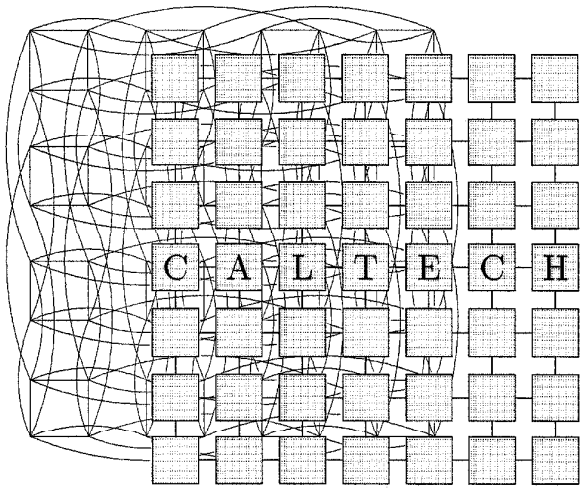

SUBMICRON SYSTEMS ARCHITECTURE PROJECT
Department of Computer Science
California Institute of Technology
Pasadena, CA 91125

Semiannual Technical Report

Caltech Computer Science Technical Report

Caltech-CS-TR-91-10

1 November 1991

The research described in this report was sponsored by the Defense Advanced Research Projects Agency, and monitored by the Office of Naval Research.

SUBMICRON SYSTEMS ARCHITECTURE

Semiannual Technical Report

Department of Computer Science
California Institute of Technology

Caltech-CS-TR-91-10

1 November 1991

Reporting Period:	1 March 1991 — 31 October 1991
Principal Investigator:	Charles L. Seitz
Faculty Investigators:	Alain J. Martin
	Charles L. Seitz
	Jan L. A. van de Snepscheut

Sponsored by the
Defense Advanced Research Projects Agency

Monitored by the
Office of Naval Research

SUBMICRON SYSTEMS ARCHITECTURE

Department of Computer Science
California Institute of Technology

1. Overview and Summary

1.1 Scope of this Report

This report is a summary of research activities and results for the eight-month period, 1 March 1991 to 31 October 1991, under the Defense Advanced Research Project Agency (DARPA) Submicron Systems Architecture Project. Previous semiannual technical reports and other technical reports covering parts of the project in detail are listed following these summaries, and can be ordered from the Caltech Computer Science Library.

1.2 Objectives

The central theme of this research is the architecture and design of VLSI systems appropriate to a microcircuit technology scaled to submicron feature sizes. Our work is focused on VLSI architecture experiments that involve the design, construction, programming, and use of experimental multicomputers (message-passing concurrent computers), and includes related efforts in concurrent computation and VLSI design.

1.3 Highlights

- Mosaic C 1.0 chips functional on first silicon, and in use in small Mosaic multicomputers (section 2.1).
- USC/ISI ATOMIC project uses Mosaic components to implement a high-speed LAN (section 2.2)
- Mosaic runtime system runs on Mosaic ensembles (section 3.1).
- A simple and efficient algorithm for task distribution (section 3.3).
- Demonstration of Page Kernel fault tolerance (section 3.5).
- Asynchronous memories and arithmetic in CMOS and GaAs (sections 4.1–4.4).
- Slack-generator chip tests new FIFO structures (section 4.5).

2. Architecture Experiments

2.1 The Mosaic Project

Charles L. Seitz, Nanette J. Boden, Jakov Seizovic, Wen-King Su

2.1.1 Summary of Results

The Mosaic C is an experimental *fine-grain multicomputer* based on the single-chip node shown in figure 1. This chip includes 32K 16-bit words of fast dynamic RAM (the lower 3/4 of the chip area) together with (left to right across the top) a processor and packet interface, clock driver, bootstrap and self-test ROM, and two-dimensional self-timed router. The 136 pins are devoted to the four bidirectional channels that connect to the north, south, east, and west neighbors; timing inputs and amplified outputs to distribute clock, reset, and refresh signals through the mesh; LED and tachometer outputs; and 36 Vdd and GND pins. The first silicon of this 9.25mm×10.00mm, million-transistor chip was fabricated by MOSIS during this reporting period, has been tested extensively, and functions correctly.

We have started the construction of a 16K-node Mosaic system in which the chips will be packaged by tape-automated-bonding (TAB) on the 8×8 boards shown in figure 2. These boards self-compose in two dimensions to form an arbitrarily extensible mesh of Mosaic nodes. The Mosaic chips and boards are being built by Hewlett-Packard Company under a subcontract; the Mosaic chip, overall packaging design, and automatic-testing software are, however, our own work. Much of our effort over this reporting period has been aimed at preparing the Mosaic for manufacture, including automatic testing of both chips and boards.

Although the completion of the first set of four 8×8 boards is not expected until February 1992, Mosaic program-development systems based on memoryless Mosaic chips have been in routine use for programming-system and application development over the last year. These program-development boards have four nodes connected as a 4×1 mesh. The external SRAM memory of each node can be read and written through a VME interface to a host workstation. Four external channel connections allow the program-development boards to double as host interfaces to Mosaic arrays. In addition, 3×3 and 4×4 test boards based on Mosaic C chips bonded in pin-grid-array (PGA) packages, but employing the same board outline as the 8×8 boards, were designed and built during this reporting period. These boards have allowed us to test the Mosaic C chips in an electrical environment similar to that of the 8×8 boards, and to provide additional hardware to support programming-system and application development.

The basic programming tool kit for the Mosaic node consists of a C compiler, library, device driver for the program-development and host-interface boards, example programs, and documentation. This tool kit, which was described in our previous report, has required few modifications during this reporting period, and

