorithms on systolic arrays without dynamic flow control between cells.

The ideas presented in the paper have been validated in the compiler for the Warp machine. The compiler has been in use in various application areas including robot navigation, low-level vision, signal processing and scientific programming. Near-optimal code has been generated for many published systolic algorithms.

[Lam 88b] Lam, M.
Software pipelining: an effective scheduling technique for VLIW machines.
In *Conference on Programming Language Design and Implementation.*, ACM Sigplan, June, 1988.

This paper shows that software pipelining is an effective and viable scheduling technique for VLIW processors. In software pipelining, iterations of a loop in the source program are continuously initiated at constant intervals, before the preceding iterations complete. The advantage of software pipelining is that optimal performance can be achieved with compact object code.

This paper extends previous results of software pipelining in two ways: First, this paper shows that by using an improved algorithm, near-optimal performance can be obtained without specialized hardware. Second, we propose a *hierarchical reduction* scheme whereby entire control constructs are reduced to an object similar to an operation in a basic block. With this scheme, all innermost loops, including those containing conditional statements, can be software pipelined. It also diminishes the start-up cost of loops with small number of iterations. Hierarchical reduction complements the software pipelining technique, permitting a consistent performance improvement be obtained.

The techniques proposed have been validated by an implementation of a compiler for Warp, a systolic array consisting of 10 VLIW processors. This compiler has been used for developing a large number of applications in the areas of image, signal and scientific processing.

[Siegell and Gross 87]
Siegell, B. and T. Gross.
Program-specific and architecture-specific simulators.
In *Proceedings of the 8th International Symposium on Computer Hardware Description Languages and their Applications*, April, 1987.

The use of compilation techniques makes it possible to automatically produce efficient functional simulators from a given machine description. A compiler produces an architecture-specific simulator by binding various parameters like the word size, unit of memory access, etc., which are otherwise repetitively evaluated at runtime by a generic interpretative simulator. This idea of early binding can be extended to include the program that is run on the simulator. The result is a program-specific simulator that simulates the target architecture for exactly one program.

We have implemented PAST, a compiler tool to generate program- and architecture-specific simulators from ISPS descriptions. Simulators compiled with PAST are an order of magnitude faster than running the ISPS interpreter. A program-specific simulator offers an additional two-fold improvement over the architecture-specific simulator. These improvements are obtained at the expense of increased simulator preparation times, and the paper discusses the tradeoffs between the different approaches.

[Sun 86] Sun,Y.
Verification of systolic arrays: a FP functional approach.
Technical Report CMU-CS-86-135, Computer Science Department, Carnegie Mellon University,
April, 1986.

There has been much interest in the use of formal techniques for the design and analysis of systolic arrays. One important aspect of the analysis of systolic arrays is the correctness problem.

A few attempts at the verification of systolic arrays have appeared in the literature. The deficiency is that all of these methods lack a straight- forward way of proving correctness. They require either proposing a solution, then applying inductive techniques or showing that the array satisfies three types of properties: safety, liveness, and termination.

In this paper, an FP functional approach is proposed. The goal is to verify that a given systolic design computes the function for which it was intended instead of the generation of a systolic architecture. The method generates a system of recursive functional equations which describes the algorithm executed by the architecture. This representation consists of several equations describing connections between cells, functions representing data streams, and functions describing the relation between the structure of input and output data and the systolic array structure. The minimum solution of the system of recursive functional equations is the function computed by the systolic architecture.

The main advantage of this approach is that it allows us to develop an algebra of functional programs. We have

developed various methods to deal with different kinds of systems of functional recursive equations. By solving the system of recursive functional equations, we can get the least solution directly. This provides a straightforward way for proving correctness.

An example is given. A typical system of recursive functional equations is generated. An algebra method is developed showing how to solve this problem, because most systolic designs can be represented by it.

4. THE PRODUCTION SYSTEM MACHINE PROJECT

4.1 Introduction

Production systems embody a form of program organization used for many applied AI systems, especially knowledge-intensive expert systems comprising large *if-then* rule sets, also called *rule-based* systems. This organizational form is especially suited to tasks that draw on well-defined bodies of expertise. The development of production systems has played a leading role in the recent, dramatic rise of AI expert systems to the point of industrial and commercial application. Their sudden ascendance, in turn, reflects the current belief that knowledge-based systems will be a central feature of tomorrow's computers.

The need for special machines to process production systems arises from the specific computational problem of finding the appropriate rules within a large (ultimately huge) collection in the knowledge base. Moreover, this search-and-recognize process must proceed continually and virtually instantaneously as the working situation changes, so that newly relevant knowledge can be immediately applied to the current task. This computational task must be addressed efficiently if the current style of expert systems continues to mature and if this style is to transform into substantially more capable systems.

Researchers have been exploring many alternative ways for speeding up the execution of production systems. While some of our efforts have focused on high-performance uniprocessor implementations, our efforts under the contract have concentrated on parallel implementations because of a production system's potential for exploiting large amounts of parallelism. We now describe this potential for parallelism in production systems in general and the Rete algorithm in particular.

4.1.1 Sources of parallelism in production systems

A production system divides easily at several levels for parallel processing. For example, a production system interpreter might exploit parallelism within each of the steps it takes to fire a rule. These steps, called the *recognize-act* cycle, are as follows:

- *Match*: Match condition elements of the left-hand sides (or "if" part) of all productions against working memory contents. It is during this step that the knowledge of the intelligent agent (the expert system) is applied to the current problem state. The result is a *conflict set* that consists of instantiations of all satisfied productions.
- *Conflict Resolution*: Choose one of the production instantiations in the conflict set for execution. Halt if no productions are satisfied.
- *Act*: Execute the selected production. These actions may change the contents of working memory. Return to the match phase.

The system may overlap processing of each step to achieve more speedup.

Match, the most time-consuming step, offers additional sources for parallelism. Even with specialized algorithms, match constitutes around 90% of the interpretation time. We therefore focused on speeding up match using parallelism. We began by implementing the highly efficient Rete match algorithm used by non-parallel OPS5 implementations.

The Rete algorithm

Rete compiles a data flow graph from the left-hand sides of productions. Data objects called *tokens* flow between graph nodes and consist of a list of working memory elements plus a tag. The working memory elements correspond to those elements that the system is trying to match or has already matched against condition elements in the left hand side. The tag ("+" or "-") indicates whether that list of elements has been added or removed from working memory. A token's arrival at one input of a *two-input* node *activates* that node: that is, it gives the node new data for processing. The system processes the activation by comparing the new token to each token stored at the opposite input. The processor sends token pairs that have consistent variable bindings to the two-input node's successor.

Rete exploits two production system features that make it an efficient match algorithm: (1) the fact that only a small fraction of working memory changes every cycle and (2) the similarity between a production's condition elements. It stores results of match from previous cycles and uses them in subsequent cycles, and it performs tests on common condition elements only once. The algorithm thus only processes changes made during the most recent production firing and avoids repeating identical tests unnecessarily.

Exploiting parallelism to speed up match

To speed the matching process, we isolated three match-phase execution levels that could benefit from parallelism: production-, action-, and node-level execution. Production-level parallelism is the most obvious source of better match speed. To achieve it, the programmer divides the program into groups of productions [Oflazer 84]. The system can then match each group in parallel. The extreme case for production-level parallelism occurs when the number of groups equals the number of productions in the program, so that the match for each production in the program is performed in parallel. Action-level parallelism involves processing working memory changes in parallel, instead of sequentially. The Rete algorithm itself adds another dimension to the potential speedups from parallelism by allowing a finer grain of parallelism than these other two levels: It permits the system to process different two-input node activations of the same or different productions in parallel.

4.1.2 Research goals and considerations

Our goal during the contract was to exploit a production system's potential for parallel processing, with special emphasis on speeding up the match process. Our strategy was to design, implement, and evaluate a parallel interpreter.

The PSM project concentrated on two production systems, OPS5 and the OPS5-based Soar architecture. We selected OPS5 for this study because a number of substantial OPS5 application programs were available to serve as machine benchmarks, including the largest production systems in existence at the time. Soar, on the other hand, is a new production system architecture developed at Carnegie Mellon University to perform research in problem solving, expert systems, and learning. It is an attempt to provide expert systems with general reasoning power and the ability to learn. Soar programs integrate learning into performance systems more generally and more completely than any other programs at this time. Currently, OPS-Soar is built on top of OPS5, and its syntax is similar to that of OPS5.

4.2 Designing a Parallel Interpreter

Our first step in designing a parallel interpreter was to evaluate possible sources for parallelism. Based on our findings, we proposed general requirements for the interpreter architecture.

