SUBMICRON SYSTEMS ARCHITECTURE

Semiannual Technical Report

Department of Computer Science
California Institute of Technology

5220:TR:86

March 1986

Reporting Period:	16 September 1985 to 15 March 1986
Principal Investigator:	Charles L Seitz
Faculty Investigators:	James T Kajiya
	Alain J Martin
	Robert J McEliece
	Martin Rem
	Charles L Seitz

Sponsored by the
Defense Advanced Research Projects Agency
ARPA Order Number 3771

Monitored by the
Office of Naval Research
Contract Number N00014-79-0597

SUBMICRON SYSTEMS ARCHITECTURE

Department of Computer Science
California Institute of Technology

1. Overview and Summary

1.1 Scope of this Report

This document is a summary of the research activities and results for the six month period 16 September 1985 to 15 March 1986 under the Defense Advanced Research Project Agency (DARPA) Submicron Systems Architecture Project. 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, and includes related efforts in concurrent computation and VLSI design.

Additional background information can be found in previous semiannual technical reports [5052:TR:82], [5078:TR:83], [5103:TR:83], [5122:TR:84], [5160:TR:84], [5178:TR:85], [5202:TR:85].

1.3 Highlights

Some highlights of the previous 6 months are:

- Working Mosaic A elements (2.2 and 4.1)!!!
- Torus Routing Chips work on first silicon (4.2).
- Mosaic C – an SCMOS Mosaic element – being designed (2.2.2 and 4.3).
- Compilation method for designing self-timed circuits (4.5).
- 128-node Intel iPSC in routine operation and available for use via the ARPAnet (2.1.3).
- New results in parallel execution of logic programs (3.1).
- A new flow framework for concurrent programming (3.3)
- Event-driven simulations chew up endless cycles on the cosmic cubes (3.5).
- A surprise about network topologies (3.6).

2. Architectural Experiments

2.1 Cosmic Cube Project

W C Athas, Michael Lichter, Wen-King Su, Chuck Seitz

The cosmic cubes are working reliably, and researchers are using them for application programming projects and for event-driven simulations of other message-passing architectures. Usage has been moderately heavy, and there have been very few problems reported.

2.1.1 Hardware Status

The 6-cube interface was moved from `sol.caltech.edu`, our main SUN file-server, a SUN-2/170, to `ceres`, a SUN-2/120, in order to lighten the load on the already overloaded file server. The 3-cube interface was moved from `ceres` to `venus`, another SUN-2/120.

Neither the 6-cube nor the 3-cube has failed in this 6-month period. The cubes have now logged 1.85 million node-hours with 3 hard failures. The calculated MTBF of the nodes of 100,000 hours reported before these machines were constructed can now be stated to be conservative at the 99% confidence level. A node MTBF in excess of 500,000 hours is probable, and can be stated now at a 55% confidence level. The systems will have to operate for another $2\frac{1}{2}$ years with a similar or smaller failure rate for us to be able to state a 500,000 hour MTBF with 90% confidence.

2.1.2 System Software Status

There have been no significant changes made in the cosmic cube system software in this period. Progress continues in converting some additional parts of the operating system kernel from 8086 assembly code to C to assist in bringing this kernel up quickly on future machines.

2.1.3 Intel cube

In October 1985 our first Intel cube, an iPSC d6 (64 nodes), was installed. In December it was upgraded to a d7 (128 nodes). These machines were contributed to the Submicron Systems Architecture Project as a part of the license agreement between the Caltech and Intel, and are accessible via the ARPAnet to other ARPA researchers who may wish to experiment with them. The iPSC host is `ipsc.caltech.edu` (or `ipsc.arpa`) on the ARPAnet. In order to request an account, please contact `chuck@vlsi.caltech.edu`.

The Intel iPSC systems are quite similar to the cosmic cubes, but have faster hardware and more primary storage per node. The system software does not include all the features of the cosmic cube due to limitations of the Intel 286/310 host, which runs Xenix, while the cosmic cube hosts run Berkeley Unix.

Each of the iPSC nodes consists of an Intel 80286, an 80287 floating point processor, eight ethernet channels, an RS422 channel, and 512K bytes of memory. Seven of the ethernet channels are used for bidirectional point-to-point cube connections, while the eighth is used as a global channel for communcation directly to the host. This global channel is an improvement over the cosmic cube's scheme, in which only node 0 has a connection to the cube host. (In the cosmic cube, node 0 and other nodes on the path from a process performing I/O to the host bear an extra load of routing a substantial volume of messages.) The RS422 channel in the IPSC is used for initializing the nodes, and for running diagnostics.

We have seen the performance of programs ported from the cosmic cube to the iPSC increase by a factor of three per node. For example, Chuck Seitz's Collatz sieve program, which spends most of its time in arbitrary precision integer arithmetic routines, runs six times faster on the Intel 7-cube than it does on the cosmic 6-cube. This performance on the iPSC d7 is more than 60 times faster than the best sequential program version of the Collatz sieve running on a VAX11/780.