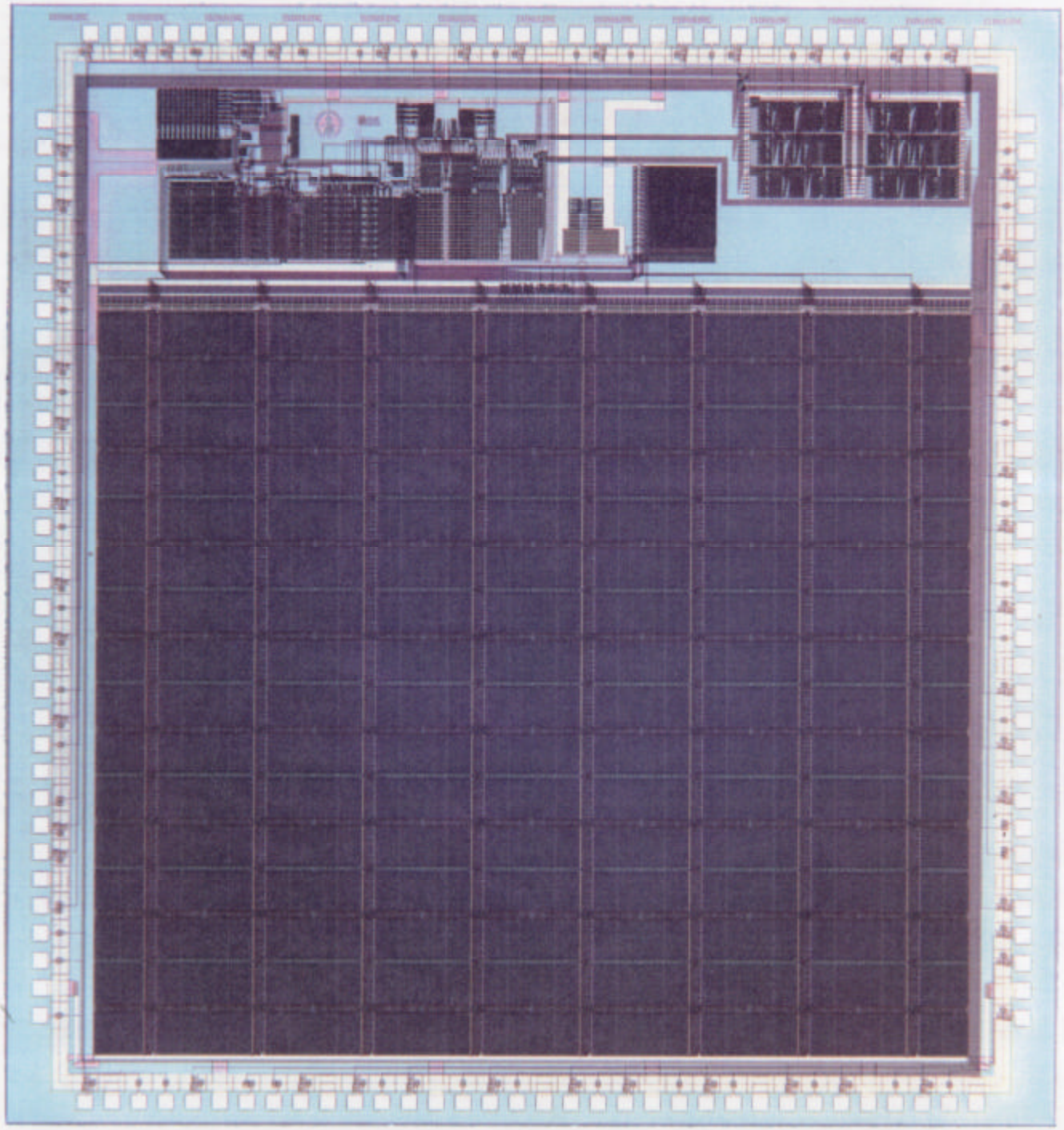

Figure 1: Photomicrograph of the Mosaic C 1.0 chip.

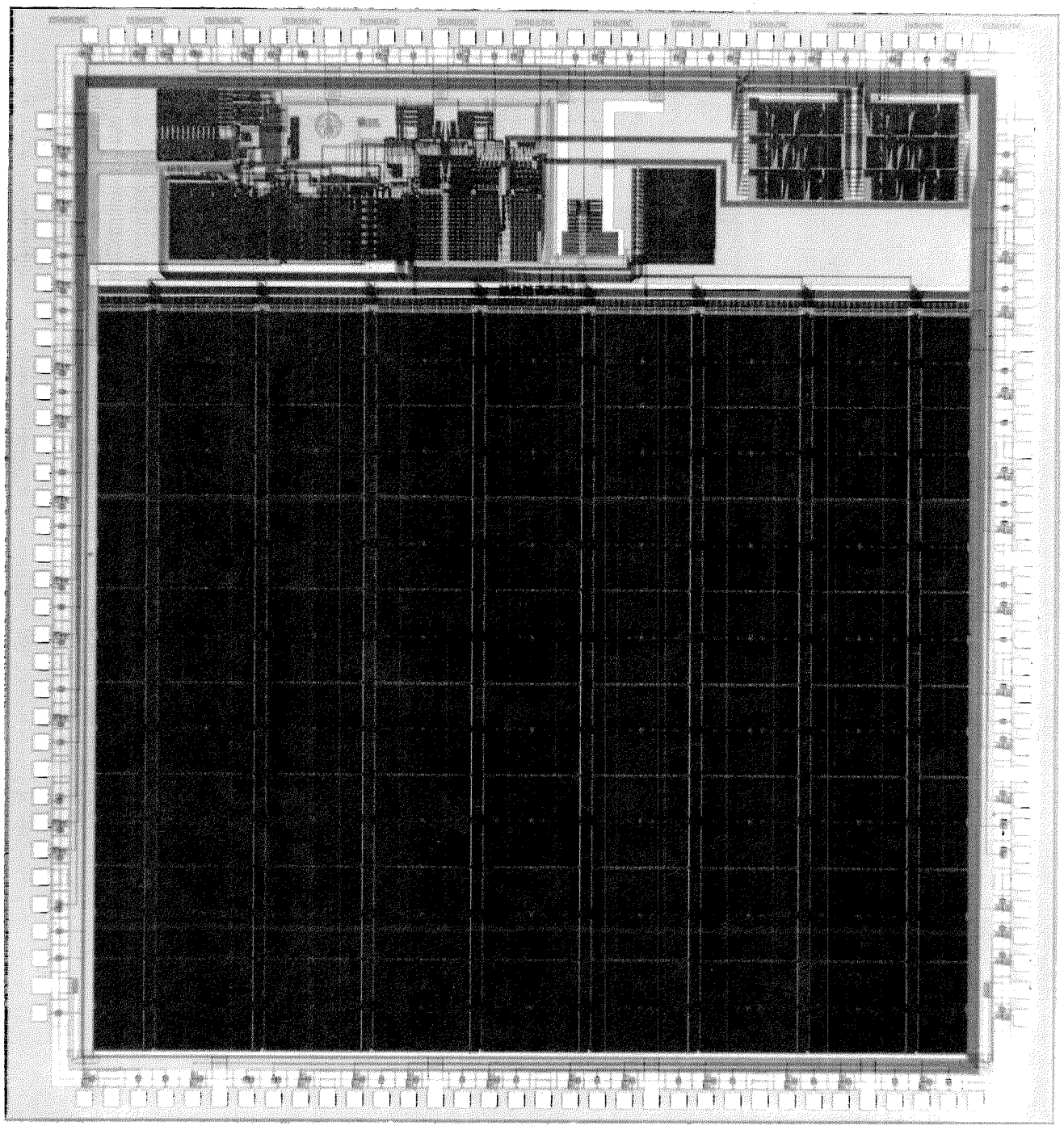

Figure 1: Photomicrograph of the Mosaic C 1.0 chip.

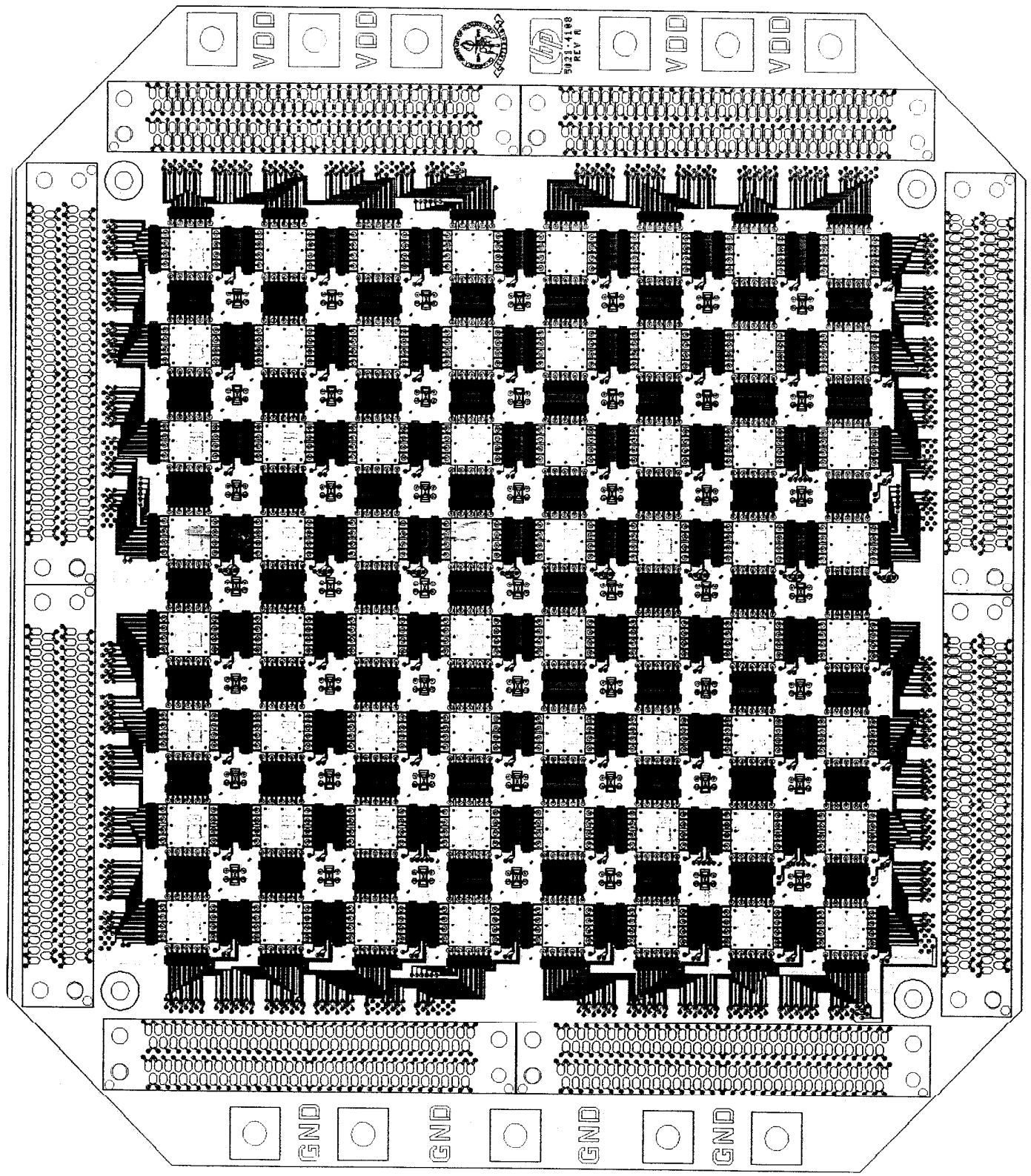

Figure 2: Plot of the Mosaic 8×8 board, nearly to scale.

has served very well for the development of programming systems and applications. Research on programming systems for the Mosaic is described in several sections of Chapter 3. A new, C++-based programming system has replaced the Cantor programming system for our internal use. This new system is still evolving, so it will not be described until our next semiannual technical report.