4.2.1 Evaluating opportunities for parallelism

Our goal was to design a parallel interpreter that efficiently exploited the parallelism inherent in a production system. Since the match operation is the most expensive part of the cycle, we examined production-, action-, and node-level parallelism more closely than other potential sources.

Previous analysis of parallelism in production systems had used very simple models capable of exploring only production-level parallelism. Those models, however, failed to consider the variation in the cost of processing the production activations. To be able to test a finer grain of parallelism, we needed a simulator that could trace node-level activations and their cost. The simulator would rely on an accurate cost model to determine system execution time, reflecting the effects of algorithms and data structures used to process node activations, code used to push/pop node activations from the task scheduler, multiprocessor structure, etc.

We solved this problem by building an event-driven simulator. The simulator's main input consisted of a detailed trace of node activations in the Rete network corresponding to a production system run. Other input sources were the cost model and a specification of the parallel computational model on which the trace was to be executed. The trace contained information about the dependencies between the node activations, and the simulator understood which node activations could or could not be processed in parallel. The trace also contained other information necessary to determine the cost of a given node activation. The simulator's output consisted of statistics for the overall run and the individual cycles in the run, including such information as obtainable speedup or number and average cost of node activations.

Our traces came from six OPS5 production systems: XCON (or R1), an expert system

for configuring computers; XSEL, an expert system to assist computer sales engineers; PTRANS, an expert system for factory management; Haunt, an interactive computer game; DAA, an expert system to design digital systems; and MUD, a system for diagnosing problems with drilling fluids. In addition, we measured the following tasks running in Soar: a part of R1, a part of XSEL, and the eight queens problem [Gupta 84].

We had four goals in performing our simulations:

- Measure the amount of speedup achievable from each source of parallelism individually, and enable comparing the extra speedup from a source against the overheads of using that source
- Identify bottlenecks in obtaining speedup from parallelism and propose means of eliminating them
- Determine the effect of different activation cost models on the amount of speedup obtainable from parallelism
- Evaluate the effect of architecture (shared memory vs. non-shared memory) on the speedup

Because of the way activity spreads throughout the network, offering opportunities for parallel processing at each node, we expected parallelism to increase system speed on the order of 100- to 1000-fold. Our simulation results showed, however, that it is possible to speed the match phase by up to only six-fold using production-level parallelism, up to eight-fold using node-level parallelism, and up to 14-fold using a combination of node-level and action parallelism. While the speedups obtained from parallelism were significant, they were much below our initial expectations. The main reasons for the limited speedup were (1) the small number of affected productions for each change to working memory (2) the large variance in the processing requirements of the production activations, and (3) the fact that successive changes to working memory affect almost the same set of productions [Gupta et al. 86].

While the first and third bottlenecks listed above are beyond the direct control of the person implementing the interpreter, we did develop a solution to the problem of variance in production-level processing requirements [Gupta 86]. To obtain more speedup, it is essential to decompose larger tasks into smaller tasks, each of which can be processed in parallel. This is exactly what node-level parallelism does: instead of evaluating one or more productions at a time, it evaluates several parts—in this case condition elements—of one or more productions at a time. Decomposing large tasks in this manner furthermore increases the "logical parallelism"—that is, the number of tasks that can be processed in parallel.

Even node-level parallelism, however, can leave periods of low parallel activity. We have discovered two causes for these periods of low concurrency: (1) Two-input nodes may require much more time to finish than other nodes in the network. This happens when a node has an unusually large number of stored tokens to examine. (2) A long chain of two-input nodes may have to be processed. This occurs when a token arrives at a two-input node and causes it to send out one or more tokens that pass through

many nodes below the originating one. Since each node in the chain has to perform a substantial amount of processing before its successor can be activated, the amount of parallelism that is possible in processing one of these chains is limited.

A solution to node-level slowdown is *intra-node* parallelism, that is, processing multiple activations of the same node in parallel. This is an even finer grain than node-level parallelism. Our strategy restricted this kind of parallel processing to tokens arriving at the same input of the two-input node, as simultaneously processing tokens from both inputs would greatly complicate the code. Exploiting this kind of parallelism carries even further the goal of reducing variance in processing productions. Based on our simulation results, we determined that a production system machine could divide the match process into a large number of small tasks, as we did by moving from processing whole productions, to processing parts of those productions at the node level, to processing activations of the same node in parallel.

4.2.2 Bounding parallel architecture alternatives

Based on the results of the simulation measurements, we proposed some general characteristics for a production system machine architecture. These included processor design and number, as well as memory and scheduling requirements.

Evaluating instruction set architectures

Our first goal was to identify the type of processor that could run the production system most quickly. Code sequences used to execute production-system programs do not include complex instructions. The instructions used most often are simple loads, compares, and branches without any complex addressing modes [Quinlan 86]. Because of the simple code sequences, we concluded that a machine for executing production systems should have a simple instruction set and execute instructions in as few clock cycles as possible. We calculated the time that several processors required to execute the code sequences of the Rete data-flow graphs. We estimated that a complex instruction set machine requires four to eight cycles per instruction. A reduced instruction set (RISC) machine, on the other hand, could execute most instructions in two machine cycles. The simple RISC machine thus promised to be two to four times faster than the more complex machine, making it the better processor choice [Quinlan 86].

We also explored the feasibility of gallium arsenide (GaAs) technology as a means of increasing production system execution speed. We designed a GaAs implementation as one of our simulated instruction set architectures [Lehr and Wedig 87]. Simulation results showed that a GaAs RISC machine could perform a machine cycle in about one tenth the time of a standard RISC machine. While current technology makes the actual implementation of such a system impractical, our investigation of a customized GaAs processor design allowed us to approximate an upper bound execution speed for a single processor using OPS5 or other production system languages.

Establishing an appropriate number of processors

Our measurements show that in both OPS5 and Soar production system programs the average size of the affect-set (i.e., the set of productions affected by a change to the working memory) is quite small, about 32 productions. Furthermore, our studies indicate that the average size of the affect-set is almost independent of the number of productions in the program. This result seems reasonable when we consider that programmers recursively divide problems into subproblems, and that at any given time the program execution corresponds to solving only one of these subproblems. The size of the subproblems is independent of the size of the original problem and primarily depends on the complexity of the subproblem and the complexity that the individual can deal with at the same time.

Since the majority of the match time is taken by the productions in the affect-set, the maximum speed-up that we can expect from production-level parallelism is a factor of approximately 32. This implies that if there is a separate processor performing match for each production in the program, only 32 processors will be performing useful work and the rest will have no work to do. There are a few production systems that can use up to 64 processors, so we concluded that 32 to 64 processors were the optimum number to use [Gupta 86].

Determining memory requirements

To achieve a high degree of speedup from parallelism, Rete exploits parallelism at a very fine grain. For example, multiple activations of the same node may be evaluated in parallel, requiring that multiple processors have access to the state corresponding to that node. It is not possible to replicate the state, since keeping all copies of the state up to date is extremely expensive. This situation strongly suggests a shared memory architecture. Another important reason for using a shared memory architecture relates to the load distribution problem. In case processors do not share memory, the system must decide on which processor to evaluate the activations of a given node at the time the network is loaded into the parallel machine. Since the number of node activations is much smaller than the total number of nodes in the Rete network, the system must assign several nodes in the network to a single processor. Partitioning nodes among the processors presents a difficult challenge. A shared memory architecture bypasses the partitioning problem since any processor can process any node activation, and the system can assign processors to node activations at runtime.

In shared memory architectures, the switch bandwidth between the processors and the main memory is always a concern. In order to reduce the needed bandwidth, each processor in the machine should have a cache and a small private memory. The cache has the usual function of reducing the amount of processor to memory traffic by holding copies of frequently-accessed words from main memory. The private memories further reduce the traffic by holding the parts of the data that can be replicated without introducing too much overhead. A production system machine can replicate working memory elements quite easily. Since working memory elements change only at well-defined points in the recognize-act cycle, it is not difficult to insure that all element copies remain consistent.

Using a special node scheduler

A single node may have several activations (that is, arrivals of new tokens) that it must process. It is not possible simply to assign each activation to a separate processor, since changes that one token causes to a node's saved state may conflict with changes that another token makes. Our goal was therefore to find a way to process in parallel those activations that did not conflict with each other. Our strategy was to develop a centralized task scheduler, where all node activations requiring processing may be placed and subsequently extracted by idle processors. Such a scheduler could be implemented as a very fast piece of special hardware. Hardware task schedulers are not flexible, however, in that they are not easy to change as algorithms evolve. Software task schedulers, on the other hand, offer flexibility, allowing for changes as we improve the system. To simplify experimentation, we chose a software strategy for our implementation [Forgy and Gupta 86].

4.2.3 Building a preliminary system

After establishing the general characteristics of our production system machine, be built a prototype system. We found four major issues in developing a parallel interpreter using the Rete algorithm: scheduling tasks, storing tokens before processing, locking hash tables, and considering how to achieve language-related system speedups.

