SUBMICRON SYSTEMS ARCHITECTURE

SEMIANNUAL TECHNICAL REPORT

Sponsored by
Defense Advanced Research Projects Agency
ARPA Order Number 3771

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

5160:TR:84

Computer Science
California Institute of Technology

October 1984

Submicron Systems Architecture

Semiannual Technical Report

Computer Science
California Institute of Technology

5160:TR:84

October 1984

Reporting Period: 16 April 1984 to 15 October 1984

Principal Investigator: Charles L Seitz

Faculty Investigators: Randal E Bryant
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-C-0597

SUBMICRON SYSTEMS ARCHITECTURE

Computer Science
California Institute of Technology

1. Overview and Summary

1.1 Scope of this Report

This document reports the research activities and results for the seven month period 16 March 1984 to 15 October 1984 under the Defense Advanced Research Project Agency (ARPA) Submicron Systems Architecture Project.

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].

1.3 Highlights

Some highlights of the previous 7 months are:

(1) New Cosmic Cube system software (2.1.2).

(2) Development of a concurrent circuit simulation program (3.1).

(3) New techniques in program verification (3.3).

(4) Construction and testing of the MOSSIM Simulation Engine (4.1).

(5) Symbolic representations of Boolean functions, with applications to fault modeling (4.2, 4.3).

(6) Bit-serial Reed-Solomon decoders (4.4).

(7) New results in error correction (4.5).

2. Architectural Experiments

2.1 Cosmic Cube

W C Athas, Reese Faucette, Chuck Seitz

A draft of a paper that describes the Cosmic Cube experiment is included in the appendix to this report.

A non-exclusive license covering design and patent rights, and a resale license of the operating system, has been negotiated. A commercial version of this system is in development by the licensee.

2.1.1 Hardware Status

Our 64-node and 8-node Cosmic Cubes continue to run reliably. With at this time some 600,000 node-hours of operation and only two hard failures logged, both in dRAM chips, the calculated MTBF of 100,000 hours can be regarded as conservative at the 98% confidence level.

To recap from the last report, the only recent hardware renovation was the replacement of the original Cube Life Support (CLS) board with a new version called the Rosebud CLS. This renovation was necessary due to a noise problem in the intermediate host to cube cable connection. We took advantage of this redesign to support all the functions of the old CLS board in a design that is a generic Multibus interface board for all the ensemble machine experiments underway.

Since Rosebud and other design projects in the department have come to rely heavily on Programmable Array Logic (PAL) chips for board level TTL designs, a program was written to convert logic functions expressed as truth tables into PAL equations. Since only small switching expressions can be stored in the PAL chips, the program applies all possible optimizations to the logic function to find the minimal expression. Interest has been expressed to use this program for PLA optimization in VLSI chip projects.

Once this program was completed and documented, the PAL ICs for Rosebud were programmed and Rosebud was constructed. The remedy for the noise problem included different connectors than the old CLS board, and this in turn required that the corner node boards be rewrapped.

A SUN fileserver, ARPAnet address CIT-SOL, now serves as a Cosmic Cube intermediate host. Since the connection is by a 30-foot 60-conductor ribbon cable (with alternate wires grounded), it is a good acid test for the new interface.

Under separate support (DoE) program-compatible cosmic cube "Mark II" systems are being assembled by a group at JPL for the Caltech user community. The user group now has their own 5-cube running, and their current use of the original 3-cube and 6-cube is being phased out in favor of computer science and network users.

2.1.2 System Software Status

The inner kernel of the Cosmic Kernel (CK) is now essentially complete. The scheduling, message passing, and memory allocation routines have been thoroughly tested and are quite robust. Several different programs have been written which run under the CK and give it a good workout. The internals of the kernel have changed several times in the past six months, but the design and documentation of the interface between the operating system and the user processes was sufficiently complete that the code for these programs has not had to change at all.

The message passing system has had many additions over the course of the summer. Some tuning was done to improve the speed. Message fragmentation has also been added so that long messages can be passed across the cube without requiring that every intervening node have sufficient storage available to buffer the entire message. There is a parameter to the system which defines the longest allowed single message, and any message longer than specified by this parameter is fragmented. The overhead associated with the

fragmentation process is two headers per extra fragment generated. The fragment parameter is currently 256 words.

Message passing between processes in the same node (local SEND) has been fixed so that the message is buffered if the target process has not yet executed a RECV, rather than keeping the message in the data area of the sending process, as is done for non-local SENDs. This oversight would have resulted in different semantics for local and non-local SENDs.

The memory management scheme was slightly modified recently to use indirection pointers rather than returning the actual address of the memory requested. This makes the job of the memory compaction routine much simpler since it no longer needs to know where programs that request memory keep their pointers, so it can go out and change them at compaction time. As a result of being simpler, the compaction algorithm is also now much faster.

One demonstration program that runs on the cube under the cosmic kernel consists of six user processes which are loaded into memory along with the kernel. The user processes communicate with one another as well as with the host, and use the LED's attached to each board to demonstrate message flow. Many properties of the message system can be observed using this program, such as queuing of messages, independence of messages intended for different processes in the same machine, and message routing implications.

The sixth process does not use the LED's, but is a prototype of the "spy" process mentioned in the paper in the appendix. This process is used to read memory from any board back to the host. A message is sent to this process from the host containing a starting address and the number of words requested. A reply is sent back to the host containing the contents of the memory. This operates concurrently with the LED display without affecting it, and hence can be used to observe the internal happenings associated with the message traffic, and display on the SUN intermediate host histograms of message locations.

The "spy" process described above also exists as part of a "post-mortem" debugger. This is a version of the cosmic kernel that loads into high memory and can be used to look at the memory of a crashed program to determine the cause of failure in an otherwise very unrevealing environment. This process is being refined to provide information about processes that are running, such as their current program counter or values of variables. It will also be used to monitor system resources like free memory, idle time, or message activity.

Several compilers have been found that produces code suitable for running on the cube, and will be used to compile both user and outer kernel processes.

The intermediate host (IH) is now a SUN-II file server running Berkeley Unix 4.2, ARPAnet address CIT-SOL. The Rosebud CLS board has been installed in the SUN and a device driver written for it. The 8086 assembler has been brought up on this machine, and it is now being used for all development as well as being the intermediate host.

With the Cosmic Kernel and host tools now quite stable and reliable, and with such a nice intermediate host, our attention is shifting to the IH software. The IH software runs in a Unix and local network environment that is quite similar to the process programming environment supported by the kernel running in the cube.

A "cube-master" program is planned for the IH that will support communication between user processes in the cube and Unix processes in any Unix machine connected to the host by ethernet. Thus, one will be able to run programs on the cube in conjunction with programs on any network connected machine, where currently the cube computations run in conjunction with processes that run only in the IH. This mode of operation is possible because functions such as process spawning are invoked in the CK by messages rather than by system calls. We are planning also to provide in the "cube-master" time- and space-sharing of the cube.

2.1.3 Programming Techniques

Several programs intended to exempify and demonstrate programming techniques have or are being written for applications such as computer graphics, simulation, and game playing. Preliminary user's guides have

been written, and this documentation together with programming examples will be made available on-line on CIT-SOL in the near future.

2.1.4 Advanced Technology Homogeneous Machines

We continue to study designs of advanced technology nodes for homogeneous machines with characteristics described in a previous technical report [5103:TR:83].

2.2 Mosaic Systems

Chris Lutz, Steve Rabin, Don Speck, Chuck Seitz

Work is proceeding on schedule for assembling a 1024-node system based on the single chip Mosaic node. The chip design aspect of the project is described here, and the programming aspects in part 3.

2.2.1 Mosaic Processors

We continue to fabricate and test Mosaic A prototype processors for yield evaluation for ourselves and for MOSIS. The speed of current processors fabricated at 3 μm feature size is limited entirely by the pad frame and test jig to about a 55 nsec cycle time (see 2.2.3 below), from which we conclude that our target of 10 MHz in-system operation will be exceeded. We have sufficient A processors for a useful size prototype for programming experiments, and this system, including host interfaces, is being constructed.

The Mosaic B processor microcode is thoroughly tested, and about 3 months effort in layout and verification are required to complete this design.

2.2.2 Mosaic RAM

The prototype dynamic RAM module (later to be integrated with the Mosaic processor and communication channels) was received from MOSIS, tested, and found to have a geometrical design rule error that obscured the test results. The layout was fixed, and a second iteration is being fabricated.

2.2.3 Mosaic Clock Driver

Most of the effort of at-speed testing of the RAM module (and sets of Mosaic prototypes as yield monitors for MOSIS) is in producing fast clock waveforms in the slow TTL technology of wire-wrapped test jigs. *n*MOS chips designed in the "hot-clock" style give very good performance, but require external clock drivers to drive clock loads very fast with almost no skew. Mosaic processor and memory parts fabricated separately run with clock pulses as short as 15ns. We have designed a small CMOS/SOS clock chip to produce 0.6 watts/chip of precisely timed Mosaic clock waveforms, derived on-chip from a 50 MHz reference. Copies of these chips are in the June 1984 MOSIS SOS run, and first silicon (sapphire) from MOSIS is expected any day.

2.3 Super-mesh

Wen-King Su, Chuck Seitz

Super mesh is a fine-grain mesh-connected SIMD machine designed to study limiting points in synchronization, power, cost, chip area, and speed of high performance VLSI engines for highly regular high-precision numerical computations. At the heart of each node is a very efficient serial floating point unit that uses carry-save mantissa multiplication.

A trial layout of the floating point unit was completed in June with a team of students from the VLSI design laboratory course. A review of this design suggests some improvements can be accomplished with a somewhat different layout strategy, and a revised floating point unit is underway.

2.4 Other Experiments

The MOSSIM Simulation Engine (MSE) constructed in this reporting period is described in section 4.1.

A 10-node message-passing SIMD ring machine for orbital mechanics computations, called the "Orrery", was designed and built in a project led by Gerry Sussman while he was on sabbatical at Caltech.

3. Concurrent Computation

3.1 CONCISE: Preliminary Results

Sven Mattisson, Chuck Seitz

A concurrent circuit simulator for the Cosmic Cube is under development. The program is called CONCISE (CONcurrent CIrcuit Simulation Experiment), and a first version running on a sequential machine has been implemented. The programming language used is Pascal. No special language features are needed; only predefined message passing routines are required for interprocessor communication.

Preliminary results are promising, and the performance of CONCISE on a VAX is comparable to SPICE2.

A circuit simulation program solves a set of nonlinear equations

$$F(v(t), e(t), t) = 0$$

where v(t) is the unknown node voltage vector, e(t) is the source vector and t is time, for a specified time interval. In order to solve this equation concurrently, some of the algorithms commonly used on a sequential computer have to be replaced with others better suited for concurrent execution.

The different steps in a circuit simulation program can be described as: (1) formulation of network equations, (2) numerical integration, (3) linearization of the nonlinear network equations, and (4) sparse matrix equation solving.

The greatest impact on the concurrent program design is in the equation formulation method and the matrix equation solving algorithm.

The properties of the coefficient matrix are determined by the network formulation method used. The nodal formulation is attractive since it yields a matrix with a tendency towards diagonal dominance and with preserved circuit sparsity and connectivity. However, the nodal formulation has a deficiency in only being able to handle elements with an admittance description, but problems due to this nongenerality can almost always be circumvented by more realistic device models. The modified nodal formulation is an extension to the nodal formulation, to facilitate inclusion of general circuit elements. This latter formulation method has a disadvantage in that it suffers from zeroes in the diagonal of the coefficient matrix. By means of pivoting, this problem can be circumvented, but this yields a coefficient matrix with a less dominant diagonal than the original matrix.

The equation solving method in conjunction with the network formulation method determines how the different matrix entries are addressed (sequence and locality). The Jacobi and Gauss-Seidel iterative methods allow each matrix row to be treated as a separate equation and only require node voltages to be communicated. With these algorithms the matrix equation can be partitioned such that each processor solves for a number of rows, and the results from each processor are then iterated with neighboring processors. Because of convergence properties, the Jacobi method is used for internode iterations and the Gauss-Seidel method for intranode iterations.

The algorithms used for evaluating the matrix equation coefficients are not difficult to implement in a concurrent manner by letting each node compute a subset of the coefficients. The only penalty for this is some redundancy in terms of stored data (voltages and device data), and that parts of some device equations are evaluated in more than one node. The amount of redundancy depends on the network formulation and equation solving methods being used, as well as on the partitioning scheme.

With the above-mentioned partitioning scheme, the integration and linearization algorithms only have to operate on coefficients within the row being solved for. Furthermore, the Newton-Raphson algorithm, in conjunction with the Jacobi or Gauss-Seidel method, can be implemented in a straightforward manner without the need for matrix inversion. Similarly the variable-step variable-order backward differentiation

predictor-corrector formula can be the same as for any sequential program, except that each node may be integrated separately. In order to preserve charge, each device terminal should be associated with a state vector, allowing the charge state variable to be integrated rather than the node voltage. This requires some extra storage but simplifies device modeling, since the derivatives of the charge state variable can be discontinuous without causing nonconservation of charge.

CONCISE uses the nodal formulation, despite its nongenerality, to formulate the network equations because it can be mapped on a multiprocessor system without difficulties. The resulting matrix equation can easily be partitioned with any number of rows in each processor. Furthermore there is no need for pivoting and thus the tendency towards diagonal dominance and the connectivity can be preserved. It is very important to have a matrix with a dominating diagonal if an iterative method is used for solving the matrix equation.

With a partitioned matrix, direct equation solving methods become expensive since they address the coefficients in a way that is difficult to distribute without excessive communication. Some iterative methods require only operations within a row at a time and are thus easy to distribute. CONCISE use the Jacobi and Gauss-Seidel methods as described in the previous section. This allows each row, or chunk of rows to be solved for concurrently with rows in other processors. When each row equation has been solved, the intraprocessor iteration, and interprocessor iteration can be performed to check convergence among the rows. Intraprocessor and interprocessor iterations are repeated until sufficient convergence is achieved.