Program-development boards and the programming tool kit were provided to USC/ISI for ATOMIC, a project to demonstrate a high-performance local-area network based on Mosaic components. Their initial efforts, reported briefly in section 2.2, have already been highly successful.

The following subsections describe details of selected aspects of the Mosaic project.

2.1.2 The Mosaic C Chip

The Mosaic C 1.0 chip, 9.25mm×10.00mm, and with well over 1M transistors (including the 2^{19} transistors used as capacitors in the dRAM), was fabricated through MOSIS using the MOSIS λ=0.6μm SCMOS (HP CMOS34) process, and functioned correctly on first silicon at a clock rate in excess of 30MHz. This accomplishment is less remarkable than it may appear, considering that the dRAM and logic sections of this chip had previously been fabricated and tested separately.

The yield on the first batch of Mosaic C 1.0 chips, cut from two wafers, was 12/50. The yield on a second batch, cut from two additional wafers from the same run, was 24/50. MOSIS users will find it interesting that there is such a significant difference in yield between one set of two wafers and the other set. The combined yield, 36/100, is close to the 40% we predicted from separate tests of the dRAM and memoryless Mosaic chips. Of the 64 bad chips, 50 had dRAM faults, and 14 had faults in the logic section that may have been masking additional memory problems. Of the 50 chips with dRAM faults, 12 exhibited single-bit errors, 6 had two-bit errors, 3 had three-bit errors, 4 had decoder problems, and 25 had multiple problems that could not be isolated.

Tests of the Mosaic C 1.0 chips on the 3×3 boards allowed us to make a number of small changes for the chips in our pilot-production fabrication run through Hewlett-Packard. These changes included adjustments to the transistor sizes in the output drivers and input hysteresis circuits, and correcting an inversion in the amplification chain used to broadcast the reset and refresh timing signals through a Mosaic array. In addition, a missing piece of metal-2, which was redundantly connected with poly, was added for aesthetic rather than practical reasons. Fabrication of 48 wafers of these Mosaic C 1.1 chips is expected to be complete by the end of November. These chips will be tested on the wafers at Hewlett-Packard. We expect this run to yield at least 1,000 working chips, some of which will be used to build the first four 8×8 boards, and others of which will be packaged in PGAs to build additional 4×4 boards.

The extensive tests to which our entire population of functional Mosaic C 1.0 chips were subjected had the principal goal of learning how to distinguish possibly marginal chips during the wafer test. For example, the necessary refresh period for the dRAM varied between the best and worst chips by a factor of more than twenty. The small population of chips that require a short refresh period are possibly poor risks to build into the 8×8 boards; they will, accordingly, be rejected in wafer testing. Similarly, the population of working chips was characterized according to minimum operating voltage so that we would be able to perform wafer testing at an operating voltage that causes another small population of possibly marginal chips to be rejected. In addition, Hewlett-Packard will reject any wafer that exhibits an uncharacteristically poor yield. Another set of tests that stressed the message-passing communication was performed to assess the reliability of the interchip communication.

2.1.3 Memoryless Mosaic Chip

Since our previous semiannual technical report, we have gone through four more iterations of memoryless Mosaic chips. MM3.3, MM3.4, MM3.5, and MM3.6 were all completely functional chips, with minor changes aimed at improving the performance and/or safety margin. Correct operation at 38MHz was achieved on the program-development boards with 15ns external SRAMs.

Because the memoryless Mosaic chip is now so well characterized, we submitted a $\lambda = 0.5\mu$m (HP CMOS26) version at the request of MOSIS for a test run. This test chip required designing a new set of pads for the CMOS26 process. These chips are currently being fabricated.

2.1.4 Mosaic Program-Development Board R2.0

Following the success with the first batch of ten R2.0 Program-Development and Host-Interface (PD/HI) boards, described in our previous semiannual technical report, we built eight additional boards using 15ns SRAM chips instead of the 25ns chips. A 33.3MHz, 32-node Mosaic multicomputer was constructed by connecting these boards in a linear array in the card cage of a Sun4/390. Other boards were installed singly in various Sun3/120 boxes in our department. These boards have been used for programming-system development as well as for class exercises in a third-term concurrent-programming class.

During the summer, the PD/HI boards were removed from the Sun-3/120s during a department-wide upgrade from Sun3s to Sun4s. Some of the sparc boards were loaned to USC/ISI for their experiments with the ATOMIC network, a high-performance LAN built with Mosaic components (see section 2.2). One board was configured to serve as the host interface for a 27-node array of Mosaic C elements (see figure 3). Two additional boards have been earmarked for use in a pair of test fixtures to be built and used by HP for testing 8×8 boards.