Building the software task scheduler

In building our software task scheduler, we had to decide whether to make it active or passive. An *active* scheduler corresponds to an independent process to which messages for pushing and popping tasks may be sent. Once the processor has issued the request, it may proceed with what it was doing earlier. The requesting processor does not have to wait while its request is being processed. Active schedulers present a number of overheads, however. For example, scheduling a task involves sending a message to the active scheduler and then processing this request. Furthermore, when the processor sends a message to an active scheduler, the scheduler process may not be running and must be swapped in before the message can be processed. We needed a scheduler with fewer overheads. A *passive* scheduler, preferably a task queue, corresponds to an abstract data structure where node activations may be stored or retrieved using predefined operations like push-task and pop-task. Scheduling with a software task queue presents fewer overheads than an active scheduler, so we implemented the software task queue. If a task scheduler is not to be a bottleneck, however, it must be able to schedule a task within the period of about one instruction. Because it is not feasible to expect such performance out of a single software task scheduler, we used *multiple* software task queues to achieve reasonable performance [Gupta 86].

Storing tokens before processing

A second issue we faced in building the parallel implementation of a production system was how best to store tokens before processing. Existing OPS5 and Soar interpreters stored the contents of the memory nodes as a linear list of tokens. Thus when a token with a "-" tag arrived at a memory node, a corresponding token had to be found

and deleted from the memory node. Finding the corresponding token with a linear search would require, on average, a look up of half of the tokens in that memory node. Similarly, for an activation of a two-input node, the system must look up all tokens in the opposite input's memory to find the set of matching tokens.

Our goal was to store and retrieve tokens more efficiently. Making the cost of deleting a token from a memory node a constant, for instance, instead of being proportional to half the number of tokens in that node would make nodes with long lists of tokens as quick to process as nodes with short lists. Making the cost of finding matching tokens proportional to the number of successful matches instead of to the number of tokens in the opposite input's memory would likewise reduce processing time at each node. We solved the problem of token storage by implementing a hash table instead of a linear list. Using a hash table made the cost of deleting a token from a memory node a constant, and it made the cost of finding matching tokens in the opposite memory proportional to the number of successful matches. A hash table furthermore cut down the variance in the processing time required by the various memory node and two-input node activations, which is especially important for parallel implementations. The main disadvantage of using hashing is the overhead of computing the value of the hash function for each node activation. However, because hashing reduces the processing time variance, hash table-based memory nodes are best for parallel implementations [Forgy and Gupta 86].

Locking hash tables

Many resources in a parallel system have to be protected with mutual-exclusion locks: task queues, the active token count, the conflict set, etc. Most of these are relatively straightforward to protect (and a simple variation of standard spin locks is used), with the exception of locks used to handle hash tables for storing tokens in memory nodes. The problem here is that the system performs several kinds of operations on the hash tables: searching for matching tokens, adding and removing tokens, and adding and removing conjugate tokens (token pairs with identical working memory element pointers and opposite tag signs). Because of the importance of the hash tables to the performance of the system, we implemented and tested several locking schemes in order to develop one best suited for a wide variety of production system programs [Gupta et al. 87]. We describe two of these schemes here.

In the first scheme we gave each line in the hash table a flag to control its use (we define a "line" as a pair of corresponding buckets from the left and right hash tables along with their associated conjugate token lists). The flag takes on two values: "Free" and "Taken." When a process has to work with the hash table, it examines the flag for the line it needs, and if it finds the flag set to "Taken" it takes a different token from the task queue. This scheme works, but it becomes a bottleneck when several tokens arrive at about the same time for processing, all of which require access to the same hash table line.

The second, more complex, scheme permits several tokens to be processed in the same line at the same time, though some serialization of the processing is necessary

when destructive modifications to the lists of tokens are performed. This scheme requires two locks, a flag, and a counter for each line in the hash table. The flag takes on three values: "Unused," "Left," and "Right," to indicate respectively that the line is not currently being processed or that it is being used to process tokens arriving from the left or right. The counter indicates how many processes are using that line in the hash table; it is needed only so that the last process to finish using the line can set the flag back to "Unused." The first lock insures that only one process at a time can access the flag and the counter, so that tokens from two different inputs are not processed at the same time. The other lock insures that only one process at a time can modify the token lists. We expected the complex locks to benefit those programs that (1) generate multiple activations of the same two-input node from the same input, all requiring concurrent processing, and (2) have long lists of tokens in hash table buckets, where the complex locks help by allowing multiple processes to read the opposite input's memory at the same time. However, programs for which the above two conditions are not true may slow down because of the extra overhead caused by complex locks.

We implemented and tested both designs. The results are discussed in Section 4.3.1.

Language-based considerations in achieving speedup

Most production system interpreters at the time of our study were Lisp-based implementations. These tended to be slow: The FranzLisp implementation of the Rete interpreter for OPS5, for example, runs on a VAX-11/780 uniprocessor at around eight working memory element changes per second, while a Bliss-based implementation runs at around forty changes per second [Gupta et al. 87]. Part of the system slowness is due to the nature of the Lisp language. The system also lost a significant amount of speed because of node interpretation overheads. Our goal was to speed up the system by using a language better suited for our envisioned production system implementation and by reducing the node interpretation overhead level.

The computer language **C** is a faster language than Lisp. We solved the problem of the slow Lisp implementation by using a highly-optimized **C**-based implementation of OPS5 for the run-time interpreter. To handle the node interpretation overhead, we compiled the network directly into machine code, thus completely avoiding the interpretation problem. While it is possible to escape to the interpreter for complex operations during match or for setting up the initial conditions for the match, the majority of match is done without an intervening interpretation level [Gupta et al. 87].

4.3 Parallel Interpreter Implementations

After designing and building our parallel interpreter, we implemented two versions of it: One used our highly-optimized **C**-based OPS5 system, and the other added a Soar process to the first version.

4.3.1 Testing the OPS5 parallel interpreter

Because few good multiprocessor debugging tools exist, we began the debugging procedure on a uniprocessor, moved briefly to a small multiprocessor, and finally implemented the interpreter on our system of choice, the Encore Multimax, when it became available. We tested the parallel execution of the following three production-system programs:

- Weaver, a VLSI routing program with about 600 rules.
- Rubik, a program that solves the Rubik's cube with about 80 rules.
- Tourney, a program that assigns match schedules for a tournament with about 25 rules.

We chose Weaver because it represents a fairly large program and it demonstrates that our parallel OPS5 can handle real systems. Rubik is a smaller program that demonstrates some of the strengths of our parallel implementation, and the Tourney program demonstrates some of the weaknesses of our parallel implementation.

Initial uni- and multi-processor versions

We first used a MicroVAX-II uniprocessor to implement our highly-optimized, C-based version of OPS5. The speedup of this version compared to the FranzLisp-based OPS5 implementation was significant: almost 13 times faster for Weaver, 12 times for Rubik, and over 24 times for Tourney. We also used the MicroVAX to test differences in list-based and hash-based memories and found the time-saving effects of hash-based memories were substantial: approximately 15% for Weaver, 58% for Rubik, and 71% for Tourney over list-based memories [Gupta et al. 87].

Our final goal on the MicroVAX was to implement a task queue and get a rough idea of its overhead cost. A task queue is not necessary for a uniprocessor implementation and constitutes an overhead not offset by the parallelism possible with a multiprocessor implementation. After implementing the task queue, we concentrated on debugging rather than look for new speedups.

Before the Encore Multimax became available, we moved our parallel interpreter to a multiprocessor, the VAX-11/784 (four VAX-11/780 processors connected to shared memory). We implemented a parallel C-based version of OPS5 on this machine, but we did not test it extensively since it had only four processors, compared to sixteen on the Encore. We could thus compare results for far fewer parallel processes on the 11/784 and could not get as good a picture of the ultimate processing capability of our design. Instead we took this opportunity to further debug our parallel interpreter, on an actual multiprocessor instead of a uniprocessor.

Testing the Encore Multimax implementation

Finally, we ported the interpreter to the Encore Multimax. The version of the Encore Multimax available to us at CMU has 16 processors, each connected to the shared memory through a high-performance bus. The shared memory is equally accessible to all of the processors. The Multimax holds 32 Mbytes of memory and runs the Mach operating system. We tested several variations of the parallel OPS5 implementation on

the Encore. The variations were in the number of task queues that we used and in the locking structures used for hash table buckets. We ran each version using from one to thirteen match processes.