Other benchmarks comparing the cosmic cube with the Intel cube show that the cosmic cube is faster for short (less than 80 bytes) messages, but asymptotically about one fourth as fast for long messages. These differences are due to the greater bandwidth but higher overhead in starting up the Intel cube's DMA-driven ethernet channels. Because of the higher message latency on the iPSC, relative to instruction rate, applications that use message passing intensively and tend towards shorter messages are best run on the cosmic cube, while problems involving coarser grain concurrency run much faster on the iPSC. A detail characterization of message-passing performance on both the cosmic cube and iPSC is currently being written up.

The Intel system software at present resembles an early version of the cosmic environment, but both performance and convenience continue to improve with later releases. In order to accomodate differences between the iPSC and cosmic cube functions, a compatability library was written that allows cosmic cube programs to run on the iPSC without changes in the source code. This library will track expected future changes in Intel's software.

The Intel hardware has been reliable, and the system software has been reasonably free of bugs. We have encountered several problems with the TCP/IP implementation for the 286/310 host, but the latest release appears to work reliably. We have sources for all of the system software, and have been able to make experimental modifications to and reassemble the node operating system.

2.1.4 Application programs

We continue to make the cosmic cubes available to selected guest researchers for problems that appear to be particularly interesting or difficult.

Several researchers from the Aeronautics department at Caltech, under the direction of Professors Brad Sturtevant and Tony Leonard, report very good results on some difficult fluid mechanics computations. Some of these computations are are based on a vortex model in which the interaction of vortices is computed in a way that is essentially similar to an n-body computation. Others of these computations employ a Monte Carlo method for computing local interactions for molecular dynamics, rarified gas (such as jet plumes), and granular flow, with a non-uniform grid to balance the load. These results and others were reported recently in a "Workshop on Applications of the Cosmic Cube to Selected Problems in Fluid Mechanics" organized by the Caltech Aeronautics Group.

A program written by Craig S. Steele of our research group is suggestive of the way in which these machines are used on problems that come up in our research. The Intel iPSC was used to support a conjecture in information theory. Mutual information is a measure of the maximum information capacity of a set of noisy communication channels. Essential mutual information is the information transfer possible if the probabilistic input/output function is replaced by the best possible "forced choice" of an output for each input wire. For most input probability distributions, the essential mutual information is a large fraction of the mutual information of the original noisy configuration. The question of just how bad the best "forced choice" can be seems analytically intractable.

Recently the problem was attacked by a randomized search of possible configurations. After considerable effort on conventional computers, a concurrent version was produced for the iPSC. For this particular problem, the iPSC d7 proved to be about 80 times faster than a VAX-11/750 under UNIX. It was possible to conveniently search 100 times more cases than previous efforts. A new lower bound on the ratio of the essential mutual information to mutual information was found, and the results supported the conjecture that the ratio is low only for configurations of extremely low information capacity.

Another problem that would be practically out of reach is our ongoing evaluations by simulation of preliminary designs for future message-passing concurrent computers. In most cases analytical techniques are not adequate to model blocking in the message-passing networks. An implementation of event-driven simulation (see section 3.5) using a technique similar to the Chandy-Misra-Bryant distributed simulation algorithm has been used to "benchmark" machines not yet constructed. Simulations of message-flow performance done in 1981 in connection with the cosmic cube design, and for machines up to 96 nodes, required about 1000 hours of DEC-2060 time. These more ambitious simulations, performed for machines of various sizes up to 1024 nodes, are expected to consume a similar amount of time on the cosmic cubes.

2.2 Mosaic Project

Over the past six months, the Mosaic project has been the focus of our system-building activities.

- With many trials and tribulations past, and more no doubt to come, we now have working prototype Mosaic A elements. This work is described in section 4.1.

- A number of ideas have developed – thanks to the work of several of the senior graduate students in the group – to the point where we are confident not only that we can program relatively fine-grain systems such as the Mosaic without extraordinary efforts, but that these fine-grain systems are efficient and surprisingly general. See sections 2.4 and 3.3.

- Some recent results on message routing and network topologies suggest that Mosaic systems with large numbers of elements can use a low-dimension (2D or 3D) mesh connection More complex topologies such as shuffles and binary n-cubes are unnecessary. See section 3.6.

- The design of a Mosaic C, in MOSIS SCMOS technology, is underway. See sections 2.2.2 and 4.3.

2.2.1 Mosaic software tools

Steve Rabin, Chuck Seitz

The Mosaic A assembler has been upgraded to be essentially compatible with the Mosaic B assembler, and several programming conveniences have been added to both assemblers. Modifying the assembler to accommodate the Mosaic C is expected to be straightforward.

Because bugs in Mosaic programs can garble the message system flow control, it is difficult to debug programs directly on Mosaic hardware. The Mosaic A element, which lacks interrupt hardware and clears its state on RESET, is unable to restore system context in the event of a serious error in the message system. For this reason most of our software development must be performed on a simulated Mosaic ensemble rather than on Mosaic hardware.

The current Mosaic simulation environment supports the simulation of Mosaic A and B ensembles with arbitrary connectivity, and with elements that are loaded with variable microcode and bootstrap programs. There is a simulated host interface to the ensemble, and provisions for loading data into or unloading data from the ensemble. Debug commands allow one to monitor communications and to examine the state of each computer in the ensemble. The simulation is, to the best of our knowledge, absolutely accurate on a clock cycle by clock cycle basis, in order to be able to replicate precisely any behavior of a Mosaic ensemble.