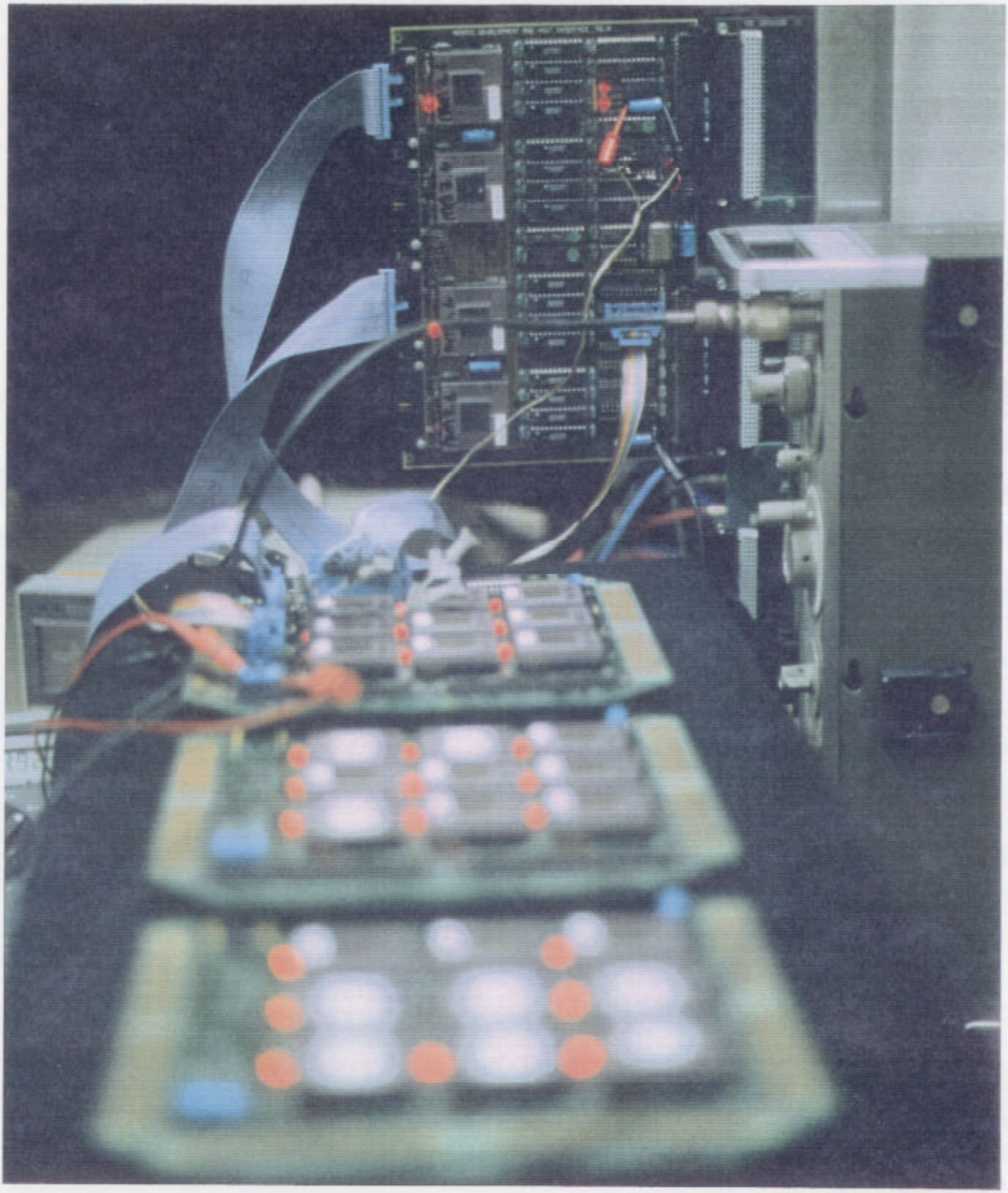

Figure 3: PD/HI board connected to a 27-node Mosaic C array.

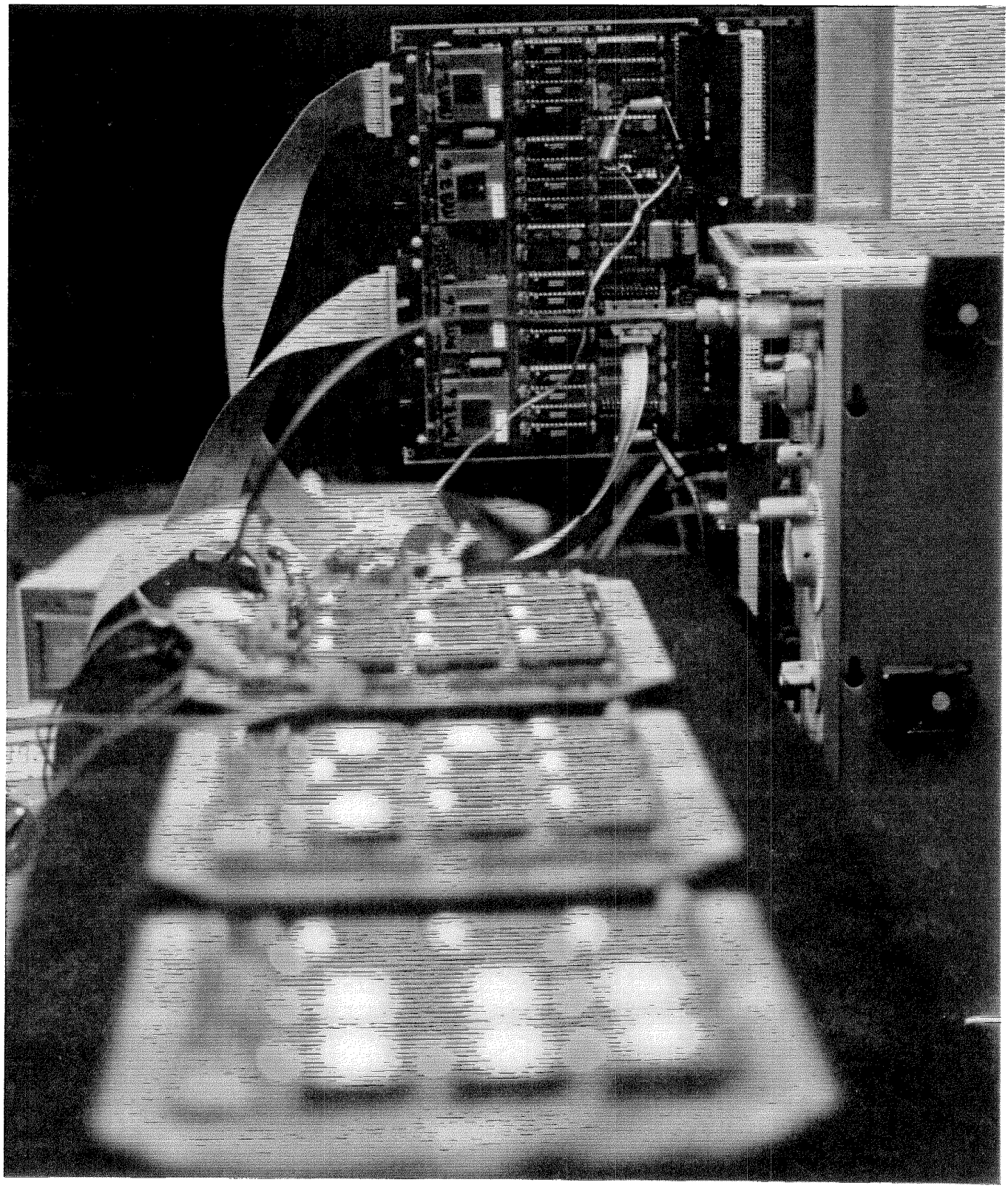

Figure 3: PD/HI board connected to a 27-node Mosaic C array.

2.1.5 Mosaic 3×3 and 4×4 Boards

In order to test the Mosaic C elements in a 2-D array configuration, and to test the mechanical and electrical characteristics of the checkerboard composition of 8×8 boards, we designed a PC board to hold a 3×3 array of Mosaic C chips bonded in pin-grid-array (PGA) packages. This 3×3 test board (figure 4) has the same dimensions and stacking-connector placements as does the 8×8 board, and is the smallest array that still allows at least one element to be connected to four neighboring elements of the same board. The design left enough room on the board for additional connectors and circuits for performing various tests and measurements on the Mosaic C 1.0 chips. Ten of these boards were built, but until the run of Mosaic C 1.1 chips is completed, we have only enough functional Mosaic C chips to populate three boards. These three boards are connected to produce a 9×3, 27-node Mosaic multicomputer. Although most of the chips are capable of higher speed, some marginal chips that we must include to complete a set of 27 forced us to run the array at a clock speed of 25MHz.

In anticipation of the demand for additional Mosaic C development platforms and the expected shipment of a large lot of Mosaic C 1.1 chips from HP, we also designed and built a similar PC board that holds a 4×4 array of Mosaic C chips. This 4×4 array of 132-pin PGA-bonded chips is the largest that will fit on a board with the same dimensions and stacking-connector placements as the 8×8 board. The density of this board with PGA packages is one quarter that of the board with TAB bonding, which illustrates well one of the advantages of using TAB; another advantage is cost. The spacing between the PGA chips on the 4×4 board is so tight that decoupling capacitors are placed under the PGA sockets, and the LEDs are viewed from the back side of the board through holes drilled through the sockets.

2.1.6 Automatic Testing

Because replacing defective TAB-bonded chips on the 8×8 board is relatively difficult, and because there are 64 chips on the board, it is extremely important to screen out as many of the defective chips as possible before they are bonded to the board. (If 1% of the chips that pass the chip test turn out to be defective, then 47% ($1 - 0.99^{64}$) of the boards will be have to be reworked.)

The Mosaic was designed to test itself at every key point in the manufacturing process. The chip contains a self-test program in ROM, with which the processor, ROM, dRAM, and most of the router are checked upon reset. When a wafer leaves the fab line, chips are checked individually by a Sentry-15 tester. The test programs and test vectors were written according to Hewlett-Packard's specifications, and sent to HP via e-mail. The tester will reset the chip, allow it to complete its self-test sequence, check the remaining portions of the router, and send a message to the chip to fetch the self-test result. The same test will be applied when the wafer is diced and "good" chips are inner-lead bonded into the TAB frame.