System speedup was disappointing when we implemented a single task queue and simple locks (allowing processing of only one token from the same hash table line at a time): When using only a single match process at a time, Weaver demonstrated a speedup of 1.02-fold, Rubik of 1.00-fold, and Tourney of 1.10-fold. When using thirteen match processes at a time, Weaver showed only a 3.90-fold increase, Rubik a 6.30-fold increase, and Tourney a 2.41-fold increase [Gupta et al. 87]. Possible reasons for the lack of speedup were contention for access to the single task queue and contention for access to the hash table buckets.

Our second and third versions explored the effects of removing these bottlenecks by using multiple task queues and a more complex hash table locking scheme. Using multiple task queues while retaining simple hash table locks increased system speed by removing some of the sequentiality imposed by the single task queue. The speedup was slight, however, for Tourney, although significant for Rubik and Weaver: Using thirteen match processes, Rubik showed a speedup of 11.42-fold as opposed to 6.30-fold in the single queue implementation [Gupta et al. 87]. Because our studies of differences in contention for task queues showed that Rubik had the largest such contention, increasing the number of task queues helped this bottleneck, causing Rubik's speedup.

Because Tourney's long lists of tokens in hash table buckets produce a large contention for hash table locks, we expected the program to benefit from our more complex locking scheme because the scheme allows simultaneous processing of several tokens from the same line in the hash table; thus potentially increasing system speed. We did not expect this scheme to help the other programs significantly since they do not produce the same contention for hash table locks. Our results showed that, while this scheme did reduce lock contention in all three programs, it provided only small speedup in the three programs. Weaver's and Rubik's small gains were not surprising, since they were not suffering from severe lock contention. However, Tourney's slight speedup (only 2.67-fold running thirteen processes, as compared to 2.30-fold with the simpler lock scheme) indicates that a complex locking scheme does not sufficiently reduce lock contention.

Our results therefore demonstrated that although task scheduling can be a bottleneck and must be handled by solutions such as multiple task queues, match-level exploitation of parallelism can provide significant speedups in production system execution.

4.3.2 Implementing a parallel Soar interpreter

Soar differs from OPS5 in that it uses a learning mechanism to add new productions to its knowledge base. These new productions, called *chunks*, later fire in appropriate situations, thus providing a learning-transfer mechanism. Large and complex systems

built in Soar execute their productions slowly, slowing down research and limiting such a system's utility. The dominating factor in this slowdown is the matching procedure: As chunking adds new productions, the demands on the matching procedure increase. It is thus important to speed up match as much as possible.

Results of our OPS5 implementation on the Encore (hereafter referred to as "PSM-E") suggested that Soar could benefit from using parallelism in match. However, Soar's chunking mechanism provides a new dimension in match parallelism that non-learning systems do not encounter. Chunking requires the ability to add productions at run time, but cheaply, since all the gains of a highly optimized system such as the PSM-E could be nullified by such overheads. [Tambe et al. 88]. Our goal was to adapt our parallel interpreter for use with Soar. To do this we had to enable the system to add productions at run time with very low computational cost.

Run time production addition

Adding a new production at run time on the PSM-E is a significant problem that requires the production's direct compilation into machine code. The major problem is how to keep code generation itself efficient so that encoding the new production does not become a serial bottleneck. Another important consideration in run time production addition is exploiting network sharing. The Rete network shares common tests and nodes among different productions to save work at run time. Sharing is especially important in Soar, since chunks are generated from the existing set of productions. To exploit the benefits of sharing, the system must therefore integrate the new code into the existing network instead of compiling the chunk as a separate piece of code.

Our strategy to increase chunk compilation speed and exploit the shared network was to employ two mechanisms, a tree data structure and a jump table. The tree data structure allows easy location of the points where sharing is possible in the network. The jump table maintains the link between any two sections of code where the code for a new node could in principle be inserted. The process of integration of the new code then reduces to changing entries in the jump table. We included this strategy as part of the run time system, providing a speedup of 20% in one of the test programs and 30% in another [Tambe et al. 88].

Run time update of state

A second overhead in adding productions at runtime stems from the fact that Rete is a state-saving algorithm: that is, it saves the partial results of match in various memory nodes in the network. When the system adds chunks at run time, the unshared memory nodes of the chunks are empty. The system must update the empty memories using tokens representing partial matches of working memory contents with the new production. The procedure updating the unshared memory nodes of the newly added chunk has to ensure that no duplicate state is added to memory nodes already containing the required tokens. The update procedure must not become a serial bottleneck by being very complex.

A simple method of updating the node memories for the new production would be to pass the contents of working memory back through the network and permit only those

node tasks associated with the new production to execute. In this way, the benefits due to parallelism in match could also be used to speed up the state updating process. However, some of the nodes associated with the new production are shared with the existing network, adding duplicate states to those nodes. To avoid adding duplicate states to memory nodes we confined the updating process to unshared nodes. Identification of unshared nodes is facilitated by the fact that the Rete net is *linear*, i.e. once one node in the production loses sharing, all its descendents remain unshared. Therefore, we used a simple node ID scheme to allow identification of the nodes to be updated: Nodes in the network all have incrementally-assigned unique ID numbers and a newly added node is always assigned an ID greater than any other existing node in the network. Thus identifying the IDs for the last shared node and the first new node allows the determination of all nodes that the system must update.

Results showed that exploiting a shared network using the node ID strategy reduced the update phase workload and produced an update phase speedup of about 20% for one test program and 25% for another. More importantly, update phase network sharing also benefited from parallelism. Using eleven match processes, all three test programs showed an update phase speedup of about three to five times over using a single processor.

Implementation and performance results

The Soar/PSM-E implementation of Soar on the Encore consists of one Soar process that maintains all its usual functionality except the matching capability, a PSM-E control process, and one or more PSM-E match processes. The number of match processes remains fixed for the duration of a particular run.

The Soar process is coded in Lisp, while PSM-E is **C**-based. The reason for running Soar as a Lisp process is that it has many man years worth of effort invested in coding, and an effort to convert Soar into **C** would have caused us to divert from our primary goal of investigating parallelism in the match. But since Lisp and **C** processes cannot share memory on the Encore, this arrangement causes some data structures to be duplicated in Soar and PSM-E. Further, the communication has to occur through UNIX pipes provided by the operating system.

Thus this Soar implementation uses PSM-E as a matching engine. Both Soar and PSM-E keep a copy of working memory. When Soar adds or deletes working memory elements, a message is sent to PSM-E to repeat those operations on its working memory elements. If this results in instantiations into the conflict set maintained on the PSM-E side, then PSM-E passes the instantiations on to Soar. Both Soar and PSM-E then fire these instantiations, updating their copies of working memory and repeating match. If new chunks are created, Soar passes them over to PSM-E at the end of the elaboration cycle.

We used three Soar programs to examine the various aspects of the implementation and the results of parallelism:

- Cypress-Soar, an algorithm design system with 196 productions. We chose a run that derives the quick-sort algorithm.

- Eight-puzzle-Soar, a system that solves the eight-puzzle mini task with 71 productions.
- Strips-Soar, a system that plans in the domain of robot control with 105 productions.

We tested the Soar implementation before and after chunking. Before chunking, the learning mechanism is not turned on, so that the system performs like any production system. During chunking, the learning mechanism is turned on, so that the system is in the process of learning and creating new chunks. After chunking, the same program is run again after having created chunks with that input. It should run faster since it knows more about possible solutions to the problem now than it did on the first run.

When we ran the programs before chunking, we found low speedups. The best speedups for Cypress and Strips were with seven match processes: Cypress ran 3.51 times faster, while Strips ran 2.15 times faster. Eight-puzzle ran fastest with five match processes, but only increased speed 1.70-fold [Tambe et al. 88]. The causes of the low speedups were the slow execution rate of working memory changes by the PSM-E control process and the spurious overheads of paging and other system-related activities. The PSM-E control process is responsible for all working memory element changes in an elaboration cycle. However, since the control process has to simultaneously communicate with Soar, its rate of execution of working memory changes is reduced, and this reduces the available parallelism. We compensated for this factor by changing the behavior of the system to start match after the PSM-E control process completes the working memory element changes in each cycle. The low overheads of the operating system become significant because the total run time of the match processes is sometimes reduced by parallelism to about 10 seconds. After taking these factors (the low rate of working memory element changes and the system time) into account, we found that parallelism increased in all three programs by a factor of about two to three.

Running the programs after chunking also demonstrated system speedup. Using eleven processes, Cypress ran 6.78 times faster than using a single process, Strips ran 7.28 times faster, and Eight-puzzle ran 8.96 times faster [Tambe et al. 88].

Our results thus demonstrated that exploiting match-level parallelism can provide very good speedup in a system capable of learning.

4.4 Bibliography

[Forgy and Gupta 86]
Forgy, C. and A. Gupta.
Preliminary architecture of the CMU production system machine.
In *Proceedings of Hawaii International Conference on System Sciences*, University of Hawaii, January, 1986.