The cost of this precision is performance. The current implementation of the Mosaic simulator – written in MAINSAIL – executes roughly 50 simulated Mosaic cycles per VAX11/780 cpu-second. At this rate the Mosaic A v1.2 bootstrap program, which moves a march pattern through memory, requires roughly 6 VAX11/780 cpu-minutes multiplied by the number of machines in the ensemble being simulated. Fortunately, for bootstrap programs we are interested primarily in ensembles of one element.

Work is now under way to provide a concurrent simulation environment for Mosaic ensembles, intended to run either on the cosmic cubes or Intel iPSC computers. Our goal, now that the Mosaic processors are so well characterized, is to perform macroinstruction rather than microinstruction level simulation, and be able to simulate full size (*e.g.*, 1024 node) ensembles on the 128-node iPSC at no worse than 1,000 times slower than a hardware ensemble.

2.2.2 Mosaic C

Bill Athas, Lounette Dyer, Fritz Nordby, Steve Rabin, Don Speck, Wen-King Su, Chuck Seitz, and the 1985-86 Caltech VLSI Design Class

The design of a new Mosaic element, called Mosaic C, was started in January 1986. Some of the details of the chip design are described in section 4.3, and an overview of the architecture is summarized here.

- Target technology: MOSIS SCMOS with $0.6\mu\text{m} \leq \lambda \leq 1.5\mu\text{m}$. Target maximum chip size is 36mm^2, or $100\text{M}\lambda^2$ with $\lambda = 0.6\mu\text{m}$, and $16\text{M}\lambda^2$ with $\lambda = 1.5\mu\text{m}$. Speed, storage size, and top-level floorplan will necessarily vary with feature size.

- Storage: dynamic RAM, with refresh accomplished by the processor microcode as in Mosaic A & B, and small bootstrap ROM. 2T, 3T, and 4T RAMs are being designed. The primary storage is expected to require between 420 and 675 λ^2 per bit including peripheral circuits, depending on which design is used. The 2T RAM, with only $420\lambda^2$ and one contact per bit, is looking very promising.

- Processor: 16-bit, similar to the Mosaic B processor in layout and microcode style, except that the channels are implemented separately. Also, the Mosaic C processor is a "two-sequence" machine: It has two program counters and two sets of registers to be able to switch instantly between task and channel processing contexts.

- Channels: memory mapped, bit serial, two virtual channels per physical channel, with hardware assist for cut-through routing. These channels are a compromise between the performance of channels such as those on the Torus Routing Chip (TRC) (see section 4.2) and generality. The TRC channels also require more pins. Routing will be controlled by software, but after the routing decision is made the flow control units (flits) are shuttled from incoming to outgoing channel without processor intervention.

The processor, channels, ROM, and peripheral circuits on the Mosaic C are expected to consume about $10\mathrm{M}\lambda^2$. The amount of space left for storage depends strongly on feature size: about $6\mathrm{M}\lambda^2$ or 1.5K bytes at 3μm feature size – similar to Mosaic A –, about $26\mathrm{M}\lambda^2$ or 8K bytes at 2μm feature size, and up to $90\mathrm{M}\lambda^2$ or 24K bytes at 1.2μm feature size. The 3μm version can be used to prove the design. The 2μm version with 8K bytes of storage would be nearly ideal for experiments with message-driven programming styles (see section 2.4).

2.3 Future Architecture Experiments

Chuck Seitz, Bill Athas, Bill Dally, Craig Steele, Wen-King Su

We are developing a number of designs for second generation medium grain size message passing systems to combine the performance of commercial 32-bit processor chips and the low-latency communication of devices such as the Torus Routing Chip (TRC). Some of the critical design decisions are being investigated by simulations (see sections 2.1.4 and 3.5).

Since many cosmic cube programs display their results as images, and because these programs are I/O limited, we intend to include in these second generation systems a distributed frame buffer to allow the system to generate graphic displays directly.

2.4 Reflections on Models of Concurrent Computation

Chuck Seitz

In our experiments of the past year or so with message-passing concurrent computers, we have started to introduce an object discipline into the use of the concurrent process model on which the cosmic cube and Mosaic are based.

In the basic process model, concurrent processes communicate by asynchronous, non-blocking *send* and *receive* operations, and messages are queued while they traverse the message system. These aspects of the process model appear to be dictated by physical design considerations in VLSI technologies. The way in which this model is captured in programming notations has its roots both in Hoare's Communicating Sequential Processes (CSP) notation and in Lang's work in our own group on concurrent extensions of object-oriented programming notations [5014:TR:82] such as Simula.

We like to think of CSP as the FORTRAN of concurrent programming notations, because in CSP programs are static entities. The process structure – the set of processes that participate in the computation and their references to other processes – remains static for the duration of a computation. Processes communicate with operations in which (by definition) the number of completed *send* and *receive* operations on a channel are identical, much like signaling on a wire. CSP is then, to us, a satisfactory notation for describing either hardware (*c.f.*, section 4.5) or quite simple programs, but is not able to express programs of a more dynamic and interesting character.