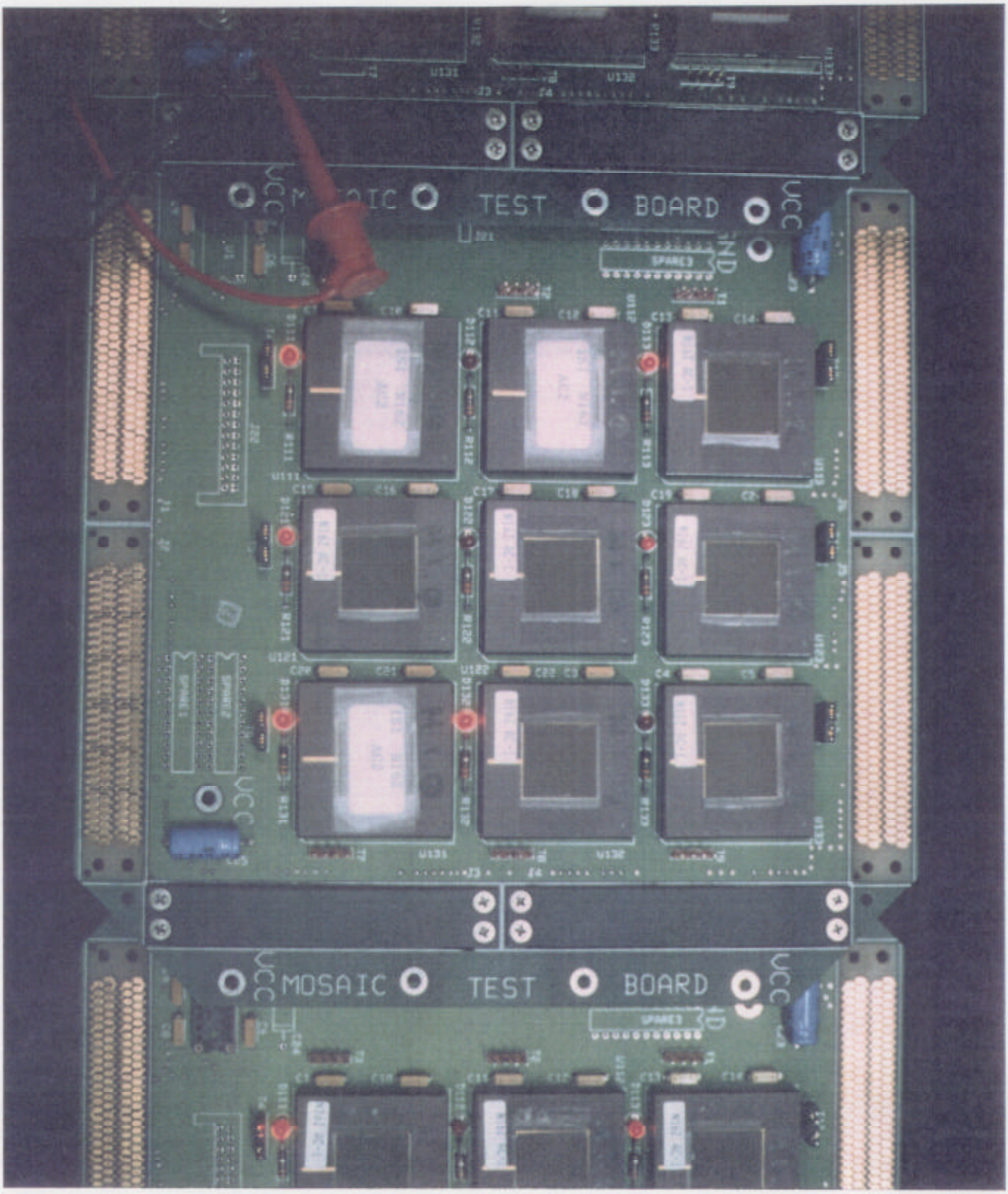

Figure 4: 3×3 test board inside a Mosaic C array.

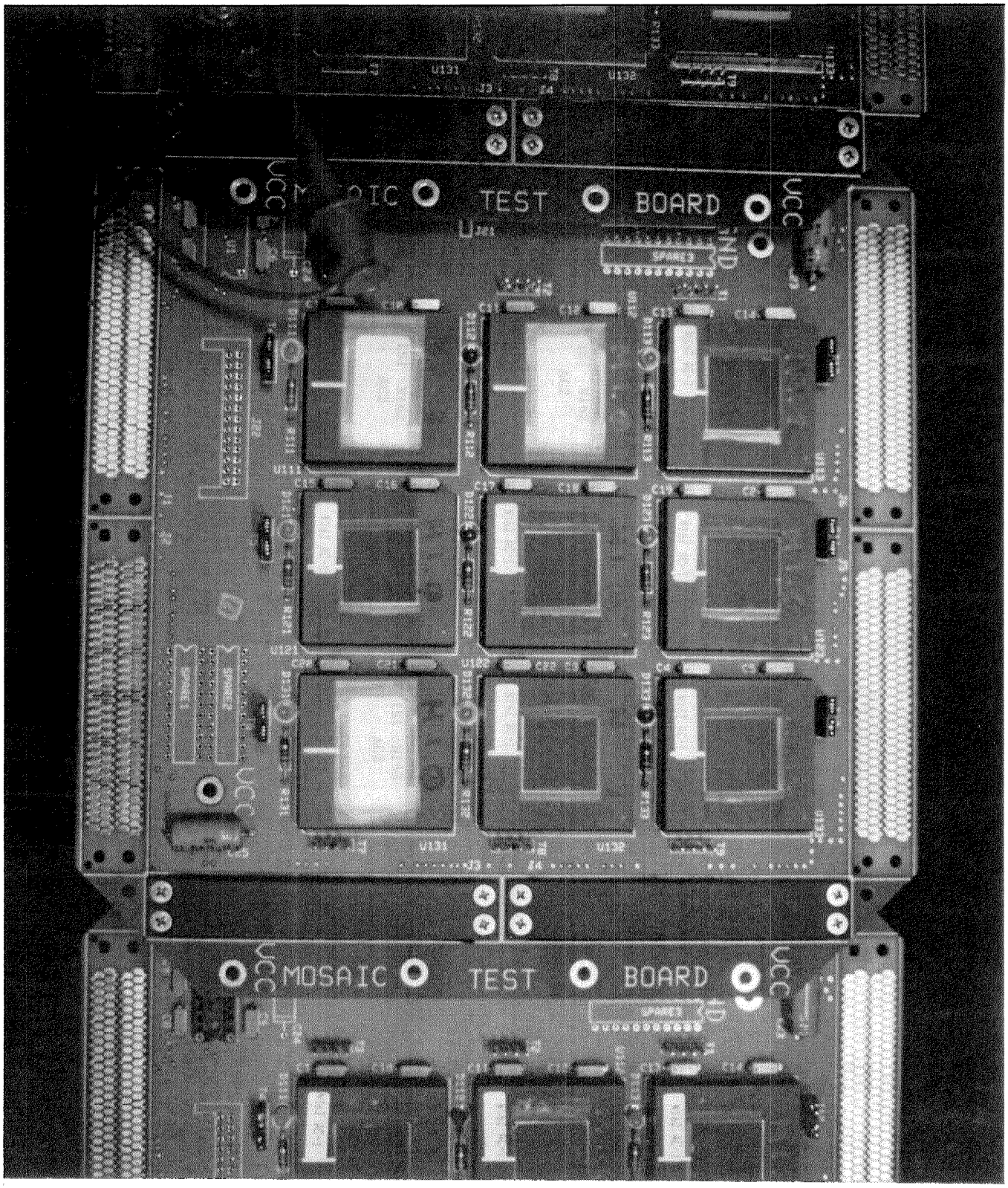

Figure 4: 3×3 test board inside a Mosaic C array.

After the chips are outer-lead bonded to the 8×8 boards, the boards are tested for defects. A custom-built test fixture allows a Sun workstation to test the node at the corner of the board through a PD/HI board. Testing then progresses away from the corner with the help of those nodes that have already passed their tests.

2.2 The ATOMIC Project*

Danny Cohen, Charles L. Seitz, Wen-King Su

ATOMIC is a very-high-speed local-area network (LAN) built by USC/ISI using Caltech Mosaic components and programming tools. Each of its switches is capable of routing IP packets at very high rates, while providing added value through computing and buffering. ATOMIC already supports the IP protocol and all the communication protocols above it, such as TELNET, FTP, and e-mail.