PSM, the Carnegie Mellon University Production System Machine, is being designed to execute rule-based production systems as efficiently as possible. It is planned that the machine will contain 32 to 64 relatively powerful processors, a large shared memory, and a hardware device to perform run-time scheduling of tasks for execution. The machine will run a parallel variant of the Rete algorithm, which is currently used in most sequential implementations of production systems. This paper describes the architecture of the machine in detail and discusses how the Rete algorithm has been adapted for parallel execution.

[Gupta 84]
Gupta, A.
Parallelism in production systems: the sources and the expected speed-up.
Technical Report CMU-CS-84-169, Computer Science Department, Carnegie Mellon University,
December, 1984.

Production systems (or rule-based systems) are widely used in Artificial Intelligence for modeling intelligent behavior and building expert systems. On the surface production systems appear to be capable of using large amounts of parallelism-it is possible to perform match for each production in parallel. Initial measurements and simulations, however, show that the speed-up available from such use of parallelism is quite small. The limited speed-up available from the obvious sources has led us to explore other sources of parallelism. This paper represents an initial attempt to identify the various sources of parallelism in production system programs and to characterize them, that is, to determine the potential speed-up offered by each source and the overheads associated with it. The paper also addresses some implementation issues related to using the various sources of parallelism.

[Gupta 86]
Gupta, A.
Parallelism in production systems.
Technical Report CMU-CS-86-122, Computer Science Department, Carnegie Mellon University,
March, 1986.

Production systems (or rule-based systems) are widely used in

Artificial Intelligence for modeling intelligent behavior and building expert systems. Most production system programs, however, are extremely computation intensive and run quite slowly. The slow speed of execution has prohibited the use of production systems in domains requiring high performance and real-time response. This thesis explores the role of parallelism in the high-speed execution of production systems.

On the surface, production system programs appear to be capable of using large amounts of parallelism - it is possible to perform match for each production in a program in parallel. The thesis shows that in practice, however, the speed-up obtainable from parallelism is quite limited, around 10-fold as compared to initial expectations of 100-fold to 1000-fold. Since the number of productions affected and the number of working-memory changes per recognize-act cycle are not controlled by the implementor of the production system interpreter, the solution to the problem of limited speed-up is to some how decrease the variation in the processing cost of affected productions. The thesis proposes a parallel version of the Rete algorithm which exploits parallelism at a very fine grain to reduce the variation. It further suggests that to exploit the fine-grained parallelism, a shared-memory multiprocessor with 32-64 high performance processors is desirable. For scheduling the fine-grained tasks consisting of about 50-100 instructions, a hardware task scheduler is proposed.

The thesis presents simulation results for a large set of production systems exploiting different sources of parallelism. The thesis points out the features of existing programs that limit the speed-up obtainable from parallelism and suggests solutions for some of the bottlenecks.

[Gupta et al. 86] Gupta, A., C. Forgy, A. Newell, and R. Wedig.
Parallel algorithms and architectures for rule-based systems.
In *Proceedings of the 13th International Symposium on Computer Architecture* , June, 1986.

Rule-based systems, on the surface, appear to be capable of exploiting large amounts of parallelism_ it is possible to match each rule to the data memory in parallel. In practice, however, we show that the speed-up from parallelism is quite limited, less than 10-fold. The reasons for the small speed-up are: (1) the small number of rules relevant to each change to data memory; (2) the large variation in the processing required by the relevant rules; and (3) the small number of changes made to data memory between synchronization steps. Furthermore, we observe that to obtain this limited factor of 10-fold speed-up, it is neces-

sary to exploit parallelism at a very fine granularity. We propose that a suitable architecture to exploit such fine-grain parallelism is a bus-based shared-memory multiprocessor with 32-64 processors. Using such a multiprocessor (with individual processors working at 2 MIPS), it is possible to obtain execution speeds of about 3800 rule-firings/sec. This speed is significantly higher than that obtained by other proposed parallel implementations of rule-based systems.

[Gupta et al. 87] Gupta, A., C.L. Forgy, D. Kalp, A. Newell, and M. Tambe.
Results of parallel implementation of OPS5 on the Encore multiprocessor.
Technical Report CMU-CS-87-146, Computer Science Department, Carnegie Mellon University,
August, 1987.
Anoop Gupta is now a member of the Computer Science Department, Stanford University.

Until now, most results reported for parallelism in production systems (rule-based systems) have been simulation results -- very few real parallel implementations exist. In this paper, we present results from our parallel implementation of OPS5 on an Encore multiprocessor with 16 CPUs. The implementation exploits very fine-grained parallelism to achieve significant speed-up. Our implementation is distinct from other parallel implementations in that we attempt to parallelize a highly optimized C-based implementation of OPS5. This is in contrast to other efforts where slow lisp-based implementations are being parallelized. The paper discusses both the overall structure and the low-level issues involved in the parallel implementation and presents the performance numbers that we have obtained.

[Lehr 86] Lehr, T.F.
The implementation of a production system machine.
In *Proceedings of the Nineteenth Annual Hawaii International Conference on System Sciences*, University of Hawaii, January, 1986.
Also available as technical report CMU-CS-85-126.

The increasing use of production systems has drawn attention to their performance drawbacks. This paper discusses the architecture and implementation of a uniprocessor OPS production system machine. A brief tutorial on the OPS production system and its Rete algorithm introduces salient issues that temper the selection of a uniprocessor architecture and implementation. It is argued that general features of Reduced Instruction Set Computer (RISC) architectures favorably address these issues. The architecture and a RTL description is presented for a pipelined RISC proces-

sor designed specifically to execute OPS. The processor has a static branch prediction strategy, a large register file and separate instruction and data fetch units.

[Lehr and Wedig 87]
Lehr, T.F. and R.G. Wedig.
Toward a GaAs realization of a production system machine.
Computer, April, 1987.
In this article, we attempt to demonstrate the issues involved in realizing a gallium arsenide (GaAs) processor designed for efficient execution of the OPS5 production system language. We review the state of GaAs D-MESFET technology, which is a mature technology, and discuss how its capacities can be exploited by a reduced instruction set computer (RISC). Our work is to investigate the issues involved in realizing a RISC processor in GaAs to obtain estimates of parameters like the cycle time and the basic system requirements of such a processor. Ours was a feasibility study, and the design has not been implemented; however, through this work, we have been better able to determine the feasibility of GaAs as a system-realization technology, and we have helped to push back the limits of the execution speed of production-system programs.

[Oflazer 84]
Oflazer, K.
Partitioning in Parallel Processing of Production Systems.
In Keller, R., Editor, *Proceedings of the International Conference on Parallel Processing*, ACM, IEEE, and Department of Computer and Information Science, Ohio State University at Columbus, August, 1984.
The results of an analysis of production level parallelism in OPS5 production system programs is presented. The results indicate that contrary to most expectations, the effective production level parallelism in this class of production systems considered is very low compared to the number of productions in these systems. Hence, significant speed-ups in executing such systems would be obtained by combining the limited parallelism with fast hardware and overlapped processing; rather than by massively parallel approaches employing simple processors. Later, the problem of partitioning productions in a production system to a small number of processors in a parallel processing system is presented. The goal of partitioning is to improve the speed-up provided by the limited parallelism by finding assignments of productions to processors that achieve a more balanced load for each processor.

[Quinlan 86] Quinlan, J.
A comparative analysis of computer architectures for production system machines.
In *Proceedings of the Nineteenth Annual Conference on System Sciences*, ACM and IEEE, January, 1986.
This paper reports the results of research concerning the effect of a uniprocessor's architecture on the performance of production systems. A number of uniprocessors, both existing and proposed, are analyzed with respect to their execution of a production system interpreter known as OPS5. By using measured run-time statistics of existing production systems, the performance of each uniprocessor is calculated and analyzed. The results show that the performance gains of a specialized architecture over a conventional architecture can be significant.

[Tambe and Newell 88]
Tambe,M., and A. Newell.
Why some chunks are expensive.
Technical Report CMU-CS-88-103, Computer Science Department, Carnegie Mellon University,
January, 1988.
Soar is an attempt to realize a set of hypothesis on the nature of general intelligence within a single system. One central hypothesis is that chunking, a simple experience-based learning mechanism, can form the basis for a general learning mechanism. It is already well established that the addition of chunks improves the performance in Soar a great deal, when viewed in terms of subproblems required and number of steps within a subproblem. But this high level view does not take into account potential offsetting costs that arise from various computational effects. This paper is an investigation into the computational effect of expensive chunks. These chunks add significantly to the time per step by being individually expensive. We decompose the causes of expensive chunks into three components and identify the features of the task environment that give rise to them. We then discuss the implications of the existence of expensive chunks for a complete implementation of Soar.