When rows are solved with an iterative method requiring operations within the row only, whole waveforms can be solved and iterated instead of voltages. Solving for waveforms is advantageous when each row equation is solved independently, since the time step and order for the integration algorithm can be optimized for each node. That is, each waveform can be solved with a minimum of computational effort. Hence CONCISE uses waveform iterations to solve for the node voltages.

When the node voltages are integrated, a discontinuity at a certain time point may cause the integration algorithm to try to "home in" on a time point close to the discontinuity. For example, if a square wave is applied to a network, the error control algorithm causes steps past the edge of the input signal to be rejected, since the error for these steps is too large. A new shorter step is tried from the last accepted time point. If this falls short of the edge it is accepted, otherwise it is rejected and the try is repeated with a yet shorter step. This rejecting and accepting of steps continues until a time point close to the discontinuity has been found and until the time step is "sufficiently short". Since "sufficiently short" typically means at least an order of magnitude shorter, this "homing in" becomes expensive.

CONCISE uses a windowing scheme synchronized with the input signal events. Instead of solving for a whole waveform, the simulation interval is split into windows with each window being a time interval with continuous input signal. Thus the "homing in" behavior of the integration algorithm is eliminated. Furthermore the number of Jacobi iterations needed for solving the matrix equation can be optimized for each window, hence only the "difficult parts" of the waveform are iterated more times.

In the procedures implementing the Newton-Raphson algorithm, special care has been taken to make use of the fact that only one row equation is solved at a time. Hence each device model only has to return the value and derivative of the terminal current and terminal charge for terminals associated with the row equation being solved. But as MOS devices have four terminals, it is difficult to compute derivatives analytically when more than one terminal is connected to the same node. However it is very easy to compute the derivative numerically, as the derivative is only with respect to one voltage. For conventional programs it is not simple to compute the derivatives numerically, but it is not necessary to find parallel terminal connections as everything is plugged into the same coefficient matrix. CONCISE uses the forward difference approximation to find the derivatives with respect to the node voltage. This requires two function evaluations to find the derivative, and to minimize the number of function evaluations, the derivative is not updated each Newton-Raphson iteration; the accuracy of the solution is determined by the accuracy of the current and charge formulations and not the derivative.

Charge conservation is important in MOS circuits. Hence a state vector is stored for each device and a derivation of the state variable (e.g., charge) is done instead of multiplying the terminal capacitance with the derivative operator. As well as preserving charge, this scheme makes modeling easier since discontinuous capacitors do not cause any problems (as long as the charge is piecewise linear). The disadvantage is that each state variable has to be derived separately.

A sequential version of CONCISE has been implemented on a VAX to facilitate some experiments prior to moving the program over to the Cosmic Cube.

In this sequential version of CONCISE all algorithms needed to perform a transient analysis are included. Only the outer iterations of the matrix equation solving routine, corresponding to internode iteration, are unused. The interprocessor iteration routine essentially consists of a message passing routine asking for waveforms needed and distributing new solutions from the intraprocessor iterations. Thus the sequential and concurrent versions of CONCISE are essentially identical.

It is tempting to compare with SPICE2, but since CONCISE is implemented in another programming language and intended for parallel processing such a comparison is difficult. On the other hand the question of how CONCISE is doing compared to SPICE2 is bound to come. Thus comparisons on circuits ranging from MOS inverters to dynamic NMOS circuits with bootstrapping have been done.

When running SPICE2 it is important to be aware of the fact that the run time depends on which MOS model and what model parameters are used. For a "hot clocked" pad driver run times with SPICE2 were in the range 5-30 minutes. For the same circuit CONCISE used 2-45 minutes depending on the allowed truncation error. The MOS model used in CONCISE has the same accuracy, or better, than SPICE2 levels 2 and 3. When comparing with MOS model level 1 in SPICE2, and using a truncation error in CONCISE giving the same level of detail in the output waveforms for the two programs, the run times were roughly equivalent.

Other circuits were used with CONCISE to establish whether the run time scaled linearly with circuit complexity.

With totally decoupled circuits, i.e., only voltage sources in common, run time scaled perfectly linearly with size, as expected from theory. Circuit size varied from 2 to 200 MOS transistors.

With various circuits it was established that the windowing scheme used in the waveform iteration was very effective. Run times decreased up to five times when windowing was applied as opposed to when the whole waveform was iterated. Furthermore, for none of the circuits were the average number of window iterations, intraprocessor iterations, greater than four. This is a very important result since the Jacobi method is one of the slowest ways of iteratively solving a matrix equation. Thus the fact that only 3-4 iterations per window are needed is very encouraging, since it proves that the waveform convergence is good for realistic circuits.

Profiling was run on most of the test circuits. With the complex MOS model, 25-50% of the total time was spent doing square roots in the device equations. Happily, square roots are one thing that the cosmic cube nodes do faster than a VAX. Also by using simpler models, the run times decreased correspondingly. This shows that most of the time is spent solving device equations and the overhead in doing it is small; it pays off to use simple models., Even with an extremely simple MOS model, much simpler than SPICE2 level 1, most of the CPU time was used in evaluating the device response.

After some additional measurements and profiling of the communication demands in the internode iterations, our next step is to bring CONCISE up on the cosmic cube, a task that is expected to be routine.

(Sven Mattisson is a special computer science graduate student at Caltech, and also with the Applied Electronics Department, Lund Institute of Technology, Sweden.)

3.2 Process Placement

Craig Steele, Chuck Seitz

This effort is directed at optimization of process placement on distributed homogeneous processor architectures. In a such an environment, a computation may be represented as a graph, where communicating processes are the vertices, and logical communication channels are the edges. Likewise the underlying physical structure may be represented as a graph with processors and physical communication paths represented as vertices and edges. To run a computation on a distributed processor, the processes must be loaded onto the physical machine, requiring a mapping of the logical graph onto the physical graph.

Minimizing the communication cost of the computation is a major problem unless the logical and physical graphs are similar. While many problems of interest to physicists can take advantage of the isomorphism of the 6-cube architecture of the Cube to three-space, even in three-dimensional simulations the density of simulated objects may vary, lowering efficiency with simple partitioning. Graph structures common in applications of interest in computer science, such as trees, do not map in obviously good ways to n-cube architectures.

Using the technique of simulated annealing, good mappings may be found in moderate time for arbitrary logical computation graphs. Taken into account are arbitrary edge weights (reflecting varied utilization of logical communication channels) and arbitrary process memory sizes (allowing the physical processor constraints to affect the density of processes per processor).

This method has been applied to a comparative study of proposed interconnection structures for homogeneous machines for both single and multiple process to processor mappings. Efficient mappings are found for interconnects less costly than binary n-cubes for most problems at only modestly increased communication costs. For example, a modified shuffle-exchange interconnect for the 1024-element mosaic multiprocessor design has been simulated for equally large problems to good effect.

Work is continuing to extend the architectural comparison to include congestion factors for networks in which communication channel saturation is significant.

3.3 Techniques in Program Verification

Young-il Choo, Jim Kajiya

3.3.1 A Simple Proof Technique for Abstractions

To verify large programs it is imperative that the formal technique includes rules for abstraction constructs like procedures and functions. For simple procedures and functions without recursion the substitution of the body for the procedure or function call may be acceptable, but for procedures and functions with recursion there has to be more sophisticated techniques.

As we surveyed different proposed proof rules, we found that they fall into two fundamentally different styles. In the one we have Hoare logic-like systems with the assignment statement as the primitive operation (see [Reynolds 81]). Here, recursive procedures can be handled by induction on the proof steps but recursive functions cannot be dealt with in a logically sound and elegant manner (see [O'Donnell 82]). On the other we have proof systems for recursive functions where structural induction and computational (Scott's) induction are the main techniques (see [Manna 75]).

When one considers the logical complexity of a (mathematical) function, the proof of correctness should be similar whether it is programmed as a procedure in an imperative language or a function in an applicative language. By comparing the proofs we found a compelling similarity in that they both used induction in one way or another. When we tried to translate the proof of one into the proof of the other there were significant difficulties. We found it impossible to translate functional proofs into procedural ones. In order to translate the procedural one to a functional one what we had to do was give an explicit denotational semantics to the

assignment statement as a function from the set of states to the set of states. Then the proof is identical except for the fact that the domains of our objects are different.

Essentially, our method is based on a denotational semantics which interprets a recursive definition of a function or a procedure as specifying the unique object satisfying the equation. Given that the object exists, the recursive definition is the defining equation characterizing the object and our proof technique reduces to that of doing mathematical induction over a well founded set that is defined for each domain.

Our experience with this technique indicates that the proofs become a series of equalities that are algebraically manipulated and therefore are not too sensitive to the particular syntax of the language or of the logic. In fact the logic we use is our intuitive notions of reasoning used in mathematics. Therefore we do not need to set up a logical theory with peculiar formulas like in the Hoare system.

A very interesting consequence of this approach is the possibility of proving Prolog programs constructed out of Horn clauses. Here again the denotational semantics provides us with objects which are in some sense solutions of the recursive equations. The interesting part here is in coming up with a nice denotational semantics for Horn clauses.

3.3.2 Characterization of Admissible Predicates

By using lazy evaluation it is possible to specify infinite data objects and operate on them to produce other infinite data objects. In general, infinite functions can be defined recursively that produce meaningful results but which do not terminate. The fundamental data objects are finite or infinite lists whose elements can be finite or infinite lists. Before one can formulate proof techniques there must be a clearly defined semantics. Using the Scott D-infinity construction method we have constructed the appropriate domain for our data objects.

The proof technique to use would appear to be the Scott induction rule. This rule is very powerful in allowing us to reason about infinite objects from the properties of their finite approximations. When using Scott's induction we must make sure that the properties (expressed as predicates in some logical language) are admissible, for only admissible predicates give sound conclusions.

When we are dealing with infinite data objects, it is not always clear what properties are admissible. In the literature there are sufficient characterizations for a predicate to be admissible [Manna 74], but none showing the necessary conditions.

Since we are dealing with infinite objects, and because certain properties holding for finite ones do not seem to extend for infinite ones, we are looking at non-standard analyses where standard and non-standard models are defined and where finite and infinite objects are distinguished. Our aim is to prove the necessary and sufficient conditions for the admissibility of predicates by constructing a suitable standard and non-standard models and applying the Concurrence Theorem and notions such as internal and external sets.

References

[Manna 74] Manna, Zohar, *Mathematical Theory of Computation*, McGraw-Hill, 1974.

[O'Donnell 82] O'Donnell, Michael, "A Critique of the Foundations of Hoare Style Programming Logics", *CACM* 25 No 12, December 1982.

[Reynolds 81] Reynolds, John C, *The Craft of Programming*, Prentice-Hall, 1981.

3.4 A New Network – Sneptrees for Distributed Computation

Pey-yun Peggy Li, Alain J. Martin

A new interconnection network, the Sneptree, has been investigated in the past few months. It is an augmented binary tree configuration with homogeneous nodes. Each node has four links. The extra links

are so connected that the outgoing links of the leaf nodes are fed back to all the nodes of the binary tree. There are many such connections. One type of connection which contains two spanning cycles is of particular interest. We name this special type of Sneptree a "Cyclic Sneptree", and focus on this connection.

It has been proved that the mapping of a complete binary tree of any size onto a Sneptree is optimal. Moreover, an extremely unbalanced tree, such as a left/right skewed tree, results in an optimal mapping on a cyclic Sneptree.

The cyclic Sneptree can be laid out onto an H-structure plane. The construction rule is recursively defined so that the bigger H-structure Sneptree can be constructed by connecting smaller Sneptrees following some simple rules. Moreover, the cyclic Sneptree can be extended easily with smaller Sneptrees built in single chips. The cyclic Sneptree is not planar, neither is its H-structure layout. The number of crossings in the H-structure layout is about 3/8 of the total number of nodes in the Sneptree. The maximal length of the extra links are about the same as that of the longest wire in the H-structure binary tree layout.

The cyclic Sneptree can simulate a binary tree optimally. The mapping of other data structures onto the Sneptree, such as linear array, has also been considered. Having investigated many applications, we found it better to map the linear array onto the leaf nodes and to use a routing algorithm from a leaf node to leaf node. The routing time is bounded in n, where n is the height of the Sneptree. The route selected is shorter and less congested than on a pure binary tree.

Applications for the Sneptree are under investigation. One popular knowledge representation method in AI applications, the semantic net, has been mapped onto the Sneptree. It works well for some special examples. How to map an arbitrary Semantic Net to the Sneptree and achieve the maximal concurrency in searching the Net is still an open question.

3.5 Functional Programming

Jerry Burch, Young-il Choo, Alain J Martin

We have started investigating functional programming methods. After experimenting with various functional programming styles, we have decided to choose a language with lexical scoping and lazy evaluation as a rule, thus allowing infinite lists. In a first instance, we want ot find simple semantics and proof techniques that are able to deal with lazy evaluation and infinite lists. The various forms of fixed point induction techniques currently available are not sufficient for that purpose. Our long-term goal is to find out whether such languages are suitable for highly concurrent and distributed implementatons.

3.6 Concurrent Data Structures

Bill Dally, Chuck Seitz

The appearance of concurrent computers such as the Cosmic Cube and Mosaic is creating a need for concurrent data structures. Conventional data structures such as heaps and B-trees have many bottlenecks which limit their potential concurrency and make them unable to take advantage of the computing potential of these concurrent machines. New data structures are required that can harness the power of concurrent computing.

Our research has been proceeding in two directions: to study the concurrency limitations of existing data structures and to develop new data structures which overcome these limitations. A heap was selected as representative of existing data structures and its concurrency properties were studied in detail [5156:DF:84]. As a result of this research some variants of conventional heaps were developed which offer improved concurrency and virtual memory performance. However, even these variant heaps are limited by their tree structure to concurrency which grows only as the log of the number of elements in the heap.