The initial proof-of-concept installation at ISI uses VME-based Sun workstations as hosts for Mosaic PD/HI boards, whose external channels are connected through standard Mosaic-channel cables. This current prototype ATOMIC uses 25MHz PD/HI boards, which limit the memory bandwidth and channel rate to 50MB/s (400Mb/s), which is about 60% of the capability expected from mature Mosaic hardware. In addition, the Suns cannot provide the ATOMIC network with data fast enough to stress its performance.

To measure the performance we use traffic generators and move packets from one Sun/ATOMIC shared memory to another Sun/ATOMIC shared memory. Using these memories we run 1,500B packets (typical FTP packets) between two Suns at rates over 370Mb/s (longer packets approach 400Mb/s, which is the limit of the 25MHz PD/HI boards). 54B packets (such as ATM) are transferred by ATOMIC at rates above 125Mb/s. The shortest packets run at rates exceeding 454K-packets/s. The above figures indicate half-duplex performance; full-duplex performance is double the numbers stated, since the input and the output channels are totally independent of each other. All these performance figures will scale up with faster Mosaic components.

Unlike most LANs, the ATOMIC performance is per link. The aggregate performance of the entire network is limited only by its configuration, since ATOMIC traffic flows do not interfere with each other unless they share links. This characteristic is unlike Ethernet and FDDI, in which all the traffic flows interfere with each other, and share the same total bandwidth (of 10Mb/s and 100Mb/s, respectively).

---

* Danny Cohen is with USC/ISI. This effort, which is supported under a separate DARPA contract at USC/ISI, is reported here as an example of an interesting application of the Mosaic, and of interaction between DARPA contractors. Additional information about ATOMIC should be obtained by contacting Danny Cohen at USC/ISI.

The ATOMIC network was running at ISI just a few days after the arrival of the hardware. There have been no errors observed so far in test runs that have generated and received more than 10^{12} bits. Most of the initial effort in developing this demonstration was devoted to interfacing the ATOMIC driver to the UNIX kernel and to its IP driver, such that the use of ATOMIC would be totally transparent to the higher-level protocols.

The ATOMIC network is an addressless network that requires source routing for the packets to navigate through it. The discovery, establishment, and control of these routes is a task that will be based on the capabilities of the switching elements in the network, the Mosaic chips.

3. Concurrent Computation

3.1 Fine-Grain-Multicomputer Runtime Systems

Nanette J. Boden, Charles L. Seitz

As discussed in previous reports, the design philosophy of the Mosaic runtime system is to provide efficient and robust *distributed* runtime support for user programs. The primary goal of the runtime system is to balance the efficiency considerations of process management and message passing while maintaining sufficient data structures to support resource distribution. During the period covered by this report, the design of the Mosaic runtime system was completed, and a very cleanly structured prototype runtime system was implemented. The prototype runtime system is written in C, and runs on all existing Mosaic ensembles, but not yet with all of the planned features. While the prototype runtime system is an important milestone in the development of the Mosaic multicomputer, the experiments that we must perform to examine the efficiency and robustness of its algorithms rank at least equal in significance. The following paragraphs indicate the nature and objectives of these experiments.

Since executing a user program as quickly as possible is the ultimate goal of the runtime system, overhead for message passing and user-process management must be minimized. Previous work from our group has relied on compiler techniques (as employed in the Cantor programming system) and streamlined runtime support (such as employed in the Reactive Kernel node operating system) to minimize the time required for these basic operations. The Mosaic runtime system borrows heavily from the essential aspects of this previous work. In addition, the Mosaic runtime system all but eliminates copying message or process data within a node. Such copying is expensive in time and memory bandwidth, and is generally unnecessary. The notion of a "copyless" runtime system has been an important guiding force in the runtime system design and implementation.

In addition to basic multicomputer operations, the Mosaic runtime system must handle automatic resource management for the user program. Explicit naming of nodes for placing processes and for message passing between processes, the usual practice for medium-grain multicomputer programs, is impractical for multicomputers containing many thousands of nodes. Beginning with the Cantor programming system, we have removed explicit naming of nodes — in fact, any explicit reference to machine resources — from the source programming notations. This approach provides complete machine independence for user programs, and great flexibility for the runtime system in managing resources. In principle, the only case in which a program would not be able to execute to completion is if the memory demand exceeded the available (possibly virtual) memory of the entire machine. However, the performance of the runtime algorithms for placing and managing the name-space of processes is crucial to the overall performance of the execution of the user program.

Since the aggregate memory of a fine-grain multicomputer is partitioned into many small pieces, the fine-grain runtime system must actively cooperate with other nodes to accomplish many important tasks. Unless measures are taken to balance local resource demands, the memory of a single node may be exhausted while much of the total memory of the ensemble remains unused. The low message latency between Mosaic nodes ($\approx 3\mu s$) permits aggressive approaches to remote memory allocation, storage, and retrieval, so that memory demands can be more evenly distributed throughout the entire machine. The Mosaic runtime system attempts to blur the physical memory boundaries of the nodes in the ensemble. Although the aggregate memory of the Mosaic is partitioned into more and smaller pieces than the memory of a medium-grain multicomputer, the low cost of communication with other nodes enables the runtime system to treat the ensemble as a machine with a much less "distributed" memory.

The low communication costs to other nodes also permit aggressive solutions to another challenge presented by the limited size of Mosaic local memory — the location and access of user process code. Maintaining one copy of the code for each process type on every node is clearly wasteful, and will be physically impossible for large programs; the code for the user program must, accordingly, be distributed throughout the nodes of the machine. However, in order for a user process to execute, the code and the data must reside in the same node. Rather than send the code to the data, the Mosaic runtime system generally ships the data to the code. This approach is motivated by the observations that the memory of the Mosaic is remotely addressable via the runtime system, and that the size of the data for fine-grain processes is generally less than the the size of the code to be executed.

Software Tools. As part of the development of the prototype Mosaic runtime system, various software tools have been developed for debugging and program profiling.

A simple symbolic debugger is available for use on Mosaic Development Board systems. A control program running on the Sun host accepts debugging commands (*eg,* `stop`, `print`, `step`) from the console. The memory of an individual Mosaic node is directly addressable by the Sun host so that memory locations can be accessed to print values. Since the code being executed by the node is also directly accessible to the host, breakpoints can be inserted by replacing instructions at the desired breakpoint with instructions to jump to a predefined location, where instructions dictate that register values be dumped, and to leave the Mosaic spinning in an infinite loop. To continue processing, the host program writes over the destination of the spin jump with the previous instruction pointer so that processing resumes at the breakpoint.

Program profiling tools currently focus on collecting and displaying information in counters that are inserted into the runtime system and user programs. Counters (a type of "post-mortem" analysis) are very useful for profiling without significantly influencing the execution of the program.

In addition, host-interface routines were written so that the host Sun can pass messages to the Mosaic ensemble. A Mosaic PD/HI Board is used as the hardware interface between the Sun and the Mosaic ensemble. One node on the PD/HI board is loaded with code that accepts messages from the message network and queues them internally; another node is loaded with code that reads from an internal queue and sends messages into the Mosaic ensemble. A program running on the Sun workstation can send and receive messages by executing functions analogous to the Cosmic-C primitives `xsend` and `xrecv`. These functions read and write the internal queues of the nodes on the PD/HI board via the shared memory between the Sun and the resident Mosaic nodes.

3.2 Systematic Design of Concurrent Programs

H. Peter Hofstee, Jan L.A. van de Snepscheut

One of our ongoing efforts is to develop concurrent programs in a systematic way. We have continued our experiments with stepwise refinement, a technique that is well understood for the design of sequential programs. In the case of concurrent programs, we start with a simple program that solves the problem, except that it is a sequential program instead of a concurrent one. In a number of refinement steps, this program is transformed into a concurrent version.

First, the variables are partitioned over processes. Second, the coupling between the processes is made less tight by postulating copies of variables that are not necessarily up to date. Next, the copies are made via message passing. Finally, execution of the individual actions is controlled in such a way that deadlock is excluded, and progress is guaranteed. Even though we presently understand this method only informally, and cannot yet state the transformation rules with sufficient precision, we have succeeded in applying this method to a number of problems and have obtained efficient programs. The complexity of the programs, especially their flow of control, is such that we would never have been able to construct them without employing systematic methods.