[Tambe et al. 88] Tambe, M., D. Kalp, A. Gupta, C.L. Forgy, B. Milnes, A. Newell.
Soar/PSM-E: investigating match parallelism in a learning production system.
In *Proceedings of Parallel Programming Environments: Applications, Languages, and Systems (PPEALS)*, July, 1988.
Soar is an attempt to realize a set of hypotheses on the nature of general intelligence within a single system. Soar uses a production system (rule based system) to encode its

knowledge base. Its learning mechanism, chunking, adds productions continuously to the production system. The process of searching for relevant knowledge, matching, is known to be a performance bottleneck in production systems. PSM-E is a C-based implementation of the OPS5 production system on the Encore Multimax that has achieved almost linear speedups in matching. In this paper we describe our implementation, Soar/PSM-E, of Soar on the Encore Multimax that is built on top of PSM-E. We first describe the extensions and modifications required to PSM-E in order to support Soar, especially the capability of adding productions at run time as required by chunking. We then present speedups achieved in the match on the Soar/PSM-E and discuss some effects of chunking on parallelism. Finally, we point out the factors that limit parallelism on Soar/PSM-E and discuss the work in progress to deal with some of them.

5. THE CAT EXPERT SYSTEM PROJECT

Our goal for the CAT (Command Action Team) project was to develop an expert system to monitor and assess potential threats against a carrier group and to recommend possible actions for countering those threats. For a system to carry out these tasks it must have a certain amount of basic knowledge about situations and objects it might encounter. It also has to accept new information and make inferences about the situation based on that information and on the knowledge it already has. Finally, it must interact with a human user in order to communicate its knowledge and to expand its knowledge about situations it might encounter.

Our research during the contract built on a preliminary working prototype that could make inferences using incoming information and its own knowledge base. The prototype could also interact on a limited basis with a human user, sending simple warning messages and responding to a user's requests for summaries and explanations. Because the CAT system could ultimately contain thousands of rules and would have to process incoming information in real time, however, our goal in this period was to significantly improve its efficiency at handling new information and making inferences. Furthermore, because the system was to cooperate with a human expert, we wanted to make interaction with the system more flexible and expressive than in the prototype.

Our work focused on three areas:

- CAT's internal machinery, specifically emphasizing speed and efficiency of knowledge-base maintenance
- CAT's external interface, emphasizing flexible alert and knowledge acquisition systems
- System-testing tools, including a smaller version of the system and scenarios to simulate real-life situations

5.1 Developing the Internal System

CAT uses its basic knowledge about objects and situations it might encounter to assess new information it receives from the outside world. We first briefly describe the mechanisms the original system used to perform these tasks, then address the issues we faced in improving these mechanisms: increasing the mechanisms' speed and efficiency by controlling the size and expansion of the inference network.

5.1.1 Structure and maintenance of CAT's knowledge base

CAT is a production system containing a permanent memory of *productions* (heuristic condition-action rules) and a working memory composed of currently active assertions supported by reasons and evidence. An assertion contains one piece of information about an object or event in the current tactical situation: One assertion might give a ship's length, for instance, while another might give the ship's type or its name. CAT

builds a dependency structure called the *inference net* among assertions, reasons, and evidence. It does this by firing productions from its knowledge base called *tactical inference rules.* A tactical inference rule contains condition elements in the form of assertions on its left hand side. The system attempts to match these condition elements with inference net assertions. When all the condition elements are successfully matched, the rule can fire, creating a new assertion and the reason and evidence elements that support it.

The *inference engine* controls the system. It contains the code that accepts and transforms incoming information from a data communication link into *report* elements. It then transforms report elements into assertion elements and inserts these assertions as well as consequent assertions created by tactical inference rules into the inference net.

5.1.2 Improvement of inference net maintenance rules

CAT must monitor numerous objects: surface ships, submarines, aircraft, land bases, satellites, etc. For each object, updates of position, movement, and activity may come frequently, perhaps 10 per second. After the system handles the reports, extracts their knowledge, and updates CAT's world picture, an even greater task lies in propagating the effects of these updates throughout the inference net. Any new piece of information may have far-reaching implications that affect the interpretation of current system intelligence.

We observed that the system spent an exceptionally long time updating the inference network on the basis of incoming reports, a problem we called "choking." Handling new information and propagating its effects were such severe bottlenecks that we devoted substantial attention to obtaining significant decreases in the amount of computation demanded by these tasks in order to speed up the system.

A major function of inference net maintenance rules is pruning outdated or unnecessary assertions. If this is not done, the size of working memory grows monotonically with time, severely degrading system performance and eventually exceeding resource limitations. We therefore concentrated on alleviating choking via three different mechanisms:

- Pruning outdated information
- Pruning repetitious reports
- Matching only on reports closely related in time

Pruning outdated information

In deciding how much old data to discard, we first pinpointed three situations where we could safely discard information. These situations were:

- The subject of the assertion is no longer of interest
- The assertion has been superseded by a more recent assertion from the same source
- The information in an assertion is so old that it can no longer be trusted

We furthermore had to keep in mind that certain information changes constantly, potentially causing constant slowdowns as the system updates the inference net. Constant reports of a ship's new position, for example, could slow the system down, yet changes in a ship's position can be useful in inferring a ship's tactics. We therefore did not want to remove too many of these assertions.

Our concerns in discarding old data were to achieve the greatest improvement in system speed with the least degradation of system performance. To determine the course of a ship, it is necessary to have at least three descriptions of position. To determine the speed of a ship, three such descriptions are also useful. Because three seemed to be the necessary number in these cases, we proposed keeping only three assertions of a particular type at a time. Experimental results using this strategy showed a significant gain in system speed while demonstrating the same accuracy as the original system. We therefore implemented this design in our system.

Pruning similar reports

The system receives frequent updates about objects and events. Some of this information does not change significantly over time. For example, a ship's length remains the same from one update to the next. If it is moving very slowly, its position likewise will not change significantly. Incorporating such repetitious information into the inference net slows the system down unnecessarily. Our goal was to control the unnecessary slowdown caused by this repetitious information. To do this, we developed a method of ignoring superfluous information about an object by evaluating report elements and deleting those that resemble information already established in an assertion. For example, if a new report element described a ship's position, CAT checked that ship's current position assertion. If the new information was the same or very close to the older information, CAT deleted the new report element. We tested this strategy, and because it significantly increased the system's efficiency, we implemented it in the system.

A problem with this strategy is that deleting repetitious information can prevent a weak assertion from becoming stronger. That is, some assertions are less certain than others and can be strengthened with new information. When the system creates an assertion, it assigns it a *confidence factor* which determines how reliable that assertion is. The system computes the confidence factor based on the confidence factors of previous assertions upon which the assertion is built. If an assertion is based on a weak inference, it receives a low confidence factor and must be corroborated by subsequent updates to become more certain. For example, a ship may be *thought* to be of a certain nationality simply because it is near another one whose nationality has been established. This is therefore a weak inference that could become stronger with more information. Our strategy did not distinguish between weak and strong inferences when it deleted new information. This meant that information that could raise a weak assertion's confidence factor would be deleted if it was similar to information that had already been reported. Future research into deleting similar reports might address this problem. A possible solution is to use a lower threshold on confidence factors to determine which assertions receive updates. If an assertion already has a high confidence

factor, so that any update of that assertion will not increase its certainty, then that update should not be made, since it would be a waste of system resources. If, on the other hand, the assertion is less certain (has a low confidence factor), it should continue to receive updates to increase its certainty.

Making the best time matches

The system constantly receives information updates, each with a new observation time. Since an update can cause a rule to fire as the update satisfies a condition element, the same rule could fire repeatedly with every update. Repeated rule firings are not necessarily a problem: As the system acquires more information about an object or event it *should* be able to demonstrate its increased knowledge and confidence about that object or event. An unnecessary system slowdown occurs, however, when a rule fires because two assertions that are not the best time matches fulfill the rule's conditions.

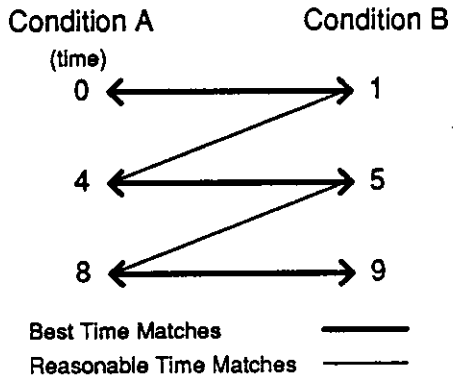

Figure 5-1: Possible time matches

For example, conditions A and B, shown in Figure 5-1, receive updating reports at the times indicated in the figure. The *best* time matches are at (0, 1), (4, 5), or (8, 9). The system also allowed, however, the *reasonable* time matches at (4, 1) or (8, 5). Our goal was to allow the system to make only the best time matches, ignoring the other possible matches.