Our process model, as expressed by adding extra primitives to programming languages such as C, is much more "liberal" than CSP. The synchronization between *send* and *receive* is potentially unbounded. The number of completed *receive* operations of a given type is not more than the number of completed *send* operations of the same type. Our model is also liberal in its notion of a "channel". All that a process requires to send a message to another process is reference to the other process, and reference can be passed in messages. Finally, we permit processes to spawn or to kill other processes. One can express programs with the dynamic character that we prefer. However, all of these capabilities are invoked explicitly.

This process model has a number of virtues. It is more than adequate for many types of scientific computations, and is compatible with familiar concepts from operating systems based on processes and message-based interprocess communication mechanisms. It is extremely powerful: any *trigger* mechanism [1] can be programmed. For example, it is relatively easy to build an object-oriented programming environment on top of the process model with the methods run under a receive-dispatch-execute loop.

The object-oriented programming notations used by Bill Dally (Concurrent Smalltalk, or CST) to express concurrent data structures, and by Bill Athas (Experimental Concurrent Programming Language, or XCPL) for compilation experiments, forego the use of an explicit *receive* in favor of the objects being *reactive* or *message-driven*. In this respect these programming notations are Actor [2] languages. An Actor similarly *reacts* to an incoming message by *send*ing a (possibly empty) set of messages, creating a (possibly empty) set of *new* Actors, and changing its state (or behavior, or, in Actor semantics, it *becomes* another Actor). This trio of *send, new, becomes* is then the foundation of Actor semantics. One important difference in CST or XCPL objects is that they can exercise *discretion* in which messages they may react to. This concept was also present in Lang's work [5014:TR:82] of four years ago. We have persisted in the belief that the message system rather than the objects is right place to buffer messages.

It appears that there are sound reasons both with respect to programming style and architecture to use an object rather than process model for our next round of experimental machines – both those machines based on commercial processors and the Mosaic C. From the programming standpoint, the object model introduces a discipline into the use of *send* and *new*. From the architecture standpoint, recently developed message routing techniques reduce communication latency (see 3.6 and 4.2) to an extent that puts pressure on the node processor to reduce the overhead associated with *send* and *receive* operations. There is some speed advantage for a node to schedule process (object) execution based on received messages – that is, to make the processes (objects) reactive and the processors message-driven (see 3.7). It also appears that the operating system of a node can be made simpler than, for example, the cosmic kernel, if its entire job is (1) to run a process (or a particular method, if the method lookup is in the operating system) according to the messages received, and (2) to be able to spawn new processes. The organization of Mosaic C as a two-sequence computer is motivated in part by the desire to make it a good testbed for this model. The communication sequence, in addition to controlling message routing, would be able to provide the desired front-end services for message-driven operation of the node with no context-switching delay.

References:

[1] see Kotov's classification of trigger mechanisms in chapter 5 of *Algorithms, Software, and Hardware of Parallel Computers*, edited by J Miklosko and V E Kotov, Springer-Verlag, 1984.

[2] Gul A Agha, "Actors: A Model of Concurrent Computation in Distributed Systems", Technical Report 844, MIT AI Lab, 1985.

3. Concurrent Computation

3.1 The Sync Model for Parallel Execution of Logic Programming

Pey-yun Peggy Li, Alain J Martin

The Sync Model is a multiple solution data driven model which realizes AND-parallelism and OR-parallelism in a logic program assuming a message-passing multiprocessor system. The Sync Model constructs a dynamic tree of processes to represent the AND/OR proof tree of a logic program and performs breadth-first search on the tree. AND parallelism is implemented through the contruction of dynamic data flow graph of the literals in the clause body by an ordering algorithm. OR parallelism is achieved by adding special Synchronization signals to the stream of partial solutions and synchronizing the multiple streams with a merge algorithm.

In the past six months the Sync Model has been improved in several aspects: (1) Besides AND and OR parallelism, stream parallelism is also implemented in the model. Stream parallelism is the pipelining of the shared list data structure between two sibling AND processes. It is implemented by identifying the "stream variables" in the program automatically and transmitting the partial solutions of the stream variables with special communication protocol. (2) Tail recursion is optimized in two ways: by using a *map* operator to transform a tail recursive clause into a set of parallel AND processes to perserve the inherent parallelism in the definition, and by looping back and repeatedly using the same process for the recursive calls of the same goal. (3) The data flow graph is refined to avoid unnecessary computations. "Selective channels" are added to the literals that have zero outdegree in the graph to transmit the input values that make the literals true. The refined graph is also maintained acyclic to prevent deadlock.

The performance of the mapping of the Sync Model onto the Sneptree is analyzed in terms of the load factor and the communication overhead. The simulation results show that the *Exchange Sneptree* is one of the best connection patterns for the Sync Model, where the Exchange Sneptree is a Sneptree in which all the outgoing links of the leaf nodes in the left half are directed to the incoming links of the nodes in the right half, and vice versa. We also found that load balancing heuristics do not improve the performance of the mapping. After comparing this mapping with the mapping of an arbitrary tree onto other networks, such as a mesh or a binary n-cube, we have found that the Sneptree is indeed the ideal architecture for the Sync Model.