One of the examples is a distributed sorting algorithm. The program was tested both on the Ametek S2010 and on Mosaics. Even though the problem specifies a linear ordering of the values to be sorted, the program takes advantage of the higher connectivity of the networks in the two machines.

3.3 A Distributed Task Pool

H. Peter Hofstee, Johan J. Lukkien, Jan L.A. van de Snepscheut

A method that is often used for constructing a concurrent program is called a processor farm (or manager–worker, or dealer–player formulation). The program consists of one scheduler plus a number of work processors. The scheduler farms out the work to the individual processors. It leads to simple and elegant programs but,

unfortunately, it does not scale well to large systems because the scheduler becomes a bottleneck.

We have developed a distributed version of the processor farm that eliminates the bottleneck. The algorithm works roughly as follows: Every processor has a number of tasks in its local task pool. Whenever a processor becomes idle, it executes one of the tasks from its local pool, and whenever it generates new tasks, they are added to the local pool. This scheme tends to favor locality, thereby avoiding unnecessary communications. However, the pools need to be adjusted such that no processor is idle when there is a task available in some other pool. On the other hand, one wants to avoid a lot of message passing just to deal with a minor imbalance.

To that end, we choose a threshold, T. Typically, T is a small number, such as 2 or 3. The algorithm sees to it that either all pool sizes are at least T, or that all pool sizes are at most T. In either case, the load is well distributed over the processors. The algorithm is such that the above condition is enforced via local communications only, and by recording information about neighboring processors only. This is essential to make it scale well to large systems. The resulting algorithm is very efficient, and requires very few communications, and then only local ones. It compares favorably to random placement, which requires more communications, and those communications are typically non-local.

3.4 An Interpreter for a Functional Programming Language

John Brodoff, Jan L.A. van de Snepscheut

We have written a number of widely different implementations of a functional programming language. Since the functional language has no notion of processes or of storage allocation, they serve as a test vehicle for seeing how well we can provide those automatically in an implementation. One of the problems faced in the implementation is the recycling of parts of the store that are no longer in use (garbage collection). Many garbage-collection algorithms exist, and one of them is reference counting. A major problem with reference counting is that the scheme requires that increment and decrement messages arrive in the order in which they were sent, even if they were originated by different processors. This is a very strict requirement, and we have devised and implemented an alternative scheme in which the requirement is relaxed so that only the order of messages between any pair of processors need be maintained.

3.5 The Page Kernel

Craig S. Steele, Charles L. Seitz

The previously-described Page Kernel (PK) concurrent-programming environment is now substantially complete as a research project. This report will recapitulate some of the basic features of the system, along with some recent refinements that provide a mechanism for fault tolerance and load balance.

PK is an evolution of the reactive-programming model for medium-grain programming on second-generation multicomputers. Virtual-memory hardware is used to support a distributed, shared-memory programming system based on low-context processes called *actions*. The programmer accesses shared data structures, called *blocks*, much as in a shared-memory machine, but without the need for explicit locking for concurrency control. The execution cycle of actions implicitly induces atomicity and consistency of effect in modifying computational state. Mutually-exclusive updating of all modified data is implemented by a transaction-style rule that guarantees that changes are effected either completely or not at all. The rule is applied to the set of data blocks actually accessed during execution, the *write set*. Data in the write set are known to be the current versions. Computations that are tolerant of data staleness may improve concurrency by relaxed coherency requirements for read data; the triggering mechanism for scheduling makes this class of problems surprisingly large.

Scheduling of actions for execution is distributed and nondeterministic. Actions may set *triggers* on data blocks. When such a data block is modified, all actions with set triggers will be scheduled for subsequent execution. This externalization of the computation's control flow allows programs to be written as small code fragments that establish particular relations between input and output data objects. At the termination of the computation, the desired relations are established on the final results.

The decision to make data-block sharing and mutually-exclusive writing part of the programming model has simplified many lower-level tasks, and generated some interesting properties. The core functions of the kernel create, update, and cache data blocks. These operations provide the services required by the user-level programming model, but surprisingly few extensions are needed to support kernel-level operations, such as the construction of action contexts and performance monitoring. Action code is loaded into a data block created by the kernel, but is otherwise identical to user-instantiated blocks. Features such as demand-fetching of the code to a multicomputer node and code-sharing among the actions on a node require no special mechanisms. The primary difference between user and kernel use of the basic data-block functions is that the kernel can directly probe the status of data blocks and may block pending satisfaction of a remote service request. The explicit locking in memory and the privileged-access-level bypass of MMU page protection are used to reduce overhead for such features as action dispatch and the maintenance of kernel performance statistics, but the standard data block services are used for creation and communication of the supporting data structures.

PK actions have minimal *essential* (semantically significant) context; data blocks contain virtually all of the computational state that persists between action activations. An instance of an action is specified by little more than references to its argument-list binding, code, and trigger set. Action relocation is a minor variation of action creation. A short message containing a couple of references sent

from one node to another is sufficient to allow an action to be loaded. Of course, large amounts of *transient* context may be required during actual execution of an action. Data blocks are mapped, fetched, and cached as needed while the action is active. After the action completes successfully, any modifications made to the node data-block cache are written back to master copies. Upon action completion, the transient context, *ie,* stack content and block mappings, are redundant. Upon success (or failure) of the write-back update, the node data-block cache contents are redundant. The node data-block cache retention policy will affect performance but not computational correctness.

The externalization of scheduling and computational state, and the well-defined state-update process by successful actions, provide the possibility for fault tolerance. The optimistic consistency mechanism used by PK was chosen to preclude deadlock (at the cost of possible livelock), but an interesting additional benefit is that the node cache contents are dispensable except for the brief interval at the end of the update protocol after action completion. If we consider independent single-node fault-stop as the most probable failure mode, relatively straightforward extensions of PK can provide transparent fault tolerance. The state of a PK computation is defined by the contents of the data-block master copies and by the essential context of the extant actions. Duplication of the master copies via node-pairing with standard logging protocols can protect against data loss from single-node failure. The tasks of embedding failure-detection timeouts in all internode protocols, and providing for a switch-over to the backup master copy, are tedious but practical.

Replication of actions is more interesting. Instead of recording the binding for each instantiated action somewhere and then initiating a recovery procedure upon detection of a node failure, we can simply replicate the action. The PK consistency rules provide the necessary mutual exclusion to avoid conflict if replicated actions should execute concurrently. The semantics of the program are the same regardless of the number of replicated actions. (A simple scheduling-priority system was added to improve efficiency in lightly-loaded machines, where replicated actions might unduly compete for write access to the same data.) The technique has been successfully demonstrated, following a cleanup of the kernel's action-scheduling code to regularize the treatment of trigger-set and action-descriptor data blocks. Redundant copies of actions are scheduled and executed without any requirement to detect node failure; the scheme is essentially transparent to the kernel as well as the user. The freedom to replicate PK actions can also be used to provide a form of computational load balance that is transparent to the kernel, but more-sophisticated schemes may directly control action placement.

The current implementation split naturally into two distinct functions: master-copy data-block maintenance and user action execution. Scheduling and data-block maintenance operations do not require any interpretation of data-block content, and could be executed on hardware distinct from the computational engine. High-performance communications network hardware diminishes locality considerations;

on the Ametek S2010, PK runs well both with uniformly distributed master-copy services or with dedicated master-copy server nodes. Ideally, master-copy server nodes would be specialized for high reliability and low communications latency.

The set of design decisions for PK hang together well. The computational model is appropriate for multicomputers with capable communications systems, and is reasonably efficient and scalable. The implicit consistency mechanism seems natural for programming. The externalization of computational state and scheduling provide additional flexibility in load balancing and fault tolerance.

3.6 Multicomputer C

Marcel van der Goot, Alain J. Martin

Multicomputer C, "mcc," is a programming language for message-passing multicomputers. As reported earlier, we wrote a compiler that compiled an mcc program into an ANSI C program that can be executed on a Sun workstation. Early this summer, we finished an updated version of the compiler that now generates code for multicomputers as well.