To overcome the concurrency limitations of conventional data structures we have developed two new data structures for ordered sets: the balanced cube and the B-cube [5159:DF:84]. By using a Boolean n-cube

rather than a tree to organize data, these structures overcome the bottleneck of a single entry point or root node. As a result, the concurrency of these cube structures grows linearly with the number of elements in the set enabling them to effectively make use of concurrent processors.

3.7 ANIMAC: A Multi-Processor Animation Machine

Daniel S Whelan, Jim Kajiya

This thesis work proposes a new architecture for animation and flight simulation engines. This architecture consists of a nearest neighbor network of specialized processors. The targeted performance of 100,000 polygons per frame time represents an improvement of almost two orders of magnitude over today's systems. Furthermore, scenes are calculated with shadowing effects. Anti-aliasing techniques are utilized to achieve acceptable image quality.

Development of the ANIMAC architecture involved the design of a partitionable visible surface determination algorithm. This algorithm allows individual engines to be connected into a multi-processor and still utilize only very local information for calculating shadowed images. In order to achieve load balancing in the ANIMAC, both static and dynamic techniques for associating processors with image space regions were examined.

The design of the individual engines involved the extension of the processor per object paradigm to handle shadowing of objects. A processor per pixel algorithm is also utilized in determining whether non-local objects cast shadows on local objects.

This work rests on a model of scene composition that has been determined by analyzing statistics gathered from images created by a rendering program. This program is in wide use by the Caltech graphics community. Selected images were meant to be representative of scenes encountered in animation runs.

Previous work in the field has been re-examined in light of the scene composition model. Strong points of these architectures have found their way into ANIMAC. Hopefully, the weak points have been avoided.

3.8 Language Research

Mike Newton, Howard Derby, Jim Kajiya

The language investigation of the last few years into rewrite and logic programming systems is finally entering the implementation phase. With Mike Newton and Howard Derby, the past 6 months have seen the construction of a preliminary design and implementation of a logic programming language which allows functional programming constructs.

We do this by modifying the standard prolog interpreter to include a rewrite subsystem to be invoked upon the failure of ordinary unification. If ordinary unification fails, the rewrite subsystem attempts a bottom up rewrite of terms in the goal. These rewrites are fully trailed so that backtracking will explore all alternatives.

Several small programs have been written in the new language. Indications are that its possesses a much greater expressive power than standard Prolog. We estimate something like a compression ratio of 4 to 10 times.

During the course of this investigation we have invented a new method for controlling backtracking which is in a sense dual to Prolog's cut mechanism. It appears to be natural from a programmers point of view and relatively easy to compile. A report is being written now and will appear shortly.

Future language work involves studying the compilation issue for our functional logic language. At present we have a rather inefficient interpreter. There is every indication that the techniques used to compile Prolog and Lisp will be applicable to this language. Second, we wish to extend the data structures which can be

manipulated by the language–currently limited to trees and lists. Finally, the impact of this language on machine organization is undergoing active investigation.

3.9 An Object Oriented Computer Architecture

Jim Kajiya, Bill Dally

With Bill Dally, we have been exploring new architectures for late binding object oriented languages, most notably Smalltalk. These promise a potential 100 times speed up over conventional instruction set processors implemented in an equivalent technology. We propose to investigate these ideas with experiments on a modified Smalltalk-80 implementation.

The architecture is based on a conventional pipelined von Neumann machine with two extensions: associative threading, and floating point addresses. About one third of the time is spent hashing messages in an object oriented interpretation loop . The associative threading scheme effectively pipelines this overhead to a rather small value. Another one third of the time is spent in storage management: mostly in the allocation and deallocation of contexts. Our architecture minimizes this overhead with segmented memory.

Associative threading is a key to fast object oriented machines. Conventional instruction set processors use operation codes whose interpretation is independent of operand datatypes. In associative threading the meaning of an instruction is looked up in an association table whose key is formed from the opcode and the datatypes of operands.

The association mechanism is implemented much like a conventional demand paging virtual memory system. First, the machine maintains a complete set of association tables. These are simply the message dictionaries for each class. Second, the machine caches most recently used associations in a local memory called the instruction translation lookaside buffer (ITLB).

The overhead of storage management, bounds checking, and data security is minimized by hardware segmented memory – one object per memory segment. But with conventional segmented memory schemes, the "small object problem" immediately arises. A memory address must be able to specify a segment as well as an offset to select a field of a given segment. To accomodate a large number of small objects, the segment index field of an address must be large. But then for a reasonable address wordsize, the number of bits which can be devoted to a segment offset is uncomfortably small. Our addressing scheme handles this problem by adding a binary exponent. The exponent determines the boundary between segment and offset fields. The integer part of the resulting number forms the segment index while the fractional part forms the offset. This scheme allows for both a large number of small objects as well as a few very large objects.

Our preliminary simulations with an ITLB are very encouraging. With a 4 way set associative cache of modest size (8K), we measure hit ratios in the high 90 architecture we estimate that a machine based on conventional TTL technology will be able to execute perhaps 5 to 10 million 3 address instructions per second. A single 3 address instruction would be equivalent 2-3 Smalltalk bytecodes.

4. VLSI Design

4.1 Switch Simulation Tools

Bill Dally, Randy Bryant

Much of the effort in switch level simulation tools in this period is concentrated on the MOSSIM Simulation Engine (MSE). A paper on the MSE is included as an appendix to this report. A paper by Randy Bryant: "A Switch-Level Model and Simulator for MOS Digital Circuits" appeared in the IEEE Transactions on Computers, February, 1984.

Design verification is essential in the development of VLSI systems. The complexity of VLSI circuits, inaccessability of internal nodes, and difficulty of repair make the probability of producing a working chip very low without extensive design verification. As the complexity of VLSI circuits approaches 10^6 devices, the computational requirements of design verification are exceeding the capacity of general purpose computers. To provide the computing power required to verify these complex VLSI chips, we are developing the Mossim Simulation Engine (MSE) [5123:TR:84].

The Mossim Simulation Engine is a special purpose processor which, in a single processor configuration, performs switch-level simulation of MOS VLSI circuits 200-500 times faster than a VAX 11/780. In multiple processor configurations even greater speedup can be achieved.

The MSE overcomes two limitations of existing simulation engines. By using the switch-level model developed by Bryant [5065:TR:83], the MSE performs accurate simulation of MOS circuits. Existing simulation engines perform logic simulation and cannot model MOS effects such as stored charge, charge sharing and transistor ratios. Also, by using the concept of virtual processors the MSE can simulate a circuit many times larger than the size of the processor. Existing simulation engines are limited to simulating circuits which fit in the processor.

A prototype MSE has been constructed and is now in the final stages of debugging. A debug monitor and a microassembler have been written to support this hardware development. The prototype will be used both as a research tool to support the simulation requirements of the next generation of VLSI circuits and as a test bed for experiments in switch-level simulation including: the development of a virtual simulation processor system, experiments in the application of multi-processing to switch-level simulation, and a study of the locality of activity in MOS circuits.

4.2 Symbolic Manipulation of Boolean Functions

Randy Bryant

Many problems in computer science and engineering can be described in terms of symbolic manipulations of Boolean functions. For example, to verify that a combinational logic network correctly implements a desired behavior, we could construct symbolic representations of the Boolean functions describing the network and the desired behavior, and then test the two for equivalence. This, of course, is an NP-complete problem, but our hope is to develop algorithms which under most conditions do not exhibit exponential behavior. Other problems for which this approach may yield practical results include automatic test generation (described below), and certain combinatorial problems such as graph coloring.

We have developed a new set of algorithms for Boolean function manipulation whereby the function is represented by an acyclic directed graph, similar to the binary decision diagram notation introduced by Lee [1] and further popularized by Akers[2]. While the graph representing a function of n arguments can have $O(2^n)$ vertices in the worst case, most commonly encountered functions are represented by smaller graphs. We have developed algorithms for performing various operations on Boolean functions using this representation, where the complexity is related to the size of the graphs being operated on. Hence, as long

as the graphs do not grow too large, these algorithms have good performance. We are currently running experiments to determine the practicality of our approach.

4.3 Automatic Test Pattern Generation

Randy Bryant

We have developed a new method to generate test patterns that detect faults in combinational logic circuits based on Boolean function manipulation. We construct a symbolic representation of the functions computed by the good circuit and by each faulty circuit. We then attempt to find a small set of input patterns such that for each faulty circuit at least one of these patterns will yield a different value on some output than in the good circuit. Similar techniques were used over 20 years ago at IBM but failed to be practical for larger circuits due to the use of inefficient algorithms for Boolean function manipulation.

Some advantages of our approach over more traditional test generation programs (such as those based on the D-algorithm) include: (1) general fault models – anything that can be represented by Boolean functions, (2) easy recognition of undetectable faults – when both the good and the faulty circuit implement the same function, (3) guaranteed fault coverage – 100% of all detectable faults modeled, (4) less sensitivity to the type of circuit – fewer problems with exclusive-or's and reconvergent fanout, and (5) smaller test sets.

Preliminary experiments with this approach have been quite promising. Using the 74181 4-bit ALU chip as a benchmark (MSI technology, 14 inputs, 75 logic gates, 204 single faults), we have generated several test sets containing only 12 patterns that detect all single faults. It has been shown that no smaller test set can give complete fault coverage for this circuit, hence our tests are minimal. To our knowledge, only hand generated test sets have matched this result. Our program generates these test sets with less than 20 minutes of CPU time on a VAX 11/780 – an acceptable performance for this task.

We plan to experiment with both larger and different classes of circuits to evaluate this approach more completely. Remaining areas of research include extending these methods to sequential and to switch-level circuits.

References

Lee, C.Y., "Representation of Switching Circuits by Binary Decision Programs", *BSTJ*, July, 1959.

Akers, S.B., "Binary Decision Diagrams", *IEEE TC*, June, 1978.

4.4 Bit-Serial Reed-Solomon Decoders in VLSI

Douglas L Whiting, Robert J McEliece

Reed-Solomon codes are among the most versatile and powerful error-correcting codes available, with an inherent capability of correcting both random and burst errors and of incorporating a limited amount of soft decision information. Thus, Reed-Solomon (RS) codes are a prime candidate for VLSI implementation. Efficient RS encoder design is relatively well understood, but the problem of optimal decoder architectures for VLSI remains an open question. Conventional RS decoders have been implemented with sequential architectures, but our aim was to produce decoder architectures that take advantage of the parallelism available in the decoding algorithms. Original results have been obtained in several key areas which can be combined to produce efficient decoder designs. These results, presented in detail in the thesis "Bit-Serial Reed-Solomon Decoders in VLSI" by Douglas L Whiting, represent a significant contribution to the techniques available for design and implementation of Reed-Solomon decoders.

First, efficient structures for performing finite-field arithmetic in digital logic are examined. A general expression for the transformations involved in bit-serial multiplication is derived. This expression includes Berlekamp's dual basis multiplier and Omura's normal basis multiplier as special cases, and it is shown that

the dual basis multiplier is not restricted to multiplication by a constant. Next, the necessary and sufficient condition is derived for the dual basis to be identical to the canonical basis, which consists of consecutive powers of a primitive element. Also, it is shown how reciprocals over the field can be computed in an efficient bit-serial fashion; traditionally multiplicative inversion has been considered quite costly. We also present a proof that normal basis multiplication (proposed by Omura and Massey) involves roughly twice the number of product terms as a dual basis multiplier.

In the area of decoding methods, known decoding algorithms for Reed-Solomon codes are reviewed and compared. New techniques for erasure initialization in the Berlekamp and Euclidean decoding algorithms are presented which fit much more naturally into hardware than previous approaches.

Using these results as a foundation, we introduce several decoder architectures that utilize as much parallelism as possible and are suitable for VLSI implementation. In particular, for a t error correcting code over $GF(2^m)$, these architectures have area $A = O(mt)$ and pipeline period $P = O(1)$. Our decoders have the property that the throughput in bits per second is directly related to P ; *ie*, a 10 MHz clock rate implies a 10 Mbit/sec decoder throughput. Further, it is shown that a single decoder chip can handle a variety of redundancies and block lengths with little additional area overhead. Among these architectures is an array of cells to solve the key equation which requires only a single central controller yet is also entirely systolic. This structure can also be used for encoding and seems to have several practical advantages over Kung's proposed systolic array.

In the area of burst modelling, we developed a method of computing exact output bit error probabilities using the Gilbert channel model for Reed-Solomon codewords of a given blocklength, redundancy, and depth of interleaving. Previous attempts at such computations have assumed that all character errors are independent, but our method allows direct modelling of bursty channels by introducing an extra degree of freedom.

4.5 Soft Error Correction for Increased Densities in VLSI Memories

Robert J McEliece, Khaled Abdel-Ghaffar, Henk Van Tilborg

If VLSI RAM densities are to continue to increase, it will be necessary to take the problems associated with "soft errors" much more seriously than has previously been done. We have undertaken a serious study of how on-chip error-correcting codes can be used to enhance the reliability of RAM chips.

We have analyzed the limits of code performance as feature size increases indefinitely, and our results have been published in an invited paper entitled "Soft Error Correction for Increased Densities in VLSI Memories," in the proceedings of the IEEE 1984 Conference on Computer Architecture. A copy of this paper is reproduced in the appendix to this report.

Also, we have written an expository article called "The Reliability of Computer Memories," which will appear in the January 1984 issue of Scientific American.

Our research indicates that the most important source of VLSI soft errors in the near future will be alpha particle-induced errors. As cell dimensions decrease, we find that a single alpha particle may cause a two-dimensional "burst" of errors. Thus our current research efforts have been focused on the problem of efficient correction of such bursts, a subject which has received very little previous attention in the coding literature. We have devised a new class of two-dimensional burst-error correcting codes, based on the new idea "burst identification," which are very easy to implement (*ie*, the encoding and decoding circuits would occupy only a tiny fraction of the RAM's area), and which are of very low redundancy. A preliminary report on this work was presented at the 1984 IEEE Information Theory workshop, and a fuller report is now in preparation.