Full details can be found in "A Parallel Execution Model for Logic Programming", Peggy Li's Ph.D thesis (in preparation).

3.2 A descriptive model of computing agents

Kevin S. Van Horn, Alain J. Martin

One indication of how well we understand some phenomenon is our ability to provide an adequate mathematical model of it. Such a model provides a firm basis for reasoning about the phenomenon; in its absence we are vulnerable to the treacheries of informal reasoning, and we are severely limited in our ability to analyze complex instances of the phenomenon. What is the appropriate *general* model of computing systems? This model should be applicable to both hardware and software systems, and must of course be applicable to concurrent systems. As a candidate for this role, we have developed a general model of computing agents, called Complete and Infinite Traces, or CIT.

We take the view that a computing agent is an object which may perform various *actions*, thus exhibiting some discrete *behavior*, which may be influenced by the actions of other agents. In the case of a digital circuit the relevant actions are voltage transitions. A computer program may perform such actions as assigning a value to a variable or writing a character to the user's screen, and may respond to actions such as the user hitting a key on the keyboard.

The mathematical structure used to model a computing agent is called a *process*, and is a set of traces (sequences of actions) plus an input and an output alphabet (sets of actions performable by exterior entities or the agent itself, respectively). If we imagine an observer who records the entire sequence of actions which are performed during the operation of the computing agent, every such sequence which he might record (called a *complete* trace) will be in the trace set of the process modeling the agent. If the agent has a cyclic behavior and may continue to operate forever, then the trace set will contain infinite traces.

To resolve some difficulties associated with infinite traces, we assume that all processes are either *sequential* (i.e., they model an agent which performs its output actions one at a time) or may be formed as the parallel composition of a set of sequential processes. Since we use recursion only in defining sequential processes, we may define parallel composition using the weave operator from trace theory [1] and thus avoid the problems associated with defining parallel composition as some form of "fair merge".

A detailed presentation of this model of computing agents may be found in [5207:TR:86].

At present, CIT is being applied to the semantics of concurrent programming languages. Concurrently (of course), compositional techniques for verifying such programs are being investigated, based on CIT.

Reference:

[1] J. L. A. van de Snepscheut. *Lecture Notes in CS, vol. 200: Trace Theory and VLSI*, Springer-Verlag, 1985.

3.3 Computation Flow

William C. Athas, Chuck Seitz

Research into compilation techniques for concurrent message passing computers (ensemble machines) has progressed from the definition and demonstration of a high level programming notation for describing concurrent objects [5196:TR:85] to research into program flow analysis. The program flow analysis is based upon a finite state process model for describing the basic or primitive behavior of a computation. The processes are capable of sending messages and spawning new processes. The finite state process shares many properties with the actor model of computation, although for our purposes it has been specifically tailored for program flow analysis. As a result of basing the program flow framework on this model, two addition frameworks of future and after flow are derived.

Future flow allocates new processes (objects) before the processes are actually needed by a computation. This allocation can be performed statically at compile time to build an initial process graph for placement on the ensemble machine, or can be done concurrently with the computation. By concurrently performing future flow analysis with program execution, the interval between the specification of a process and its subsequent use is increased. This additional time may be used advantageously in finding better locations for new processes with respect to load balancing and message traffic.

After flow is akin to the task of garbage collection as performed in LISP systems. Whereas future flow attempts to identify processes that are to be used in the future of a computation,, after flow identifies those processes that are still relevant to a computation, though not necessarily active. By determining which processes are relevant, or other processes are thereby irrelevant to the outcome of the computation and can be safely reclaimed. Like future flow, after flow can be concurrent with program flow. The concurrent execution of all three is called computation flow.

3.4 Concurrent Simulated Annealing

Craig S. Steele, Chuck Seitz

Preliminary experiments indicate that the previously reported simulated annealing technique, used for process placement, is tolerant of minor database inconsistency. Some such "staleness" of data is likely in a highly concurrent computer due to communications latency. A concurrent implementation of the process placement optimization program is being designed, and is targeted for the Intel iPSC.

In addition to the goals of greatly increased optimization speed and larger maximum problem size, it is intended that the program should be written in such a way that it may improve its own communications performance by optimizing its own placement.

With the above goals in mind, a "quasi-static" programming environment (QSE) is being designed. This environment combines medium-sized data structures with C code operator modules, similar to small subroutines. Programs are written by specifying the logical connection between specific code and data components. Messages required to update remote copies of data are generated by the runtime system rather than by the programmer's explicit use of *send* services.

This approach removes the low-level details of message addressing from the code and allows relocation of computation components during execution. Debugging is eased because most components of system state are externalized and visible to the debugger via the runtime environment. Success of the QSE will provide a convenient environment for scientific programming applications based on iterative methods, and should be a significant step in the evolution of a fully dynamic concurrent programming model.