Currently, the compiler generates code for the Ametek S2010 multicomputer, for networks of Sun3 or Sun4 workstations (a so-called "ghost cube" provided by the Cosmic Environment), as well as for single workstations. In the case of the Ametek S2010, it supports both Sun3 and Sun4 hosts. The compiler has been written to hide differences in machine formats from the programmer. For example, although structures are stored differently on an S2010 node and on a Sun4 host, this is invisible to the programmer when structures are communicated between host and nodes. There are also provisions, through compiler directives, for functions that can only be executed on certain nodes, or for which the program depends on the node type.

Effort has been made to write the compiler in such a way that it can easily be retargeted for different machines. An mcc program is compiled into a mostly machine-independent C program, which is then linked with a small runtime system. The runtime system contains the actual message-passing code, and therefore almost all the machine dependencies. Adapting the runtime system to a new machine consists of writing a number of macros, a start-up function, and, if applicable, functions that convert between data formats. To target the compiler for a new machine, a new runtime system must be generated, and a small data file describing mostly sizes of data types must be written. The user interface allows the user to select the target through command-line options. This system works well enough that a single compiler is used to generate code for all the abovementioned machines. For the S2010 and for the ghost cubes, the translation is to "Cosmic C," which uses the message passing and data conversion functions provided by the Cosmic Environment and Reactive Kernel.

4. VLSI Design

4.1 Asynchronous Static Memories

Peter Hofstee, Alain J. Martin

An extensive study of asynchronous memory protocols using formal methods has led to a series of designs of asynchronous static memories and caches. Three of these designs have been fabricated through MOSIS as 2μm tiny chips; a fourth is presently in fabrication. The series of designs (and the circuits) show a dramatic increase in speed, and decrease in area per bit as well as transistors per cell. A larger memory, intended for use with the asynchronous microprocessor, is under construction. All designs were functional on first silicon.

chip	size	read cycle	write cycle	cell size
first	16×9bit	200ns	250ns	$9000\lambda^2$
second	16×16bit	40ns	50ns	$2610\lambda^2$
third	256×1bit	45ns	55ns	$2700\lambda^2$

The first chip was a cache.

The speed for 1.2μm CMOS should increase by a factor of two (ideally a little more, but pads do not scale).

All speeds are at room temperature at 5V. The first two chips are operational between 1.5 and 9 V; the third chip is functional in a limited range.

A third-generation basic cell is $2300\lambda^2$.

4.2 Asynchronous Serial-Parallel Multiplier

Christian Nielsen, Alain Martin

We have completed the design of an asynchronous multiply–accumulate unit. The design is a serial–parallel architecture. It differs from previously designed multipliers in that it uses carry–save arithmetic and the multiplication time is dependent of the number of significant bits in the multiplicand. The carry–save arithmetic ensures that the accumulation of each bit multiplication is performed in constant time, independently of the length of a carry–chain.

A 4×4bit unit was fabricated in 2μm CMOS, but due to a transistor-sizing error, the chip was malfunctioning. A revised design has been submitted for fabrication, and the design tool will be updated to disallow similar errors. The expected performance of the unit is 39Mbit/s in 2μm CMOS, giving multiplication–accumulate times between 37ns and 448ns for 16×16bit multiplications.

4.3 Asynchronous Circuits in Gallium Arsenide

José Tierno, Dražen Borković, Tony Lee, Alain J. Martin

We have adapted all our design tools (including COSMOS) for GaAs, and have switched to Hspice for circuit simulation. Confident that we could now avoid the clerical errors that we were previously unable to catch due to the lack of simulation tools, we embarked on the design of a GaAs version of the Caltech Asynchronous Microprocessor. In the middle of the transistor-sizing phase, we realized that the logic family we had chosen was too sensitive to transistor sizes for a design of this size to ever work. We modified the logic and redesigned all cells in the new logic.

Two GaAs circuits were designed and fabricated on the same Vitesse run. The first one, a 4-bit, 16-register file, was successfully tested. Access time is about 4ns, including 1.5–2ns pad delays. The measured speed is higher than predicted by Hspice, using the parameters supplied by MOSIS. Power consumption was as calculated.

The second circuit was the microprocessor. Unfortunately, a last-minute change in a pad layout to adjust the pitch for the chosen die size had not been checked by simulation. As a result, a missing connection on the output pads makes it impossible to verify their functionality, except for power consumption, which was close to the estimate (4W). We are currently working on verifying and speeding up the circuit as much as possible at the device level. The current speed estimate (by extensive Hspice simulation) for this design is about 100 MIPS, which is quite encouraging for a first design. The revised circuit will be sent for fabrication on the next MOSIS run.

4.4 Asynchronous Floating-Point Arithmetic

Tony Lee, Alain J. Martin

We are investigating the design of efficient asynchronous circuits for floating-point arithmetic. At the moment, we are looking at floating-point addition (FADD).

If we ignore operations on the exponents, a typical synchronous implementation would allot enough time to perform two additions per FADD operation. The first addition adds the significands of the two operands. Then, depending on the result, a second add is performed to round and/or negate the result if necessary. Empirical statistics gathered on actual executions of programs indicates that the second addition is needed in only about one-third of the cases. In other words, the average number of additions per FADD operation is about 1.33; but, for the sake of timing uniformity, synchronous circuits assume the worst-case situation and allot time for two additions per operation.

Asynchronous designs, on the other hand, are not restrained by such a pessimistic assumption. Operations on different operands can take different amounts of time. We have developed a FADD algorithm that yields an average of 1.12 additions per

operation. We are currently implementing this algorithm, as an asynchronous circuit. This implementation should provide some interesting performance figures.

Also, we have gathered some empirical statistics that may aid in the design of an asynchronous floating-point multiplication unit. First, we assume that the multiplication unit contains a single adder and realizes the simple shift-and-add algorithm. We then compare the number of actual additions it would perform for different recoding of the multiplier operand. Though our work in this area is still in the early stage, the result of these comparisons already gives us some indication as to which multiplication algorithms are suitable for asynchronous implementation.

4.5 Elko Slack-Generator Chip

Wen-King Su, Charles L. Seitz

The original "Frontier" mesh-routing chip (FMRC) and its derivatives, used in the Mosaic and in a number of other DARPA-supported projects, has, for the most part, been fast, reliable, and robust. However, due to Mosaic's ability to generate messages at very high rates, we are able to induce very-low-rate errors in the router under certain conditions in every batch of Mosaic chips. We suspect but are not certain that these errors are due to the risky timing schemes used in the internal routing automata of the Frontier router. To make the "Elko" router more robust, we decided to make a robustness-*vs*-area tradeoff by employing two-cycle signaling between the internal routing automata. This scheme improves timing margins of the circuits without compromising throughput. It also increases the area and reduces the power consumption.

The fundamental Elko routing automaton is a 2-cycle self-timed FIFO. We designed and fabricated a chip containing two pairs of Elko FIFOs, both to test the FIFO design and to serve as a *Slack-Generator Chip.*

The standard Mosaic channel employs a non-interference protocol that is unsuitable for sending data through long cables because this protocol has a slack of one. When the sender emits a flow-control unit (flit) of data, it must wait for the corresponding acknowledgement for that flit from the receiver before sending another flit. When used on a long cable, the extra time for the signal to travel from one end of the cable to the other and back is added to the cycle time of the data transfer. However, if a known amount of buffering is set aside on the receiving side, and if the cable and receiver can accommodate the flit rate of the sender, the sender can inject multiple flits into the cable *up to the slack limit* without waiting for an acknowledgement for each flit. At the same time a flit is emitted, (a token for) that flit is placed into a FIFO on the sending side in order to count the number of flits in transit. When the flit arrives at the receiving side, it is placed into the buffer. When a flit is removed from the buffer on the receiving side, an acknowledgement is sent to the sender. When an acknowledgement is received at the sending side, that flit

is removed from the FIFO. If the two buffers are of the same size, we can guarantee that the buffer on the receiving side will never overflow.

In the slack-generator chip, one of each pair of FIFOs serves as the receiving buffer, and the other is used to count the flits on the sending side. The chip is packaged in an 84-pin PGA, and is 180° rotationally symmetric. The first batch of slack-generator chips was fully functional except for one layout error that allows a "glitch" to appear on the output channel. A correct layout of the same function elsewhere in the chip shows no glitch. The problem was fixed, and a second version of the chip was submitted to MOSIS. The first version of the FIFO shows a sustained throughput of 60MB/s.