2nd draft, for December 1984 issue of CACM

The Cosmic Cube

Charles L Seitz

Introduction

The Cosmic Cube is an experimental computer for exploring the practice of highly concurrent computing. The largest of several Cosmic Cubes currently in use at Caltech consists of 64 small computers that work concurrently on parts of a larger problem, and coordinate their computations by sending messages to each other. In analogy to a computer network, we refer to the individual small computers as *nodes*. Each node is connected through bidirectional, asynchronous, point-to-point communication channels to six other nodes to form a communication network that follows the plan of a 6-dimensional hypercube, or binary 6-cube (Fig 1). An operating system kernel in each node schedules and runs processes within that node, provides system calls for processes to send and receive messages, and routes the messages that flow through the node.

The excellent performance exhibited by the Cosmic Cube on a variety of complex and demanding applications, together with its modest cost and open-ended expandability, suggests that highly concurrent systems of this type are an effective means of achieving faster and less expensive computing in the near future. The Cosmic Cube nodes were designed as a simulation in hardware of the nodes we expect to be able to integrate onto one or two chips about five years hence. Future machines of thousands of nodes are feasible, and for many demanding computing problems, these machines are expected to outperform the fastest uniprocessor systems. Even with current microelectronic technology, the 64-node machine is quite powerful for its cost and size. It benchmarks between five and ten times the speed of a VAX11/780 on a variety of demanding scientific and engineering computations.

Message Passing Architecture

A significant difference between the Cosmic Cube and most other parallel processors is that this multiple instruction, multiple data (MIMD) machine uses *message passing*, while most such machines use shared variables for communication between concurrent processes. This message passing computational model is reflected in the hardware structure and operating system, and is also the explicit communication and synchronization primitive seen by the programmer.

The hardware structure of a message passing machine such as the Cosmic Cube differs from a shared storage multiprocessor by employing no switching network between processors and storage (Fig 2). The advantages of this message passing architecture derive from a separation of engineering concerns between the processor-storage communication and the interprocess communication. The critical path in the communication between an instruction processor and its random-access storage, the so-called von Neumann bottleneck, can be engineered to exhibit a much smaller latency when the processor and storage are physically localized. The processor and storage might occupy a single chip, hybrid package, or circuit board, depending on the technology and complexity of the node.

It was a premise of the Cosmic Cube experiment that the internode communication scale well to very large numbers of nodes. A *direct* network such as the hypercube satisfies this requirement, both with respect to the aggregate bandwidth achieved across the many concurrent communication channels, and also with respect to the feasibility of the implementation. The hypercube is actually the same network in a distributed form as *indirect* logarithmic switching networks, such as the

The research described in this paper was sponsored by the Defense Advanced Research Projects Agency, ARPA Order number 3771, and monitored by the Office of Naval Research under contract number N00014-79-C-0597.

Omega or banyan network, that might be used in shared storage organizations. The hypercube has the additional property that communication paths traverse different numbers of channels, and so exhibit different latencies. One may take advantage of communication locality in the placement of processes in nodes.

Message passing machines are simpler and more economical than shared storage machines, and increasing so with larger numbers of processors. However, the more tightly-coupled shared storage machine is more versatile, since it is able to support code and data sharing. Indeed, shared storage machines can simulate message passing primitives very easily, while message passing machines do not efficiently support code sharing and shared variables.

Figure 2 emphasizes the differences between shared storage and message passing organizations by picturing the extreme points. We conjecture that shared storage organizations will be preferred for systems with tens of processors, and message passing organizations for systems with hundreds or thousands of processing nodes. Hybrid forms employing local storage or cache with each processor, together with a message passing approach to non-local storage references and cache coherence, may well prove to be most attractive for intermediate numbers of processors.

Process Programming

The hardware structure of the Cosmic Cube, when viewed at the level of nodes and channels, is a difficult target for programming any but highly regular computing problems. It is the resident operating system of the Cosmic Cube that supports a more flexible and machine-independent environment for concurrent computations. This process model of computation is quite similar to the hardware structure of the Cosmic Cube, but is usefully abstracted from it. Instead of formulating a problem to fit on nodes and on the physical communication channels that exist only between certain pairs of nodes, one may formulate a problem in terms of processes and "virtual" communication channels between processes.

The basic unit of these computations is the *process*, which for our purposes is an instance of a sequential program that contains actions of sending and receiving messages. A single node may contain many processes. All processes execute concurrently, whether by virtue of being in different nodes, or by being interleaved in execution within a single node. Each process has a unique (global) ID that serves as an address for sending it messages. All messages have headers containing the destination and sender IDs, and a message type and length. Messages are queued in transit, but between any pair of processes, message order is preserved. The semantics of the message passing operations are independent of the placement of processes in nodes.

Except for the ability to distribute processes across the nodes, this process programming enviroment with interprocess communication by messages is common to many multiprogramming operating systems. A copy of the resident operating system of the Cosmic Cube, called the "kernel", resides in each node, and all of these copies are concurrently executable. The kernel can spawn and kill processes within its own node, schedule their execution, spy on them through a debug process, manage storage, and deal with error conditions. In addition, the kernel handles the queueing and routing of messages for processes in its node, as well as for messages that may pass through the node. Many of the functions that we would expect to be done in hardware in a future integrated node, such as message routing, are in the Cosmic Cube done in the kernel. Thus we are able to experiment in the kernel with different algorithms and implementations of low level node functions.

The Cosmic Cube has no special programming notation. Process code is written in ordinary sequential programming languages (*eg* Pascal, C, etc), extended with statements or external procedures to control the sending and receiving of messages. These programs are compiled on other

computers, and are loaded into and relocated within a node as binary code, data, and stack segments.

Process Distribution

It was a deliberate decision in the design of the present kernel that once a process is instantiated in a node, the kernel will not relocate it to another node. One consequence of this restriction is that the physical node number can be included in the ID for a process, thus eliminating the awkward way in which a distributed map from processes to nodes scales with the number of nodes. Messages are routed according to the physical address part of the destination process ID in the message header.

This decision was also consistent with the notion that the programmer be able to control the way in which the processes are distributed onto the nodes, based on an understanding of the structure of the concurrent computation being performed. Alternatively, since it is only the efficiency of a multiple process program that is influenced by process placement, the choice of the node in which a process is spawned can be deferred to a library process that makes this assignment based on inquiries about the processing load and storage utilization in nearby nodes.

The objective of a careful distribution of processes to nodes generally involves some tradeoffs between load balancing and message locality. We use the term *process structure* to describe a set of processes and their references to other processes. A static process structure, or a snapshot of a dynamic process structure, can be represented as a graph of process vertices connected by arcs that represent reference (Fig 3). One can also think of the arcs as virtual communication channels, in that process A having reference to process B is what makes a message from A to B possible.

The hardware communication structure of this class of message passing machines can be represented similarly as a graph of vertices for nodes and (undirected) edges for the bidirectional communication channels. The mapping of a process structure onto a machine is an embedding of the process structure graph into the machine graph (Fig 5). In general, the arcs map not only to internal communication and single edges, but also to paths, representing the routing of messages in intermediate nodes. It is this embedding that determines both the locality of communication achieved and the load balancing properties of the mapping.

Concurrency Approach

Most sequential processors, including microprocessors such as the RISC chips described elsewhere in this special issue, are *covertly* concurrent machines that speed up the interpretation of a single instruction stream by techniques such as instruction prefetching and execution pipelining. Compilers can assist in this speedup by recovering the concurrency in expression evaluations and in the innermost iterations of a program, and then generating code that is "vectorized", or in some other respects allows the processor to interpret the sequential program with some concurrency. These techniques, together with caching, allow about a 10-fold concurrency and speedup over naïve sequential interpretation.

We can use such techniques within nodes, where we are tied to sequential program representations of the processes. In addition, we want to have at least as many concurrent processes as nodes. Where are such large degrees of concurrency to be discovered in a computation? One quick but not quite accurate way of describing the approach used in the Cosmic Cube is that we exploit *overtly* the concurrency found in the outermost, rather than innermost, program constructs of certain demanding computations. It appears that many highly demanding computing problems can be expressed in terms of concurrent processes with either sparse or predictable interaction. Also,

the degree of concurrency inherent in such problems tends to grow with the size and computing demands of the problem.

It is important to understand that the compilers used to generate process code for the Cosmic Cube do not "automatically" find a way to run sequential programs concurrently. We do not know how to write a program that translates application programs represented by old, dusty Fortran decks into programs that exploit concurrency between nodes. In fact, because efficient concurrent algorithms may be quite different from their sequential counterparts, we regard such a translation as implausible, and instead try to formulate and express a computation explicitly in terms of a collection of communicating concurrent processes.

Data flow graphs, such as those discussed in this issue in the article on the Manchester data flow machine, also allow an explicit representation of concurrency in a computation. Although we have not yet tried to do this, data flow computations can be executed on machines such as the Cosmic Cube. One of the reasons we do not use data flow graphs is that many of the computations that show excellent performance on the Cosmic Cube or on other parallel machines, and are very naturally expressed in terms of processes (or objects), are in the nature of simulations of physical systems. Here the state of a system is repeatedly evaluated and assigned to state variables. The functional (side-effect free) semantics of data flow, in pure form, appears to get in the way of straightforward expression of this type of computation. The process model that we use for programming the Cosmic Cube is relatively less restrictive than data flow, and in our implementation is relatively more demanding of attention to details such as process placement.

Concurrent Formulations

The crucial step in developing an application program for the Cosmic Cube is the concurrent formulation, because it is here that both the correctness and efficiency of the program are determined. It is often intriguing, even amusing, to devise strategies for coordinating in an orderly way a myriad of concurrent computing activities.

For many of the demanding computations encountered in science and engineering, this formulation task has not proved to be very much more difficult than it is for sequential machines. These applications are typically based on concurrent adaptations of well-known sequential algorithms, or are similar to the "systolic" algorithms that have been developed for regular VLSI computational arrays. The process structure remains static for the duration of a computation.

At the risk of leaving the impression that all of the application programs for the Cosmic Cube are as simple, let us offer one concrete example of a formulation and its process code. The problem is to compute the time evolution of a system of N bodies that interact by gravitational attraction, or some other symmetrical force. Because each of the N bodies interacts with all of the $N-1$ other bodies, this problem might not seem to be as appropriate for the Cosmic Cube as matrix, grid point, finite difference, and other problems based solely on local interaction. Actually, universal interaction is easy, because it maps beautifully onto the ring process structure shown for $N = 7$ in Figure 3.

With N odd, each of N identical processes is "host" to one body, and is responsible for computing the forces due to $(N-1)/2$ other bodies. With a symmetrical force, it is left to other processes to compute the other $(N-1)/2$ forces. The process also accumulates the forces and integrates the position of the body it hosts. As shown in the C process code in Figure 4, the process that is host to body 1 successively receives guests 7, 6, and 5, and accumulates forces due to these interactions. Meanwhile, a message containing the position, mass, accumulated force, and host process ID of body 1 is conveyed through the processes that are host to bodies 2, 3, and 4, with the forces due to these interactions accumulated. After $(N-1)/2$ visits the representations

of the bodies are returned in a message to the process that is host to the body, the forces are combined, and the positions updated.

A detail that is not shown in the code in Figure 4 is the process that runs in the Cosmic Cube intermediate host (IH), or on another network-connected machine. This process spawns the processes in the cube, and sends messages to the cube processes that provide the initial state, ID of the next process in the ring, and an integer specifying the number of integration steps to be performed. The computation in the Cosmic Cube can run autonomously for long periods between interactions with the IH process. If some exceptional condition were to occur in the simulation, such as a collision or close encounter, the procedure that computes the forces could report this event with a message back to the IH process.

This ring of processes can, in turn, be embedded systematically into the machine structure (Fig 5). In mapping 7 identical processes, each with the same amount of work to do, on 4 nodes, the load obviously cannot be balanced perfectly. Using a simple performance model originally suggested by Willis Ware, "speedup", S, can be defined as:

$$S = \frac{\text{time on 1 node}}{\text{time on N nodes}}.$$

For this 7-body example on a 4-node machine, neglecting the time required for the communication between nodes, the speedup is clearly 7/2. Since computation proceeds 3.5 times faster using 4 nodes than it would on a single node, one can also say that the efficiency $e = S/N$ is 0.875, representing the fraction of the available cycles that are actually used.

More generally, if k is taken as the fraction of the steps in a computation that, because of dependencies, must be sequential, the time on N nodes is $\max(k, 1/N)$, so that the speedup cannot exceed $\min(1/k, N)$. This expression reduces to "Amdahl's argument", that $1/k$, the reciprocal of the fraction of the computation that must be done sequentially, limits the number of nodes that can usefully be put to work concurrently on a given problem. For example, nothing is gained in this formulation of an N-body problem by using more than N nodes.

Thus we are primarily interested in computations for which $1/k \gg N$, in effect, in computations in which the concurrency opportunities exceed the concurrent resources. Here the speedup obtained by using N nodes concurrently is limited by (1) the idle time due to imperfect load balancing, (2) the waiting time due to the communication latencies in the channels and in the message forwarding, and (3) the processor time dedicated to processing and forwarding messages, a consideration that can be effectively eliminated by architectural improvements in the nodes. These factors are rather complex functions of the formulation, its mapping onto N nodes, the communication latency, and the communication and computing speed of the nodes. We lump these factors into an "overhead" measure σ, defined by the computation exhibiting a speedup $S = N/(1 + \sigma)$. A small σ indicates that the Cosmic Cube is operating with high efficiency, that is, with nodes seldom being idle, or seldom doing work they would not do in the single node version of the computation.

Cosmic Cube Hardware

With this introduction to the architecture, computational model, and concurrent formulations, let us turn now to some experimental results.