We added code that performed an extra test on condition elements so that the system accepted only the best time matches. We then tested the design. Since the system ran about five times faster using this method, we implemented this design.

5.1.3 Studying alternative data representations

The large number of assertion working memory elements created during execution of the CAT system (one for each assertion) is taxing on the matching process. The matching process is further slowed by the need to check that all assertions being tested refer to the same object. Our goal was to devise an alternative data representation that reduced the number of working memory elements the system had to handle. Our strategy was to store assertions about a single object within a single working memory element. This significantly reduced working memory clutter, thus potentially reducing matching time overhead. Furthermore, this strategy saved system resources, since the system did not have to test that all assertions belong to the same object. We called our new system OBJCAT.

Working memory elements of type *object* and *reason* compose OBJCAT's inference net, in contrast with the original system's assertion, reason, and evidence elements. Assertions, no longer represented by an individual working memory element, are kept in an array within the object element to reduce the amount of searching the system has to perform. Slots in the array are not reserved for particular attributes. Assertions are assigned to array elements as they arrive. A set of access functions extracts values, assigns values, and tests for matches using the object elements.

We built and tested OBJCAT. The actual time needed to run the scenario was about 40% less than that of CAT. The new data representation had a dramatic impact on the size of working memory, reducing the mean number of working memory elements by 19 times. The mean size of the conflict set, conflict resolution time, and rule fire time were also all significantly reduced.

While the new data representation succeeded in significantly speeding up the system, we discovered two issues that warrant further investigation. The issues stemmed from deficiencies in OPS83, not from the design strategy of multiple assertions within a single object element. For each attribute in an object element, OBJCAT stores the last three assertions and matches assertions with condition elements using an OPS83 function call that can find only the most recent assertion. The original CAT system uses a temporal sequence of assertions to determine the speed of a ship, for instance, or to upgrade a confidence factor. Further research could design a way to prevent wasting the two earlier assertions in OBJCAT by letting the system use *all* reasonable assertion matches, instead of only the most recent.

OBJCAT's other area for potential improvement was match speed. Matching condition elements and assertions took longer than in the original system because the matching functions had to look down an array of assertions. Future implementations of the system could use hashing functions instead of searching through the array, or each object element could contain lists of all attribute names and slots to store the values of those attributes instead of simply assigning assertions to array elements as they arrive, as in the current OBJCAT system.

5.2 Developing the External System

Efficient interaction between a human user and CAT is essential: The user must quickly receive important information such as potential threats to the carrier group. The expert must furthermore be able to transmit his knowledge about a tactical situation to the system. To achieve these goals, we improved the existing alert system and created a knowledge acquisition system, facing issues of flexibility and ease of interaction.

5.2.1 Developing the alert facility

A major function of the CAT system is to warn the user of possible threats to the carrier group. To be effective, the system should allow the user to define the conditions under which to deliver a warning. It must also dispatch timely warnings about a wide variety of situations.

Our prototype featured an alert-generating module whose function was twofold: It allowed the user to request a warning when the system made an assertion containing a single value, that is, one condition element, that he specified, and it warned the user when the system made that assertion. For example, the user could tell the system that he wanted a warning when a Soviet ship was in the area, and each time the system inferred that that was the case, the alert facility warned the user by writing that assertion to the screen.

Our goal was to achieve a much more flexible and efficient alert system. We designed a system that met this goal in several ways. First, we made it capable of accepting assertions with more than one condition. Instead of receiving a warning only about a Soviet ship, for example, the user was able to ask for notification of a Soviet ship with a range of less than four miles, thus increasing system flexibility. We made the alert mechanism more efficient by enabling it to determine whether it was making an assertion about an alert situation for the first time or whether it was updating an earlier alert situation, thus reducing potential confusion for the user. We further increased the efficiency of the alert system by allowing the user to store the specifications he gives the system for receiving a warning, thus saving time on future runs because this information does not have to be re-entered. Subsequent users could, however, adapt these "built-in" alert specifications if necessary.

We implemented these design changes in our alert system and verified its operation. We then shipped it to our colleagues at NOSC, who implemented it with only minor adjustments.

5.2.2 Developing an automatic knowledge acquisition system

The reasoning power that CAT demonstrates is not a simple by-product of raw computer power, but instead derives from the knowledge-based approach that characterizes production systems. Knowledge is represented as a set of rules that embody the knowledge of human experts. Construction of the CAT expert system depended criti-

cally on extracting this often implicit knowledge from expert informants, formalizing this knowledge insofar as possible, and then expressing the knowledge as production rules—a process called *knowledge engineering*.

Once we had stabilized CAT's design, we were ready to address the goal of facilitating the knowledge engineering process, an important goal because knowledge engineers are in short supply. A second goal was to develop an efficient way of readjusting the tactical inference rule knowledge base so that the system could still use it after each improvement to the CAT inference engine. To achieve these goals we developed a system knowledgeable about its surroundings. It used that knowledge to help the expert enter new information. Our system furthermore kept the new information in a form that could be easily transformed as we made changes to the inference engine.

Acquiring knowledge intelligently

Our knowledge-acquisition system, SKAT (Smart Knowledge Acquisition Tool), allows a user to enter his situation knowledge in the form of tactical inference rules, editing existing rules or creating new ones. When the user invokes the "teach" program, SKAT interviews him with the goal of defining a new rule. The user specifies the conditions and conclusion of a rule in a formal command language that is an English-language subset. The user can later invoke the "generate" operation to translate the rule into a form that the current CAT inference engine can actually use.

SKAT's knowledge base

Before the user begins editing or creating a rule, SKAT loads a file that gives it *domain* knowledge about objects the carrier group might encounter. This file describes and categorizes objects, establishing their conceptual relationships using three kinds of nodes. Concept nodes represent several levels of object categorization. Categories at the most general concept level include platform and weather. A "platform" node, in turn, may connect with more specific concepts "aircraft," "ship," and "submarine," all of which specify types of platforms. The network contains its most specific information in the object nodes, each of which represents an individual physical object type. For example, the concept node "aircraft" connects with object nodes "bomber," "fighter," "helicopter," "reconnaissance," and "tanker."

Predicate nodes may also connect with concept nodes. A predicate is a way of describing a concept: For example, a ship may have a length or size or type. SKAT's domain knowledge file shows the connections between a concept and the predicates that describe it, thus establishing what kinds of predicates may be associated with certain concepts. For example, the three types of platforms (aircraft, ship, and submarine) may be described in terms of the predicates position, speed, course, maximum range, etc. They may not be described in terms of start time or end time, though, which are predicates applied to weather. An object inherits predicate attributes of those concepts above it in the object hierarchy: thus, if an object node represents a bomber, the bomber may be described in terms of the predicates position, speed, etc. without having to establish direct connections between the object node "bomber" and the predicate node "speed." Thus the domain knowledge file presents a network of relationships among categories of objects and their properties.

Easing user effort

The more SKAT knows about the situation the expert is describing, the more SKAT can aid the expert in formulating the new rule. Our goal was thus to make SKAT's interviewing process knowledgeable about situations the expert would then make new rules about. We made SKAT capable of exploiting different sources of information, including knowledge about:

- The current CAT system architecture
- The domain, using information contained in the domain knowledge file
- Constraints implied by already-specified parts of the rule being written (for example, if the rule concerns an airplane, both user and SKAT will know that the rule may not contain any assertions about its depth).

In an early version of SKAT, the domain knowledge file could not easily be altered because doing so required checking by hand the many possible relationships between objects, making sure the proper links among concepts, objects, and predicates were maintained. The highly interrelated nature of the contents of this file therefore made it difficult for us to add to it and expand the system's knowledge about the world. Our goal was to allow SKAT to accept new domain information easily, without the time-consuming task of checking all the interrelationships among the objects. To achieve this goal, we adapted the code so that SKAT drew upon its knowledge of the domain to establish automatically the proper connections between newly-established nodes and the appropriate concept and predicate nodes. For example, if the user wanted to add a new kind of airplane to the domain knowledge file, SKAT made certain that the new object node connected with the aircraft node, which, in turn, connected with the related predicate nodes. Establishing the proper relationships in this way also aided the expert when he wrote a rule concerning the new object. Because the relationships had been established, the system could appropriately prompt the expert as he entered the new rule. This method of automatically updating relationships as new objects were entered thus made it easier to use SKAT than in the preliminary version.