3.5 Event-driven Simulation on Message-Passing Concurrent Computers

Wen-King Su, Chuck Seitz

Event-driven simulation is an important class of computations in which different parts of the system being simulated interact by scheduling events to take place at some time in the future. Each event may in turn cause more events to be scheduled at a later time. On sequential computers the events are performed in chronological order by a scheduler that maintains a single forward moving clock.

A subset of such simulations map extremely well onto message-passing concurrent computers. This subset includes simulations in which the system being simulated includes a sufficiently large number of elements and activities going on at once to keep a concurrent machine busy. On a large concurrent machine, however, maintaining a universal clock is difficult. We are currently exploring an approach to event-driven simulation on message-passing concurrent computers in which the elements are connected by logical channels through which the events are delivered in the form of messages.

The basic technique, which is closely related to the distributed simulations algorithm suggested by Chandy and Misra, and by Bryant, is that an element can produce its outputs up to a future time $t + s$ if it can predict its output s time units in the future based on the inputs up to time t. Any element that does not respond instantaneously – which includes all physically realizable elements, has this property. In such a simulation the available slack, s, contributed by the elements is used increase the performance of the simulation. Messages are tagged with times, and all scheduling is local. The algorithm permits different elements of the simulation to get as much out of step as is permitted by their mutual dependencies, thus allowing a maximal number of elements to be scheduled for evaluation while not doing any redundant work.

A prototype simulator has been written and is currently giving the cosmic cubes a real workout. We are using the prototype simulation package to evaluate the performance of alternative network topologies and routing schemes under several program models. These simulations are part of our efforts at designing second generation message-passing systems (see section 2.3).

3.6 Interconnection Network Topology

William J. Dally, Chuck Seitz

A comparison of the latency of k-ary n-cube networks as a function of dimension gives the surprising result that, holding wiring bisection width constant, relatively low dimension networks (*e.g.*, the 2-dimensional torus) achieve lower latency than do high-dimensional networks (*e.g.*, binary n-cubes). The minimum latency is achieved when the component of latency due to message length is approximately equal to the component of latency due to distance. For networks of fewer than 1000 processing nodes, minimum latency is acheved by two-dimensional networks.

3.7 Message-Driven Processor

William J. Dally

A message-driven processor reduces message latency by directly executing messages rather than interpreting them with a series of instructions. In a concurrent computer built around a conventional instruction processor, interpreting a message is a time-consuming process. The processor must respond to an interrupt, save its state, fetch the message, look up the method, and finally (after executing about 100 instructions) execute the method.

Instead of nesting the fetch-decode-execute loop of a conventional instruction processor inside the receive-dispatch-execute loop required to process a message, a message-driven processor directly interprets messages. The reception and buffering of messages is performed by hardware so that no interrupt is required to receive a message. Method lookup is accelerated by using an instruction-translation lookaside buffer. Primitive methods are executed directly by the processor as if they were instructions. Defined methods are executed by creating a context object and performing a series of *sends*.

Current work involves specifying the architecture of a message-driven processor and constructing a simulation model of this architecture. The goal of this work is to integrate a message-driven processor with a TRC-like communications device and memory on a single chip to form a building block for a future fine-grain concurrent computers.

3.8 Concurrent Smalltalk

William J. Dally

The Concurrent Smalltalk (CST) programming language is being developed to combine the data abstraction and late binding features of Smalltalk-80 with the message-passing semantics of actor languages. CST includes features that allow the programmer to create abstractions for concurrency. A key feature of this language is the ability to specify *distributed objects*. The state of a distributed objects is distributed among a collection of *constituent objects* each of which can receive messages on behalf of the distributed object. Thus distributed objects can perform many operations simultaneously. They are the foundation upon which concurrent data structures are built.

3.9 A General Purpose Concurrent Architecture

John Y. Ngai, Chuck Seitz

This research, still in its very early stages, is an investigation of computing structures that can support both concurrent and general purpose computing. The ideas involved have been discussed in the research group under the heading of "soft computers", and are based on programming arrays of very small computing elements in a way that is specializes the ensemble to interpret a program representation that is tailored to the program itself.

The idea is also akin to that of silicon compilation. Instead of compiling the high level description of a computation directly into physical circuits elements such as wires and transistors, it would be translated into processes or objects, each accomplishing a certain part of the computation being compiled. The analogies between component or cell placement and process placement to achieve locality of communication is clear. These processes execute concurrently on a fine grain computing mesh.

Certain elements of this idea are very old, and were investigated by John von Neumann [1] and Arthur Burks [2] in the late 1950s in their study of cellular automata theory. One can build fine grain structures that can be programmed and composed together to form larger architectural blocks. Exactly how much concurrency one can achieve depends on the algorithmic formulation of the computation. As with XCPL [5196:TR:85], the source text is necessarily concurrent, and the compilation can at best preserve the concurrencies in execution.

Our current investigation focuses on finding suitable *soft* fine grain structures used in composition, and the corresponding simulation techniques. It appears, as in [1], that the ability to form communication paths is a necessary primitive. Future work will include a study of compilation techniques and composition rules, especially those needed to support an object-oriented programming style.

References:

[1] John von Neumann, *Theory of Self-Reproducing Automata*, Urbana, Univ. of Illinois Press, 1966.