Figure 6 is a photograph of the 64-node Cosmic Cube. For such a small machine, only 5 feet long, and large ratio of printed circuit board width to spacing, a one-dimensional projection of the 6-dimensional hypercube is satisfactory. The channels are wired on a backplane underneath the long box in a pattern similar to that shown in Figure 2b. Larger machines would have nodes arrayed in 2 or 3 dimensions, such as the 2-dimensional projection of the channels shown in Figure 1. The

volume of the system is 6 cubic feet, the power consumption is 700 watts, and the manufacturing cost was $80,000. We also operate a 3-cube machine in support of software development, since the 6-cube is not readily shared.

Most of the choices made in this design are fairly easy to explain. First of all, a binary n-cube communication plan was used because this network was shown by simulation to provide very good message flow properties in irregular computations. It also contains all meshes of lower dimension, which is useful for regular mesh-connected problems. The binary n-cube can be viewed recursively. As one can see from studying Figure 1, the n-cube that is used to connect $2^n = N$ nodes is assembled from two $(n-1)$-cubes, with corresponding nodes connected by an additional channel. This property simplifies the packaging of machines of varying size. It also explains some of the excellent message flow properties of the the binary n-cube on irregular problems. The number of channels connecting the pairs of subcubes is proportional to the number of nodes, and hence on average to the amount of message traffic they can generate.

With this rich connection scheme, simulation showed that we could use channels that are fairly slow (about 2 Mbit/sec) compared with the instruction rate. The communication latency is, in fact, deliberately large to make this node more nearly a hardware simulation of the situation anticipated for a single chip node. The processor overhead in dealing with each 64-bit packet is comparable to its latency. The communication channels are asynchronous, full duplex, and include queues for a 64-bit "hardware packet" both in the sender and receiver in each direction, a basic minimum necessary to decouple the sending and receiving processes.

The Intel 8086 was selected as the instruction processor because, at the time, it was the only single chip instruction processor available with a floating point coprocessor, the Intel 8087. Reasonable floating point performance was necessary for many of the applications that our colleagues at Caltech wished to attempt. The system currently operates at a 5 MHz clock rate, limited by the 8087, although it is designed to run at 8 MHz when faster 8087 chips become available. After our first prototypes, Intel Corporation generously donated the chips of their manufacture for the 64-node Cosmic Cube.

The storage size of 128K bytes was the subject of a great deal of internal discussion of "balance" in the design. It was argued that the cost incurred in doubling the storage size would better be spent on more nodes. In fact, this choice is clearly very dependent on target applications and programming style. The dynamic RAM includes parity checking but not error correction. Each node also includes 8K bytes of read-only storage for initialization, bootstrap loader, dynamic RAM refresh, and diagnostic testing programs.

Since machine building is not a very common enterprise in a university, some account of the chronology of the hardware phase of the project may be of interest. A prototype 4-node (2-cube) system on wirewrap boards was designed, assembled, and tested in the winter 1981-82, and this system was used for software development and application programs until it was recently disassembled. The homogeneous structure of these machines was nicely exploited in the project to accelerate the software development by use of a small hardware prototype that is similar to scaled-up machines. Being generally pleased with the results from the 2-cube prototype, we had printed circuit boards designed, and went through the other packaging logistics of assembling a machine of useful size. The Cosmic Cube grew from an 8-node to a 64-node machine over the summer 1983, and has been in routine use since October 1983. The task of building hardware to provide more cycles for the user group has been passed to a group at Caltech's Jet Propulsion Laboratory, with the intention of building a 7-cube and two 5-cubes using a program-compatible derivative design with twice the storage for each node.

In its first year of operation (560,000 node-hours), the Cosmic Cube has experienced two hard failures, both quickly repaired, and a soft error in the RAM is detected by a parity error on average of once every several days.

Cosmic Cube Software

As in many "hardware" projects, most of the effort has been in the software. The software effort has been considerably simplified by the availability of cross-compilers for the Intel 8086/8087 chips, and because most of the software development is done on conventional computers. Programs are not only written and compiled in this familiar computing environment, but their concurrent execution can be simulated on a small scale. Programs are downloaded into the cube through a connection that is managed by the intermediate host. In the interest of revealing all of the operational details of using this unconventional machine, we shall begin with the startup procedures.

The lowest level of software is part of what we will call the *machine intrinsic* environment. This environment includes the instruction set of the node processor, its I/O communication with channels, and a small initialization and bootstrap loader program stored together with diagnostic programs in read-only storage in each processor. A startup packet specifies the size of the cube to be initialized, and may specify that the built-in RAM tests be run (concurrently) in the nodes. A part of the initialization involves each of the identical nodes discovering by sending messages its position in whatever size cube was specified in the startup packet sent from the intermediate host. This initialization, illustrated in Figure 7, involves messages that also check the function of all of the communication channels to be used. Program loading following initialization typically loads the kernel.

A *crystalline* applications environment is characterized by programs written in C, in which there is a single process per node, and in which messages are sent by direct I/O operations to a specified channel. This system was developed by the physics users for producing very efficient application programs for computations that are so regular that they do not require message routing.

The operating system kernel, already described in outline, supports a *distributed process* environment with a copy of the kernel running in each node. The kernel is 9K bytes of code and 4K bytes of tables, and is divided into an "inner" kernel and "outer" kernel. Any storage in a node that is not used for the kernel or for processes is allocated as a a kernel message buffer that is used to queue messages.

The "inner kernel", written in 8086 assembly, performs message sending and receiving in response to system calls from user processes. These calls pass the address of a message descriptor, which is shared between the kernel and user process. There is one uniform message format that hides all hardware characteristics, such as packet size. The kernel performs the construction and interpretation of message headers based on the descriptor information. The hardware communication channels allow very fast and efficient "one-trip" message protocols, with long messages being automatically fragmented. Messages being sent are queued in the sending process instead of being copied into the kernel message buffer, unless the message is local to the node. Local messages are either copied to the destination if the matching receive call has already been executed, or are copied into the message buffer to assure a consistency in the semantics of local and non-local send operations.

Processes are often required to manage several concurrent message activities. Thus the send and receive calls do not "block". The calls return after creating a request that remains *pending* until the operation is completed. The completion of the message operation is tested through a lock variable in the message descriptor. Program execution can continue concurrently with many concurrently pending communication activities. A process may also use a *probe* call that determines

whether a message of a specified type has been received and is queued in the kernel message buffer. A process that is in a situation where no progress can be made until some set of message areas are filled or emptied may elect to defer execution to another process. The inner kernel schedules user processes by a simple round robin scheme, with a process running for a fixed period of time or until it performs the system call that defers to the next process. The storage management and response to error conditions are conventional.

The "outer kernel" is structured as a set of priviledged processes with which user processes communicate by messages rather than by system calls. One of these outer kernel processes performs process spawning and killing. A process can be spawned either as a copy of a process already present in the node, in which case the code segment is shared, or from a file that is accessed through system messages between the spawn process and the intermediate host. Because process spawning is invoked by messages, it is equally possible to build process structures from processes running in the cube, in the intermediate host, or network-connected machines. One other essential outer kernel process is known as the "spy" process, and permits a process in the intermediate host to examine and modify the kernel's tables, queued messages, and process segments.

Our current efforts are focused on intermediate host software to allow both time- and space-sharing of the cube.

Applications and Benchmarks

Caltech scientists in high energy physics, astrophysics, quantum chemistry, fluid mechanics, structural mechanics, seismology, and computer science, are developing concurrent application programs to run on Cosmic Cubes. Several research papers on scientific results have already been published, and other applications are developing rapidly. Several of us in the Caltech computer science department are involved in this research both as system builders and also through interests in concurrent computation and applications to VLSI analysis tools and graphics.

Application programs on the 64-node Cosmic Cube execute up to 3 million floating point operations per second. The more interesting and revealing benchmarks are those for problems in which the machine operates at less than peak speeds. A single Cosmic Cube node at 5 MHz clock rate runs at 1/6th the speed of the same program compiled and run on a VAX11/780. Thus we should expect the 64-node Cosmic Cube to run at best $(1/6)(64) \approx 10$ times faster than the VAX11/780. Quite remarkably, many programs reach this performance, with measured values of σ, as defined previously, ranging from about 0.025 to 0.5. For example, a typical computation with $\sigma = 0.2$ exhibits a speedup $S = (1/6)(64)/(1.2) \approx 9$. One should not conclude that applications with larger σ are unreasonable; indeed, given the economy of these machines it is still attractive to run production programs with $\sigma > 1$.

As an example of its applications at Caltech, a lattice computation programmed by physics post-doc Steve Otto has run for an accumulated 2,500 hours on the 6-cube. This program is a Monte Carlo simulation on a 12x12x12x16 lattice, an investigation of the predictions of quantum chromodynamics, a theory that explains the substructure of particles such as protons in terms of quarks and the glue field that holds them bound. Otto has shown for the first time in a single computation both the short range Coulombic force and constant long range force between quarks. The communication overhead in this naturally load balanced computation varies from $\sigma = 0.025$ in the phase of computing the gauge field, to $\sigma = 0.05$ in computing observables by a contour integration in the lattice.

Amongst the most interesting and ambitious programs currently in development is a concurrent MOS-VLSI circuit simulator, called CONCISE, formulated and written by computer science graduate student Sven Mattisson. In addition to being a vehicle for developing techniques for

less regular computations, this program promises very good performance in a computation that consumes a large fraction of the computing cycles on many high performance computers.

The simulation of an electrical circuit involves solving repeatedly a set of simultaneous nonlinear equations. The usual approach, illustrated in Figure 8, is to compute from the circuit models piecewise linear admittances, and then to use linear equation solution techniques. CONCISE uses a nodal admittance matrix formulation for the electrical network. The admittance matrix is sparse, but because electrical networks have arbitrary topology, does not have the crystalline regularity of the physics computations. At best the matrix is "clumped" because of the locality properties of the electrical network.

This program is mapped onto the cube by partitioning the admittance matrix by rows into concurrent processes. The linear equation solution phase of the computation, a Jacobi iteration, involves considerable communication, but the linearization that requires about 80% of the execution time on sequential computers is completely uncoupled. Integration and output in computing transient solutions are small components of the whole computation. The computation is actually much more complex than we can describe here; for example, the integration step is determined adaptively from the convergence of previous solutions.

Amongst the many unknowns in experimenting with circuit simulation, as a paradigm of less regular computations that can be performed on machines such as the Cosmic Cube, are the interaction between communication cost and load balancing in the mapping of processes to nodes. Although the "clumping" can be exploited in this mapping to localize communication, it may also concentrate many of the longer iterations occurring during a signal transient into a single node, thus creating a "dynamic" load imbalance in the computation.

Future Perfect Cubes

The present system is never as perfect as the future system, on which we have not yet had the opportunity to view mistakes and oversights. Although one can polish and fix the software on a daily basis, the learning cycle on the architecture and hardware is much longer. Let us then summarize briefly what this experiment has taught us so far, and indulge in some speculations about future systems of this same general class.

Although programming these machines has not turned out to be as difficult as we should have expected, we have a long agenda of possible improvements for the programming tools. Most of the deficiencies are concerned with the representation and compilation of process code. There is nothing in the definition of the message passing primitives that we would want to change, but because we have tacked these primitives onto programming languages simply as external functions, the process code is unnecessarily baroque.

The way in which the descriptors for "virtual channels" are declared, initialized, and manipulated (Fig 4) is not disguised by a pretty syntax, but more fundamentally, the attention the programmer must give to blocking on lock variables is tedious, and can create incorrect or unnecessary constraints on the message and program sequencing. Such tests are better inserted into the process code automatically, based on a data flow analysis, similar to that used by optimizing compilers for register allocation. These improvements may be only aesthetic, but they are a necessary preliminary to making these systems less intimidating to the beginning user.

The cost/performance of this class of architectures is quite good even with today's technologies, and progress in microelectronics can be translated particularly easily into either increased performance or decreased cost. The present Cosmic Cube node is not a large increment in complexity over the million-bit storage chips that are expected in a few years. Systems of 64 single-chip node

elements could fit in work stations, and systems of thousands of nodes are interesting supercomputers. Although this approach to high performance computation is limited to applications that have highly concurrent formulations, the applications developed on the Cosmic Cube have shown us that many, perhaps even a majority, of the large and demanding computations in science and engineering are in this category.

It is also reasonable to consider systems with nodes that are either larger or smaller than the present Cosmic Cube nodes. We have developed at Caltech a single-chip "Mosaic" node with the same basic structure as the Cosmic Cube node, but with less storage, for experimenting with the engineering of systems of single chip nodes, and with the programming and applications of finer grain machines. Such machines offer a still higher return in performance per cost than the Cosmic Cube. However, we expect them to be useful for a somewhat smaller class of problems. Similarly, the use of better, faster instruction processors, higher capacity storage chips, and integrated communication channels, suggest machines whose nodes would be about the same physical size as the Cosmic Cube nodes, but would provide an order of magnitude higher performance and storage capacity.

The present applications of the Cosmic Cube are all compute- rather than I/O-intensive. However, it is possible to include I/O channels with each node, so that as much I/O bandwidth can be created as one might need. Such machines could be used, for example, with many sensors, such as the microphone arrays towed behind seismic exploration ships. The computing could be done in real time instead of through the medium of hundreds of tapes conveyed to a supercomputer. One can, similarly, attach disks for secondary storage to a subset of the nodes.

History and Acknowledgements

The origins of the Cosmic Cube project can be traced to research at Caltech in the 1978-80 period by graduate students Sally Browning and Bart Locanthi. These ideas were also very much influenced by several other researchers. We sometimes refer to the Cosmic Cube by a term from a 1977 paper by Herbert Sullivan and T L Brashkow, a *homogeneous machine*, a machine "of uniform structure". C A R Hoare's communicating sequential processes notation, the "actor" paradigm developed by Carl Hewitt, the "processing surface" experiments of Alain Martin, and the "systolic" algorithms described by H T Kung, Charles Leiserson, and Clark Thompson, encouraged us to consider message passing as an explicit computational primitive.