Restructuring tactical inference rules

As we improved the CAT system, changing the way the inference net maintenance rules functioned in an effort to speed up the system, we had to ensure that the knowledge contained in the tactical inference rules remained usable. Each change in the inference engine code could affect the way the system processed tactical inference rules. For example, differences in data representation between CAT and OBJCAT mean the programmer must alter the way tactical inference rules' condition elements are matched with assertions. Changing each tactical inference rule by hand to reflect the changes in the inference engine code would be a tedious and error-prone process. Thus, our goal was to allow rapid restructuring of CAT's knowledge base to adjust to the changes in the inference engine. We designed SKAT so that it not only allows the user to enter and store tactical inference rules in an implementation-independent fashion, but it also allows automatic restructuring of these rules. Instead of requiring a knowledge engineer to change each rule, the only change that needs to be made is to the portion of SKAT that transforms the implementation-independent representation of the rules into

OPS5 or OPS83 target rules. This feature of SKAT proved useful as we made changes to the inference engine.

5.3 Developing System-Testing Tools

5.3.1 Developing demonstration scenarios

Development of CAT's knowledge base created a need for a way to test it in a real-life situation. We could not use the actual carrier group for testing the system because of the group's remote location and because we didn't have access to the classified knowledge base that NOSC actually implemented. To simulate a real-life situation for testing CAT, we created a scenario consisting of lengthy and complex report sequences describing relevant objects and actions such objects might take. We adapted unclassified information supplied by NOSC to create this scenario. After building the scenario we used it in testing the changes we made to the system.

A single scenario used only a limited amount of CAT'S knowledge base, so we developed additional test scenarios. In implementing these scenarios, we improved CAT's performance because we used more of the rule base than we had before and therefore were able to find and debug more problematic rules.

5.3.2 LEANCAT

As we improved the CAT system, we needed to test the changes we made before actually implementing them. Testing the changes using the complete system, which consisted of close to a thousand rules, proved time-consuming. Our answer to this problem was to reduce by half the number of rules in the system. We called this smaller testing version of CAT "LEANCAT."

Our goal in creating LEANCAT was to remove the features that slowed the system down. We did this by taking out inference rules unnecessary for the current scenario. We also deleted some features, like the briefing module, or simplified others, like the alert facility. A third method for trimming the system was to remove some of the code for computing confidence factors.

We built and tested LEANCAT, demonstrating a significant speedup in rule firing time for the new system over the larger version of CAT. This speedup was caused almost exclusively by fewer rules firing. We used LEANCAT for much of our subsequent efficiency testing, since we were able to test the new designs more quickly using LEANCAT than using the original system.

5.4 Cooperation with NOSC

A critical aspect of our research was the close cooperation it demanded between researchers at CMU and the Naval Ocean Systems Center (NOSC). NOSC's role in the project included:

- Installing the CAT software on the USS *Carl Vinson.*
- Integrating the CAT system with other software (Computer Corporation of America's SDMS System).
- Responding to the needs of the *Vinson* leadership.
- Acquiring and integrating domain knowledge, often of a classified nature, into the CAT system.

In addition, NOSC sent members of its CAT team to Carnegie Mellon to work for extended periods of time on the CAT project. We intended that, while their personnel would benefit from working in an advanced research laboratory, they could in turn provide for us useful work, ideas, and domain knowledge.

At appropriate times, we shipped our work on CAT to NOSC. This occurred at the beginning of NOSC's involvement with CAT and about twice per year thereafter. NOSC modified the CMU systems extensively. Their major contributions to CAT functionality included the development of a remote user interface for a Sun workstation and work with the *Vinson*'s carrier group on tactical situation analysis. In addition, NOSC's alert mechanism served as the basis for the alert mechanism we ultimately developed. Finally, NOSC personnel augmented CAT's knowledge base by writing additional tactical inference rules. The NOSC version of CAT was, in turn, stripped of classified material and delivered to Carnegie Mellon University about twice per year.

While both sides thus benefited from the exchange of expertise during the contract period, we experienced some communication difficulties that similar cooperative efforts would do well to address at the start. Our goal was to develop an expert system with a deeper level of intelligence than any other system at the time. To develop a more intelligent system we had to know as much as possible about the domain knowledge the system would be working with. Because much of the actual domain knowledge was classified, however, we had to build the system using false data whose similarity to the real data we could only guess.

While we had no interest in obtaining security clearance in order to have access to the real data, we would, however, have found understanding more about the *structure* of the actual domain knowledge useful in exploring representational problems, even if the *content* remained secret. The project could have benefited had both sides developed a level of communication such that we could get the information we needed to make a more intelligent system without breaching NOSC's security restrictions. Future participants in similar cooperative projects could improve such a situation with coaching on how to ask the kinds of questions and how to give the kinds of answers necessary to satisfy the needs of both sides.

I. GLOSSARY

act — The step in a recognize-act cycle where the production selected during conflict resolution is "fired," potentially changing working memory contents.

address space — A memory location, or a numerical range that specifies a memory location.

assertion — A working memory element in the CAT expert system. An assertion contains one piece of information about an object or event in the current tactical situation.

basic code block — A sequence of instructions that are always executed together. A basic block has no alternate entry or exit points: Control may enter only through its first instruction and leave only via its last instruction.

chunks — New productions added in a Soar system as a result of learning.

conflict resolution — The step in a recognize-act cycle where one of the satisfied productions from the conflict set is selected for firing.

conflict set — The set of all productions whose condition elements have been successfully matched against working memory elements.

copy-on-write — A Mach feature in which an address space is physically copied to a separate memory location only if a process writes to that address space.

function decomposition — The means by which a computing task is divided for execution on a multiprocessor system.

heterogeneous programs — A variety of systolic processing in which individual cells may execute different programs.

homogeneous programs — A variety of systolic processing in which all array cells execute a copy of the same cell program.

inference engine — CAT system control mechanism containing code that accepts and transforms incoming data and inserts new assertions into the inference net.

inference net — Network in the CAT system representing dependencies among assertions, evidence, and reasons.

interprocess communication (IPC) — The means by which processes exchange data or messages, either on a single host or over a network.

intra-node parallel processing — Processing multiple activations of the same node concurrently.

iWarp — The next-generation Warp cell, a single-chip, VLSI implementation begun in 1986.

left-hand side The "if" part of a production representing the conditions necessary to evoke a group of relevant actions.

Mach A distributed multiprocessor operating system developed at CMU-CSD.

match The step in a recognize-act cycle where the system matches a production's condition elements against working memory contents.

memory object A kernel-managed data repository: a port, physical memory, or disk space.

message A typed collection of data objects used for interprocess communication.

Matchmaker An interface language that allows processes to communicate across a distributed network regardless of the architecture or language of the sending and receiving machines.

node activation Arrival of new data (a token) at a node for processing.

node-level parallel processing Processing activations of two or more nodes concurrently.

physical copy A copy of an address space which is passed by copying that address space into the address space of the receiving process.

port A kernel-protected message queue.

process Any activity executed by the CPU. Under Mach, a process comprises a thread operating within the context of a task.

PSC Programmable Systolic Chip: A high-performance, special-purpose, single-chip microprocessor intended to be used in groups of tens or hundreds for the efficient implementation of a broad variety of systolic arrays in several application areas.

recognize-act cycle The three steps involved in firing a production system rule: match, conflict resolution, and act.

Rete An algorithm used in production systems for exploiting two sources of redundant computation: slow change of working memory and repeated condition elements. The Rete network represents the current contents of working memory and their relations to the productions of program memory.

right-hand side The "then" part of a production representing the action produced when the condition elements of the left-hand side are met.

Soar A production system with general reasoning power and the ability to learn.

systolic array A structure of interconnected processing elements that together achieve a high computational throughput without increasing input/output bandwidth with the outside world. Within the array, data "pulses" directly from one cell to the next, without passing through memory.

tactical inference rules
: CAT system knowledge base consisting of productions concerning a tactical situation.

task
: Mach's basic resource allocation unit, comprising a paged address space and access to system resources.

thread
: Mach's basic unit of computation, executing within a task.

token
: Production system data objects that flow between Rete graph nodes. Tokens consist of a tag ("+" or "-") and a list of working memory elements that the system is trying to match or has already matched against condition elements in the production's left-hand side.

two-input node
: A node in a Rete graph that tests for joint satisfaction of condition elements in the left-hand side of a production. When a token arrives at one input of a two-input node, it is compared to each token stored in the memory node connected to the opposite input. All token pairs that have consistent variable bindings are sent to the successors of the two-input node.

virtual copy
: A copy of an address space's contents that is passed via a pointer to the location of that address space. No data is physically moved.

virtual memory
: A technique that expands the apparent amount of system memory by supplementing hardware memory with disk space for temporary storage.

W2
: A high-level language, and corresponding compiler, for the Warp systolic array machine. W2 allows a user to specify each cell's actions individually while retaining access to array-level parallelism.

WPE
: The Warp Programming Environment, which provides a uniform environment for editing, compiling, debugging, and executing W2 programs.