[2] Arthur W. Burks, *Programming and the Theory of Automata, Essays on Cellular Automata*, Univ. of Illinois Press, 1970.

4. VLSI Design

4.1 Mosaic A Elements

Don Speck, Chuck Seitz

The following narrative description of the final stages of a long-term effort on the Mosaic A elements may be of interest to other VLSI program contractors who have gone through or expect to go through similar experiences.

As discussed in previous semi-annual technical reports, the two major parts of the Mosaic A element, the processor and dynamic RAM, were fabricated and tested separately. The processor gave us little trouble, but the RAM, with its exotic circuit design style, required many fabrication runs and extensive circuit simulation to achieve a robust design.

With the major parts designed, there remained only to design a ROM and to integrate those parts into a single chip. The parts all hang off of a central address/data bus, so predictably enough, that is where what few interface problems there were showed up. In designing the address drivers, it eventually became clear that the highly capacitive address lines – forced by the RAM layout to run more than twice the length of the chip – could never rise as fast as the ROM design had assumed. Timing pressure from the slow address inputs forced the ROM decoder to mutate into the same extreme hot-clock design style as the RAM decoder, sacrificing some of the desired implementation independence of the ROM from the more suspect RAM.

Since Mosaic elements are stored-program computers, they must have an initial program, contained in the ROM. A bootstrap loader is the minimum, but if any subsystem proves inoperative, the loader will likewise be inoperative, and it becomes impossible to load any diagnostic programs. In particular, we did not want to depend on the RAM, the most process-sensitive part of the chip, in performing RAM tests. Therefore, the diagnostic programs, particularly the RAM test, must already reside in the ROM if one hopes to gain any diagnostic information from partially-working chips.

In hopes of getting some early feedback on yield, a "snapshot" of the evolving fully assembled chip was sent to MOSIS even before full simulations could be attempted. In this "snapshot" version, Mosaic 1.0, the ROM code performs a purposely simplified port handshake (designed to be satisifed by a passive pullup), computes the ROM checksum, scans the RAM for errors, and reports both results before attempting to download a program. In this way the chip provides enough information to forecast yield, even if it proves unable to download any programs.

Subsequent simulation revealed that the "snapshot" version was, in fact, too defective to download a program, but still able to run its ROM/RAM tests. The downloader routine contained an error despite the simple-and-safe coding style, and an inverter was omitted from the input ports, causing the flow control circuits to ground out the input. The latter defect could be bypassed for test purposes by shorting the input to Vdd, in place of the passive pullup.

The first batch of these chips received from MOSIS yielded 4 of 7 chips with all 96 words of ROM and 512 words of RAM working correctly. Thus the "snapshot" and the paranoia in the test code were well justified. However, the chips only worked over a limited range of clock amplitudes (4.1 volts $\pm$ 5%) and clock speeds (1.5 to 5 MHz). These same limitations were characteristic of an earlier version of the RAM, and may indicate some residual process sensitivity even in our improved version. Future fabrication results may help explain these test results.

In the meanwhile, we have gone through the extraction, switch-level simulation, and bug-fixing cycle through fabricated versions 1.2, 1.3, and 1.4 with the 1.3 and 1.4 versions being free of any known bugs. The full suite of switch-level simulation tests with Mossim II requires more than 20 days on a VAX11/750.

The ROM code in Mosaic 1.3 & 1.4 has been made flexible enough to use for diagnosis, testing, and program loading. The size has quadrupled with the addition of specific tests for every subsystem except the microcode. Together, the tests are so comprehensive that they diagnosed 4 bugs in the instruction simulator. None of those bugs exist in the chip itself, as determined by simulation with Mossim II.

The Mosaic 1.3 is expected to be producible in quantity. It has 768 words (1.5K bytes) of RAM, 392 words of ROM, 65,000 transistors, a die of 4.6 by 7.1 mm^2 (including the MOSIS test structures) in a 3μm MOSIS nMOS process, and 17 pads. The pads are all located on the 4.6mm edges, so that the chip fits perfectly in a standard 18-pin dRAM package.

The use of a 2D mesh connection allows very simple packaging of Mosaic systems on PCBs with 256 node elements and 32 clock driver chips per board.

4.2 The Torus Routing Chip

William Dally, Chuck Seitz

First silicon of the Torus Routing Chip (TRC) was received in December. These chips functioned properly; however performance was below expectations. Channel period was measured to be 250ns and channel delay 150ns. Even with this dissapointing performance – due to an oversight in the location of the critical timing path – the TRC offers a substantial improvement in performance over existing communication networks. For example in a 1024 processor 32-ary 2-cube a message across half the diameter of the machine crosses 31 channels with a latency of only 4.65μs (31×150ns). In contrast, store and forward networks used by machines such as the cosmic cube have single channel latencies in the order of a millisecond. The torus routing chip reduces these millisecond latencies to microseconds, making possible much finer-grain concurrency.

Continuing work aims at improving the performance of the TRC. A 1.4μm feature size version of the TRC was submitted for fabrication in January, and a redesign of the TRC to improve performance and function is in progress.

A preprint of a paper describing the Torus Routing Chip is reproduced as an appendix to this report.