The Cosmic Cube design is based in largest part on extensive program modeling and simulations carried out in 1980-1982 by Charles R Lang. It was from this work that the communication plan of a binary n-cube, the bit rates of the communication channels, and the organization of the operating system primitives were chosen. Together with early simulation results, a workshop on "homogeneous machines" organized by Carl Hewitt in the summer 1981 helped create the confidence that it was time to build an experimental machine.

The logical design of the Cosmic Cube was done by computer science graduate students Erik DeBenedictis and Bill Athas. The early crystalline software tools were developed by physics graduate students Eugene Brooks and Mark Johnson. The machine intrinsic and kernel code was written by Bill Athas, Reese Faucette, and Mike Newton, with Alain Martin, Craig Steele, Jan van de Snepscheut, and Wen-King Su contributing valuable critical review of the design and implementation of the distributed process environment.

This extended experiment is sponsored through the VLSI program in the Information Processing Techniques Office of the Defense Advanced Research Projects Agency. We thank Bob Kahn, Duane Adams, and Paul Losleben both for their support and for their interest in these efforts.

References

Sally A Browning, "The Tree Machine: A Highly Concurrent Computing Environment", Caltech Computer Science Technical Report 3760:TR:80, 1980.

William D Clinger, "Foundations of Actor Semantics", PhD Thesis, MIT Dept of EE&CS, May 1981.

Geoffrey C Fox and Steve W Otto, "Algorithms for Concurrent Processors", *Physics Today*, May 1984. This special issue on "advances in computers for physics" includes 4 other related articles.

C A R Hoare, "Communicating Sequential Processes", *CACM* 21, 8, August 1978.

R W Hockney, C R Jesshope, *Parallel Computers*, Adam Hilger, Bristol, 1981.

H T Kung, "The Structure of Parallel Algorithms", in *Advances in Computers*, vol 19, Academic Press, 1980.

Charles R Lang, "The Extension of Object-Oriented Languages to a Homogeneous, Concurrent Architecture", Caltech Computer Science Technical Report 5014:TR:82, 1982.

Bart N Locanthi, "The Homogeneous Machine", Caltech Computer Science Technical Report 3759:TR:80, 1980.

Chris Lutz, Steve Rabin, Chuck Seitz, and Don Speck, "Design of the Mosaic Element", *Proc Conf on Advanced Research in VLSI*, MIT, Artech Books, pp 1-10, 1984.

Alain J Martin, "A Distributed Implementation Method for Parallel Programming", *Information Processing 80*, North-Holland, 1980.

Charles L Seitz, "Experiments with VLSI Ensemble Machines", *Journal of VLSI and Computer Systems*, vol 1, no 3, Computer Science Press, 1984.

J T Schwartz, "Ultracomputers", *ACM Trans Programming Languages & Systems*, vol 2, no 4, pp 484-521, October 1980.

Herbert Sullivan and T R Brashkow, "A Large Scale Homogeneous Machine I & II", *Proc 4th Annual Symposium on Computer Architecture*, pp 105-124, 1977.

Willis H Ware, "The Ultimate Computer", *IEEE Spectrum*, March 1972, pp 84-91.

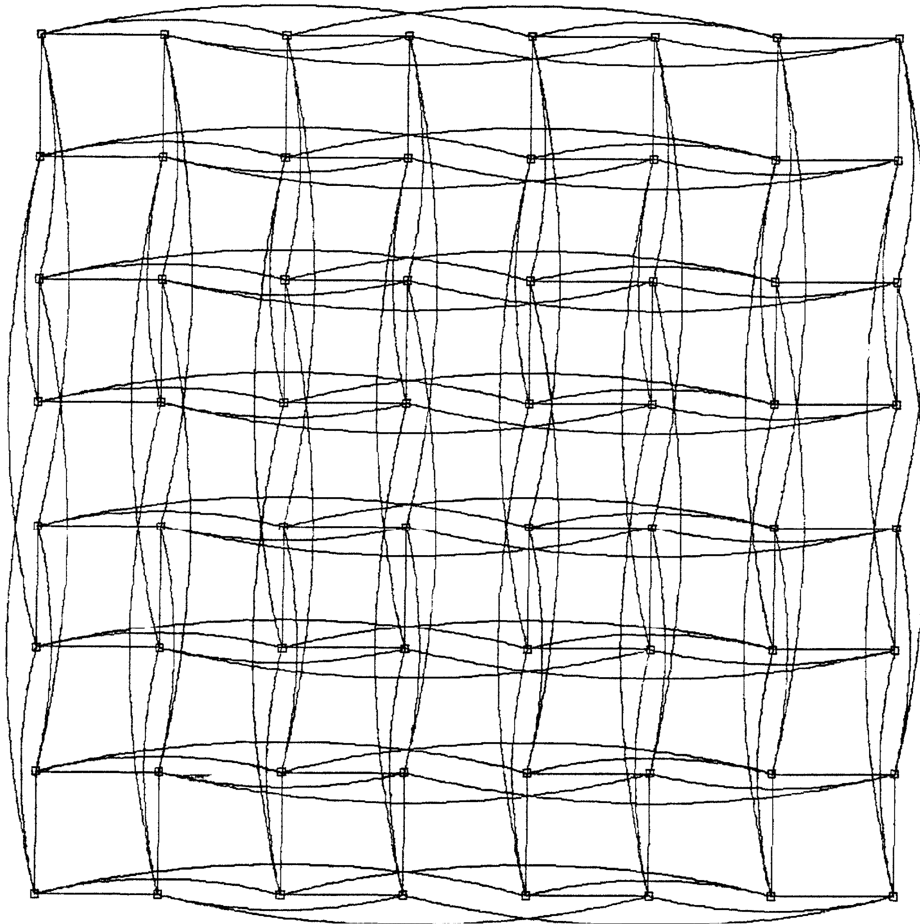
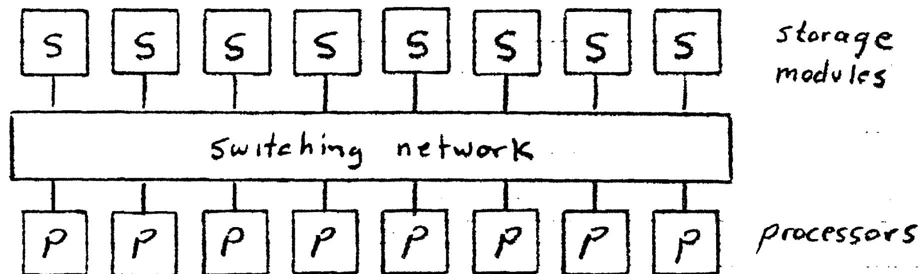
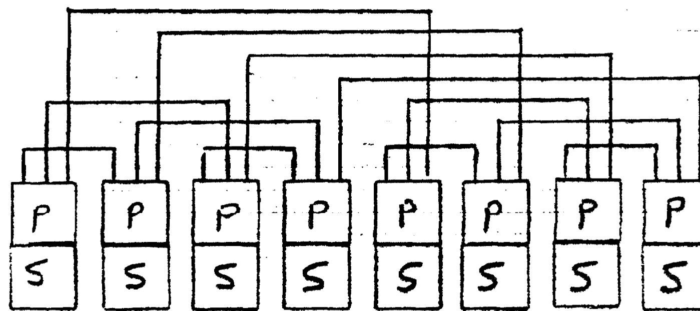


Figure 1: A hypercube, also known as a binary or Boolean n-cube, is the way in which $N = 2^n$ small computers called nodes are connected by point-to-point communication channels in the Cosmic Cube. This 6-dimensional cube, or binary 6-cube, corresponding to a 64-node machine, is shown projected onto two dimensions.

<<Note to editor: we can provide this illustration from (computer-driven) laser printer output at any scale and line density you desire.>>



(a) Most multiprocessors are structured with a switching network, either a crossbar connection of buses or a multistage routing network, between the processors and storage. The switching network introduces a latency in the communication between processors and storage, and does not scale well to large sizes. Communication between processes running concurrently in different processors occurs through shared variables and common access to one large address space.



(b) Message passing multicomputer systems retain a physically close and fast connection between the processors and their associated storage. The concurrent computers, called nodes, can send messages through a network of communication channels. The network shown here is a 3-dimensional cube, a small version of the communication plan used in 6 dimensions in the 64-node cosmic cube.

Figure 2: A comparison of shared storage multiprocessors and message passing machines. Various hybrids between the two extreme points pictured here are also possible.

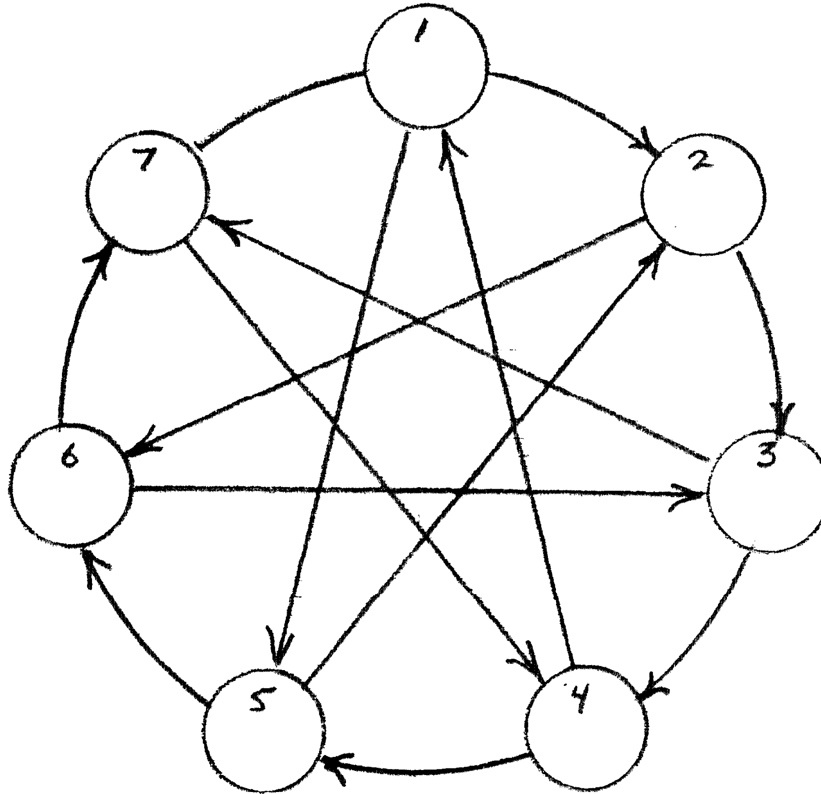


Figure 3: Process Structure for a concurrent formulation of the N-body problem. In this example, one is computing the time evolution -- orbital positions -- of N=7 bodies that interact by a symmetrical force such as gravity. Messages containing the position and mass of the particles are sent from each process (N-1)/2 steps around the ring, accumulating the forces due to each interaction, while the process that is host to that body accumulates the other (N-1)/2 forces. The messages are then returned over the chordal paths to their host process, the forces are summed, and the position and velocity of the body is updated. This example is representative of many computations that are demanding simply because of the number of interacting parts, not because the force law that each part obeys is complex.

```
/* process for an n-body computation, n odd, with symmetrical forces */
#include "cubedef.h"       /* cube definitions */
#include "force.h"         /* procedures for computing forces and positions */

struct body     {   double pos[3];          /* body position x,y,z          */
                    double vel[3];          /* velocity vector x,y,z        */
                    double force[3];        /* to accumulate forces         */
                    double mass;            /* body mass                    */
                    int    home_id;         /* id of body's home process    */
                } host, guest;

struct startup {    int n;                  /* number of bodies             */
                    int next_id;            /* ID of next process on ring   */
                    int steps;              /* number of integration steps */
                } s;

struct desc my_body_in, my_body_out, startup_in; /* IH channels            */
struct desc body_in, body_out, body_bak;         /* inter-process channels  */

cycle() /* read initial state, compute, and send back final state */
{
    int i; double FORCE[3];

    /* initialize channel descriptors */
    /* init(*desc, id, type, buffer_len, buffer_address); */
    init(&my_body_in ,0,0,sizeof(struct body)/2,&host); recv_wait(&my_body_in);
    init(&startup_in ,0,1,sizeof(struct startup)/2,&s); recv_wait(&startup_in);
    init(&my_body_out,      IH_ID, 2, sizeof(struct body)/2, &host);
    init(&body_in    ,          0, 3, sizeof(struct body)/2, &guest);
    init(&body_out   , s.next_id, 3, sizeof(struct body)/2, &guest);
    init(&body_bak   ,          0, 4, sizeof(struct body)/2, &guest);

    while(s.steps--) /* repeat s.steps computation cycles */
    {
        body_out.buf = &host;                  /* first time send out host body */

        for(i = (s.n-1)/2; i--;)                   /* repeat (s.n-1)/2 times    */
        {
            send_wait(&body_out);                  /* send out the host|guest   */
            recv_wait(&body_in);                   /* receive the next guest    */
            COMPUTE_FORCE(&host,&guest,FORCE);     /* calculate force           */
            ADD_FORCE_TO_HOST(&host,FORCE);        /* may the force be with you */
            ADD_FORCE_TO_GUEST(&guest,FORCE);      /* and with the guest, also  */
            body_out.buf = &guest;                 /* prepare to pass the guest */
        }
        body_bak.id = guest.home_id;                       /* send guest back    */
        send_wait(&body_bak); recv_wait(&body_bak);        /* the envoy returns  */
        ADD_GUEST_FORCE_TO_HOST(&host,&guest);
        UPDATE(&host);                                     /* integrate position */
    }
    send_wait(&my_body_out);  /* send body back to host, complete one cycle  */
}

main() { while(1) cycle(); }  /* main execute cycle repeatedly */
```

Figure 4: C language process code for the N-body example.

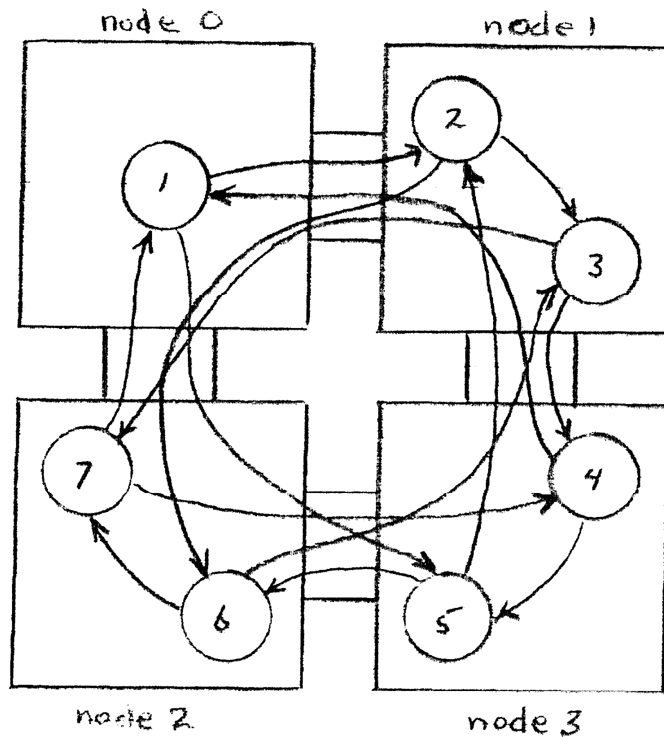


Figure 5: Embedding of the process structure for the 7-body example into a 4-node machine. The way in which the processes are distributed does not influence the results computed, but does influence through load balancing and message locality the speedup achieved by using 4 computers for this task rather than one.

Figure 6: Photograph of the 64-node Cosmic Cube in operation. The nodes are packaged as one circuit board per node in the long card frame on the bench top. The 6 communication channels from each node are wired in a binary 6-cube on the backplane on the underside of the card frame. The separate units on the shelf above the long 6-cube box are the power supply and an "intermediate host" (IH) that connects by an extra communication channel to node 0 in the cube.

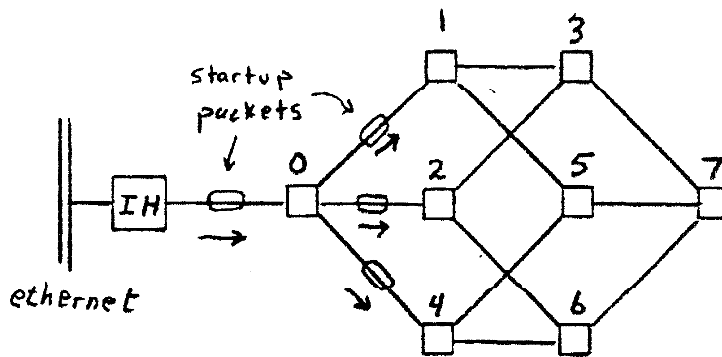


Figure 7: In the initialization of the Cosmic Cube, each of the identical nodes discovers its identity, and checks all the communication channels, with a message wave that traverses the 3-cube pictured from node 0 to node 7, and then from 7 to 0. If node 3 did not respond to messages, nodes 1, 2, and 7 would report this failure back to the corner over other channels.

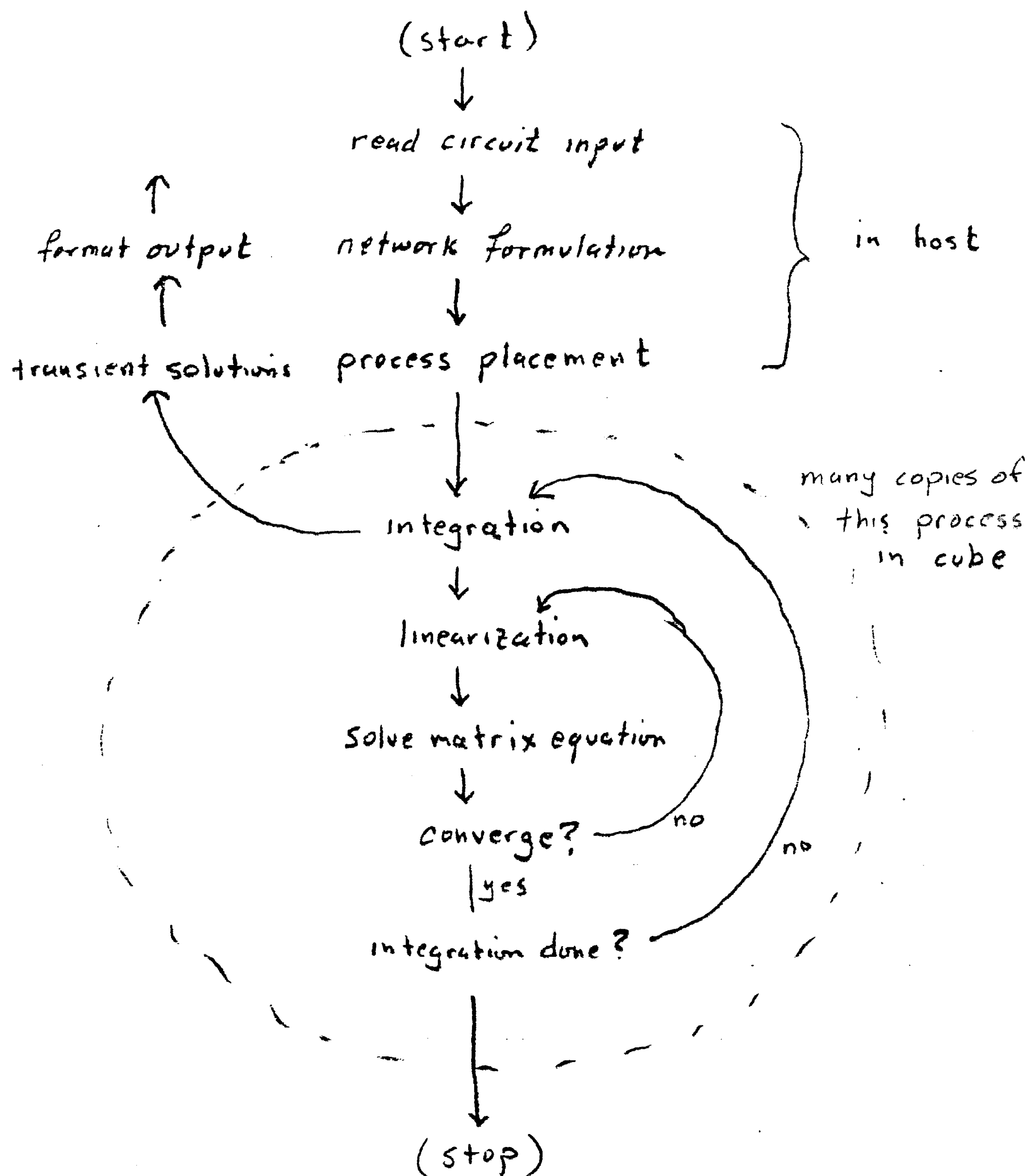


Figure 8: The Organization of the circuit simulator CONCISE. The sequential and concurrent versions of this program differ in the concurrent program employing many instances of the process shown in the dashed circle. Communication between processes is required in the "solve matrix equation" phase.

A Special Purpose Processor for Switch-Level Simulation

William J. Dally and Randal Bryant

Department of Computer Science 256-80
California Institute of Technology
Pasadena, California 91125

Abstract

Special purpose hardware for performing switch-level simulation of MOS VLSI circuits 500 times faster than a conventional computer is described. A virtual processor architecture allows circuits of any size to be simulated independent of the number of processors installed.

Introduction

As the complexity of VLSI circuits approaches 10^6 devices, the computational requirements of design verification are exceeding the capacity of general purpose computers. To provide the computing power required to verify these complex VLSI chips, special purpose hardware for performing simulation is required. Existing logic simulation engines [1,2,3,4] are inadequate for MOS VLSI because they cannot accurately model MOS circuits. Switch-level simulation, on the other hand, models the effects of capacitance and transistor ratios at speeds comparable to logic simulation. Existing machines limit the size of a circuit which can be simulated by binding circuit elements to hardware at compile time. Virtual network processing allows circuits of any size to be simulated by binding circuit elements to hardware at run-time.

A state of the art VLSI chip in 1982 contained $\approx 10^5$ devices and required $\approx$ 1 week of CPU time to complete a single verification cycle. The complexity of VLSI circuits is increasing at an exponential rate and will soon reach the level of 10^6 devices per chip. Since both the number of test vectors required to verify a chip and the amount of computation required to simulate one test vector scale at least linearly with the complexity of a chip, the amount of computation required verify a chip scales at least quadratically with complexity. Thus, as shown in figure 1, a 1986 chip containing $\approx 10^6$ devices will require about 2 years of CPU time to completely verify on a conventional computer. Clearly, special purpose hardware is required to simulate these complex chips in a timely manner.

This paper describes the architecture of the MOSSIM Simulation Engine (MSE) [5], a special purpose processor for performing switch-level simulation of MOS VLSI circuits. The MSE implements the MOSSIM algorithm [6] in hardware. Subnetwork concurrency, functional concurrency and specialization are used to accelerate the algorithm so that a single processor MSE performs switch-level simulation 200 to 500 times faster than a VAX 11/780. Several MSE processors can be connected in parallel to achieve additional speedup. The MSE performs run-time binding of circuit elements to make more efficient use of parallel processors and to allow circuits of any size to be simulated.

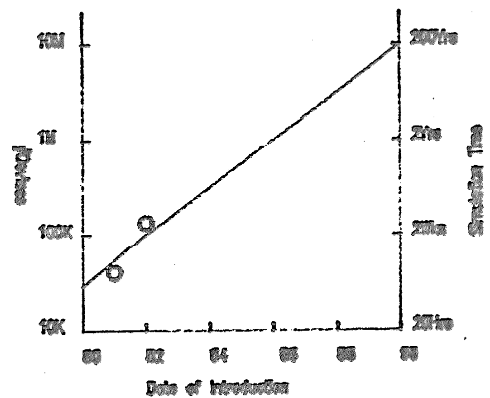


Figure 1. Scaling of Simulation Time

Algorithm

The MSE implements the MOSSIM algorithm [6] in hardware. This algorithm, is based on a formal switch-level model of MOS transistor networks. By modeling MOS transistor ratios and node capacitances and by considering an MOS transistor as a truly bidirectional device, MOSSIM provides considerably more accurate simulation of MOS LSI than conventional logic simulators. MOSSIM achieves performance comparable with logic gate simulators by using an incremental algorithm which exploits the sparseness and locality of events in a circuit.

The switch-level network model consists of a set of nodes connected by a set of transistors. Nodes are assigned sizes based on their relative capacitance. The abstraction of node size accurately models the behavior of charge sharing in MOS circuits while avoiding the complexity of parametric capacitances. Transistor ratios are modeled by assigning each transistors a strength based on its relative ratio. Both nodes and transistors have states from the set $0, 1, X$.

The MOSSIM algorithm performs repeated solutions of the steady state excitation of the network until a stable state is reached. Unlike logic simulators which propagate logic values through a gate network, MOSSIM operates by analyzing paths on the graph formed by conducting transistors. Each solution involves finding the strongest path from a source of logic zero or logic one to each scheduled node and proceeds in four phases. In the first phase all nodes which may be affected by a changing transistor are scheduled for evaluation. In phase two, the strongest definite path to each scheduled node is found. The

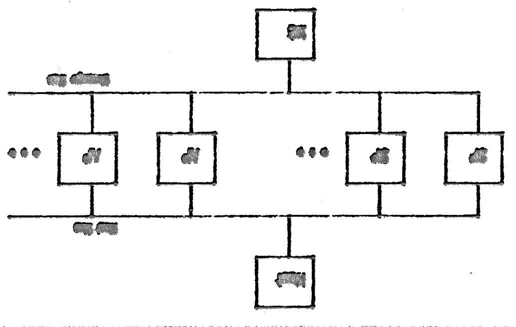


Figure 2. MSE Block Diagram

strongest unblocked path from logic zero or logic one to each node is found in phase three and the logic state of the node is set based on this path. In phase four, the state transistors controlled by changing nodes are updated. This simulation using blocking paths handles unknown logic states in a very accurate manner [6].

The MOSSIM algorithm was modified for the hardware implementation by manipulating each phase of the simulation to fit into an algorithm template. The template was then implemented in hardware by building dedicated units for the tasks of scheduling, updating nodes, and traversing the network.

Architecture

The MSE achieves its performance through subnetwork concurrency, functional concurrency and specialization. Subnetwork concurrency involves partitioning the network into several subnetworks and simulating the subnetworks in parallel. Within each processor functional concurrency is achieved by performing the operations of scheduling, node evaluation and network traversal in parallel. Finally, in performing each of these operations specialized logic circuits are used to implement the time critical functions offering orders of magnitude speedup over general purpose computer instructions.

As shown in figure 2, the MSE consists of a number of subnetwork processors (SP) connected by a message bus (MB) to a message switch (MS). Auxiliary processors (AP) may also be connected to the MB to perform functional simulation. A host processor (HP) is connected to all processors by the host bus (HB). The HP controls the operation of the MSE, performs virtual processor swapping and has the ability to read and write each register and memory location in the machine.

The SPs and APs simulate subnetworks of a circuit in parallel. Interactions between subnetworks create messages which are routed through the MS. The MS performs a virtual subnetwork to physical SP translation for each message and queues messages for subnetworks which are swapped out. Simulation studies indicate that up to eight SPs may be attached to a single MB/MS without significant degradation due to bus contention.