4.3 Mosaic C

Bill Athas, Lounette Dyer, Fritz Nordby, Steve Rabin, Don Speck, Wen-King Su, Chuck Seitz, and the 1985-86 Caltech VLSI Design Class

A new version of the Mosaic element, called Mosaic C, was started in January 1986. The target technology is MOSIS SCMOS with $0.6\mu m \leq \lambda \leq 1.5\mu m$. The architecture is discussed in section 2.2, and some of the rationale behind the unusual features of the element is found in sections 2.4, 3.3, 3.6, and 4.2.

A first cut logic design and layout is being done by the 32 "hard-core" members of the full-year VLSI design class, all of whom have already done projects in the fall quarter. The design effort is being led by seven of our most experienced chip designers. The class has been broken into four groups, each group led by two experienced designers:

RAM/ROM: (Group leaders: Don Speck, Fritz Nordby) This group has developed designs and done extensive simulations for 2-, 3-, and 4-transistor dynamic RAMs. The 2T RAM cell, essentially two 1T cells written with complementary data and sensed differentially, is both the densest possibility and – because it uses only one contact per bit – is likely to have the highest yield. The density, including decoding circuits, is about $400\lambda^2$ per bit. For this RAM we are aiming for medium speed, approximately 10 MHz operation at 3μm feature size, and relatively low (dynamic) power consumption.

Channels: (Group leaders: Steve Rabin, Chuck Seitz) The channels group has had to start at a higher level of design than the other groups, in that it was decided for architectural reasons that the Mosaic C channels should be substantially different from the Mosaic A channels. The channels designed are memory mapped: they appear to the programmer as special storage locations. Each channel is bidirectional, synchronous, and bit serial, with one wire in each direction. Unlike the Mosaic A channels, the wires are always driven. The flow control unit (flit) is one Mosaic word, and there are two virtual channels per physical channel. Each transmission starts with a lead bit, followed by a bit to distinguish a message from an acknowledge, and followed by a bit to specify the virtual channel. If the transmission is a message rather than an acknowledge, this preamble is followed by a tail bit, one or more programmable control bits, and 16 bits of data. The channel interface also includes a hardware assist for wormhole and cut-through routing.

Datapath: (Group leaders: Wen-King Su, Chuck Seitz) The processor datapath has the same general floorplan as that in the Mosaic B, but in SCMOS rather than *n*MOS, it quite different in its logical design and layout. The bit pitch is a generous 60λ, so that the 16-bit datapath is 960λ high. At this bit pitch the datapath is approximately 2500λ wide. The datapath employs a number of very nice logic tricks, including a double-rail carry chain to allow the precharged shift network to be cascaded with the precharged adder. Since the Mosaic C processor is a "two-sequence" machine, it has two program counters and two sets of registers.

Microcode: (Group leaders: Bill Athas, Lounette Dyer) The microcode group is using the Mosaic B microcode as a model, and have devoted their initial efforts to developing simulators to be able to test the somewhat more complex microcode to operate the Mosaic C.

We expect to finish all of the Mosaic C parts by June, and plan to fabricate the processor, RAM & ROM, and channel sections independently before combining them onto one chip.

4.4 SCMOS Pad Designs

Fritz Nordby, Chuck Seitz

All SCMOS chip projects need pads. The designs available six months ago, developed at MIT, were perceived as deficient in several respects, including inadequate or misdirected static protection structures, improperly sized output transistors, and over-large real estate requirements. A new set of SCMOS pads was designed to fill a perceived need.

The first versions of these pads were designed for 3.0μm SCMOS. These pads employ a polysilicon resistor for input protection, and output transistors sized to drive a 10nH lead inductance into a 50pF load without ringing. These pads have been fabricated both in a test configuration and in a larger chip (the TRC). The pads function properly in both cases; their performance is now being characterized, and will be used to evaluate possible design changes.

In addition to the 3.0μm pads, a set of pads were developed for what Ms. Mosis refers to as "1.2 micron" SCMOS, that is, the technology for which $\lambda = 0.7\mu$m. In this case, the primary concern was to reduce the pad pitch to avoid wasting chip area in pad-limited designs. The pads can be arranged in two rows with staggered pad centers, a pattern fairly commonly used in industry. Also, a different static protection strategy was employed in this design, since the space available to these structures is considerably smaller than in the case of a single row of pads. These pads are currently being fabricated as part of the 1.4μm (sorry, 1.2μm) version of the TRC.

4.5 Compiling Programs into Self-timed VLSI Circuits

Steve Burns, Alain J. Martin, Kevin Van Horn

As previously reported, we are developing a method for deriving self-timed VLSI circuits from a high-level description of a computation – a set of communicating processes. An overview of the method can now be found in: A J Martin, "Compiling communicating processes into delay-insensitive VLSI circuits", *Journal of Distributed Computing*, vol.1, no.3, 1986, a preprint of which is reproduced in the appendix to this report.

Recently we have found a solution to the delay-insensitive fair arbiter problem that is simpler than the one described in [5193:TR:85]. This new solution also presents the important advantage of being readily extensible to an arbitrary number of requesting processes. The solution can be found in [5219:TR:86].