The SP, shown in figure 3, is the hardware kernel of the MSE. Each SP has a capacity of 4096 nodes and 16384 transistors partitioned into separate physical processors of 1024 nodes each. An SP implements the MOSSIM algorithm performing all operations for its subnetwork and sending messages to the MS for operations involving other subnetworks. To exploit all possible functional concurrency a separate function unit is associated with each major data structure. The scheduling unit (SU) implements the scheduling priority queues. The node memory and a relaxation unit which operates on nodes are contained in the node operation unit (NOU). The network traversal unit (NTU), contains the link and gate lists which describe the transistors and the network connectivity. The SP also contains two additional function units. The control processor supervises operation of the SP, and the input/output unit handles inter-processor communication.

In operation the address of the highest priority node scheduled for the current simulation phase is removed from the priority queue by the SU and transmitted to the NOU. The NOU reads the node record at this address and sends pointers to its adjacency lists to the NTU. The NTU then returns the addresses of all adjacent nodes. The NOU operates on these nodes scheduling those which change by sending their address to the SU. All operations are pipelined to keep all three units busy at all times. A stall mechanism is used to interlock the pipeline.

Specialized hardware was added to each unit as necessary to balance their performance so that no one unit was a bottleneck. The SU priority queue was implemented in hardware so that an insertion takes one cycle and a deletion takes two cycles. The adjacency lists in the NTU are sequenced by counters at a rate of one node per cycle. The relaxation operation in the NOU is performed in one cycle. When this $200ns$ cycle is compared to the 50 lines of source code ($\approx 200\mu s$) required to implement this operation in software, it is easy to see that specialization achieves much greater performance improvements than concur-

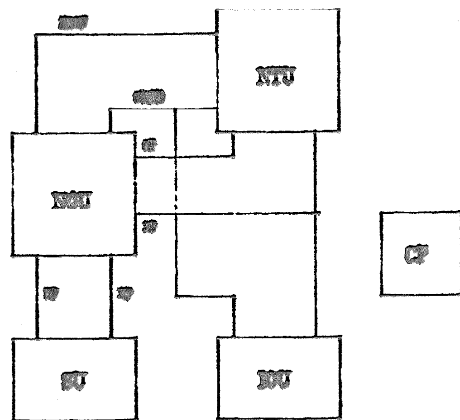


Figure 3. Subnetwork Processor Block Diagram

rency.

Virtual Network Processing

The very locality of activity which makes subcircuit concurrency possible can lead to a degradation in performance due to idle processors. Typically only 5% of the nodes in a circuit are active at a given time; however, this 5% tends to be clustered as 50% of the nodes in 10% of the subnetworks. Since the amount of activity in each processor is not equal, processors with low activity will complete a processing step early and remain idle until all processors complete the step. To avoid this potential degradation, we have developed the concept of virtual network processors (VNPs). This concept is analogous to that of virtual memory. We partition the circuit into many more subcircuits than we have physical processors. A virtual network processor, associated with each subcircuit, contains the complete state of the simulation of that circuit. To maximize throughput, the VNPs are dynamically mapped to SPs based on activity. The virtual network processor mechanism allows the MSE to simulate large circuits with the size of the circuit limited only by the amount of backing store available to hold the circuit description.

Performance

MSE performance has been measured using both a functional simulator and a chip-level simulation of the MSE hardware. At a 5MHz clock rate, a single MSE processor performs 5M relaxation operations per second. For typical networks this corresponds to 250K gate evaluations per second (GEPs). Simulation results indicate that eight MSE processors can be connected on a single bus to achieve a throughput of 2M GEPs. For comparison, MOSSIM on a VAX 11/780 performs about 300K GEPs. Accounting for I/O overhead during a simulation, we estimate that a single SP MSE will simulate 200 to 500 times faster than a VAX.

Status

A prototype MSE subnetwork processor has been constructed and is currently being debugged. Consisting of $\approx$ 400 integrated circuits packaged on a single board, this processor can hold subnetworks totaling 16K transistors and 4K nodes. We expect this prototype to be operational by early Fall 1984. The development of systems software for the processor is proceeding in parallel with the construction of the hardware.

Conclusion

The ideas incorperated in the MSE can be applied to many problems of a similar nature. For a problem to be a candidate for special purpose hardware, it must be computationally demanding, have a stable and structured algorithm, have potential parallelism (both functional and structural), and have some operations which are poorly matched to the capabilities of a general purpose computer. The concept of virtual network processing, run-time binding of sub-networks to hardware, is also applicable to many problems. Virtual network processing can be applied to any application which is characterized by sparse clustered activity. Switch-level simulation has all of these properties. Other problems which are candidates for special purpose hardware and virtual network processing include circuit simulation, geometry compaction, and routing. It is interesting to note that these problems also use a sparse dynamic graph data structure and could make use of the scheduling and network traversal units of the MSE.

Acknowledgements

We thank Chuck Seitz for providing support and guidance to this project and Hsiu-Tung Yu for contributing to the software and hardware debugging. This research was sponsored by the Defense Advanced Research Projects Agency, ARPA order number 3771, and monitored by the Office of Naval Research under contract number N00014-79-C-0597.

References

[1] Denneau, M. M.,"The Yorktown Simulation Engine," *19th Design Automation Conference*, ACM, 1982, pp. 51-54.

[2] "ZyCad LE-001 and LE-002 Product Description," ZYCAD,1982.

[3] Bartow, R. L. *et al*,"Architecture of a Hardware Simulator," *IEEE Conference of Circuits and Computers*,1980, pp.891-893.

[4] "Daisy Megalogician, Product Description," Daisy Systems, 1984.

[5] Dally, W., *The MOSSIM Simulation Engine: Architecture and Design*, Caltech Technical Report 5123:TR:84, April 1984.

[6] Bryant, R., "A Switch-Level Model and Simulator for MOS Digital Systems," *IEEE Transactions on Computers*, vol. C-33, pp. 160-177, February 1984.

SOFT ERROR CORRECTION FOR INCREASED DENSITIES IN VLSI MEMORIES

Khaled Abdel-Ghaffar and Robert J. McEliece

Department of Electrical Engineering
California Institute of Technology

1. INTRODUCTION

If VLSI RAM densities are to continue to increase, it will undoubtedly be necessary to take the problems associated with "soft errors" much more seriously than has previously been done. In this paper, we propose a methodology for analyzing the effects of soft errors in VLSI RAMS as feature sizes decrease, and for taking corrective action with error-correcting codes. We will take a parametric approach, making several different assumptions about how the error severity will scale as feature size decreases, and our conclusions will be stated relative to the particular assumption made. It is our hope that as more definite information about VLSI error-scaling becomes available, the results of this paper will prove helpful for designers of ultra-dense memories.

The most important source of soft errors in the near future is expected to be alpha-particle effects [1]. The charge track produced in the memory cells by a sufficiently energetic alpha particle emitted from the impurities in the IC package can change the logical state. Cosmic rays can produce errors by essentially the same mechanism [2].

If feature size becomes sufficiently small, thermal and quantum effects may become significant [3],[4],[5], and, we will argue below, will impose an ultimate limit (doubtless unrealistically optimistic) on information densities.

We wish to argue that error-correcting codes can be used to reduce the effects of soft errors on dense memories. Unfortunately, however, no accurate models have been established which predict the severity of error at submicron feature sizes. Thus, as mentioned above, we have pursued our study using a family of abstract error models. These models are introduced in Section 2. In Section 3, we will study the improvement possible (as measured in area per information bit) with coding, for each of our abstract models. In Section 4, we consider the problem of placing the encoder and decoder on the RAM chip. In Section 5, we discuss an explicit class of codes, orthogonal codes, which we show are attractive for a certain class of error models.

The research described in this paper was sponsored by the Defense Advanced Research Projects Agency, ARPA order 3771, and monitored by the Office of Naval Research under contract number N00014-79-C-0597.

Finally in Section 6, we discuss our abstract results in the light of what little we are able to say about the physics of the error-producing mechanisms.

2. AN ABSTRACT MODEL

We begin with an abstract "chip" of unit area, which contains one memory cell. We apply a linear scaling of a factor of α, which produces on the original chip α^2 cells. We assume that if error-correction is present, it is performed at regular intervals (the interval will be a multiple of the chip's refresh period, and may vary with α).

We further assume that the probability of a given stored bit of information changing its logical state during a decoding period, satisfies the following set of assumptions:

1. It is position- and time-invariant.
2. It is independent of the current state of the chip.
3. It is independent of any errors which may occur elsewhere on the chip.

With these assumptions (which are certainly subject to challenge; see Section 6), storing and retrieving bits on the chip is equivalent to transmitting them over a memoryless binary symmetric channel (BSC). We will denote the transition probability of this channel by ε. We assume that ε increases with α, and that it tends to 1/2 as α tends to infinity. In our analysis, however, the quantity $1-2\varepsilon$ occurs more directly than ε, and so we define δ to be this quantity:

$$\delta = 1 - 2\varepsilon \quad .$$

3. THE MINIMUM AREA PER INFORMATION BIT (MAPIB)

When error-correcting coding is used to increase storage reliability, a certain number of the memory cells on the chip are reserved for the code's parity-checks, and will be called redundant bits. The remaining memory cells are used for storing data and are called information bits. The code rate R is defined to be the ratio of the total number of information bits to the total number of bits on the cell (α^2). As α increases, we expect a larger probability of error, and so the needed code rate R will decrease. Thus, although the total number of bits on the cell increases as α^2, the number of information bits cannot be expected to

0194-7111/84/0000/0248$01.00©1984 IEEE

grow this rapidly. Indeed, if the increase in error severity is sufficiently rapid, the number of information bits on the chip may actually begin to decrease when α exceeds some limit.

Thus the following question arises. What is the minimum area per information bit (MAPIB) required for reliable data storage, relative to a given error scaling model, assuming the technology to produce scaled chips with any $\alpha \geq 1$? We can answer this question using known results from information theory.

According to Shannon's noisy-channel coding theorem [6], the probability of (decoded) error for the stored bits can be made arbitrarily small if and only if the code rate R is less than the capacity C, which is given by the formula

$$C = 1/2[(1-\delta)\log_2(1-\delta)+(1+\delta)\log_2(1+\delta)]$$

It follows that the minimum area per information bit for a given code rate R, which we denote by A(R), is given by

$$A(R) = \frac{1}{R[\delta^{-1}(C^{-1}(R))]^2}$$

where δ^{-1} and C^{-1} are the inverse functions of $\delta(\alpha)$ and $C(\delta)$, respectively. Hence

$$\text{MAPIB} = \min_{0 \leq R \leq 1} A(R)$$

For a given assumption about the relationship between α and δ, it is possible to draw a curve exhibiting the relationship between the minimum needed area per information bit as a function of R. Such a curve separates an "allowable region" in which reliable storage of data is, in principle, possible, from a "forbidden region," in which reliable storage is impossible. The shape of this curve depends critically on the error model, especially as α approaches infinity (Figures 1-4). However, the possible curves fall into three general categories:

Case 1 (Light Noise): $\alpha\delta \to \infty$.

For this (unlikely) set of models, an infinite number of information bits can be stored on a unit area. Fig. 1 shows this for $\delta = \alpha^{-1/2}$. MAPIB is zero.

Case 2 (Moderate Noise): $\alpha\delta \to \text{constant} \neq 0$.

In this case a finite nonzero number of information bits can be stored as $\alpha \to \infty$. Fig. 2 shows this for $\delta = \alpha^{-1}$. MAPIB is finite.

Case 3 (Severe Noise): $\alpha\delta \to 0$.

In this case, the error rate increases very rapidly with α, so that in the limit all of the bits on the chip become redundant, and the area/information bit tends to infinity. Fig. 3 shows this with $\delta = \alpha^{-2}$, and Fig.4 for $\delta = 1-2Q(10^4\alpha^{-3/2})$ (see Section 6). MAPIB is finite in this case also.

4. THE AREA REQUIRED FOR THE ENCODER AND DECODER

In Section 3 we implicitly assumed that the chip area was devoted entirely to information and redundant bits. In this section we will briefly consider chips in which the encoder and decoder themselves consume some of the chip area, so that less area is available for "bits."

In the following we will assume that the encoder and decoder are immune from error. Although this assumption is perhaps somewhat unrealistic, it is known that errors in active processors are much less frequent than errors in memories.

In Case 1 in Section 3 ($\delta\alpha \to \infty$), we showed that an infinite number of information bits can be stored reliably on the chip, and that MAPIB tends to zero. In fact, we can show that MAPIB tends to zero, even with the encoder and decoder on the chip. In this case, our analysis shows that most of the chip will be devoted to the processing circuitry, and a negligible fraction to the codeword bits.

We have not yet completed our studies of the impact of decoder area in Cases 2 and 3.

5. ORTHOGONAL CODES FOR MEMORY CHIPS

In order to achieve the potential MAPIB's promised in Section 3, it is necessary to use efficient and sophisticated codes. Here we consider only Case 1, and as before leave the consideration of Cases 2 and 3 to a later paper. Motivated by a result of Viterbi [7], we have found that the class of orthogonal codes can be used to achieve, in the limit, an infinite number of bits per unit of area. (An orthogonal code with n codewords of length N, with n<=N, can be constructed by taking the first n rows of an N x N (0,1) Hadamard matrix.)

6. DISCUSSION

In Section 3, we made a number of simplifying assumptions about the error mechanism which led us to consider the memoryless binary symmetric channel. In fact, some of these assumptions are known not to be valid for some error mechanisms. For example, it is believed that as cell dimensions decrease, a single alpha particle may cause multiple errors. If this is the case, the resulting channel will not be memoryless. However, it is a well-known if vague theorem of information theory [8] that for a given error probability, memory can only increase channel capacity, so that in principle our memoryless assumption is in fact pessimistic! We mention also that in NMOS technology, only cells representing logical-ones (uncharged depletion regions) are sensitive to alpha-particle errors. This would make the appropriate channel model a binary asymmetric channel. However, sense amplifiers and bit lines are also sensitive to alpha particles [9], and errors of this nature can effect both logical zeros and ones.

Although alpha-particles are expected to be