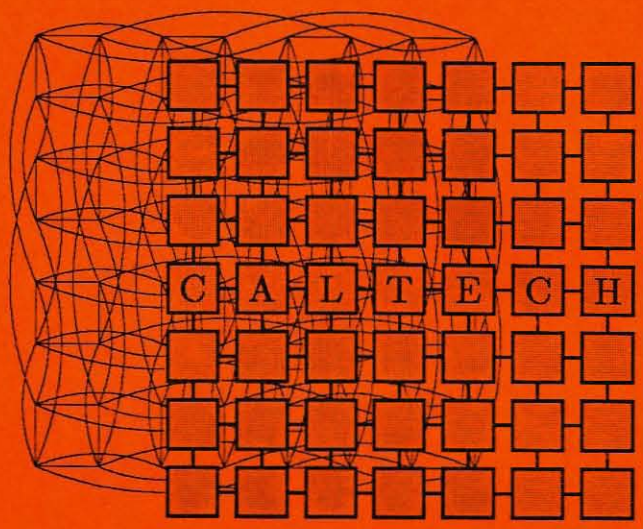


SUBMICRON SYSTEMS ARCHITECTURE PROJECT
Department of Computer Science
California Institute of Technology
Pasadena, CA 91125

Semiannual Technical Report

Caltech Computer Science Technical Report

Caltech-CS-TR-89-4

31 March 1989

The research described in this report was sponsored by the Defense Advanced Research Projects Agency, DARPA Order number 6202, and monitored by the Office of Naval Research under contract number N00014-87-K-0745.

SUBMICRON SYSTEMS ARCHITECTURE

Semiannual Technical Report

Department of Computer Science
California Institute of Technology

Caltech-CS-TR-89-4

31 March 1989

Reporting Period:	1 November 1988 – 31 March 1989
Principal Investigator:	Charles L. Seitz
Faculty Investigators:	K. Mani Chandy
	Alain J. Martin
	Charles L. Seitz
	Stephen Taylor

Sponsored by the
Defense Advanced Research Projects Agency
DARPA Order Number 6202

Monitored by the
Office of Naval Research
Contract Number N00014-87-K-0745

SUBMICRON SYSTEMS ARCHITECTURE

Department of Computer Science
California Institute of Technology

1. Overview and Summary

1.1 Scope of this Report

This document is a summary of research activities and results for the five-month period, 1 November 1988 to 31 March 1989, under the Defense Advanced Research Project Agency (DARPA) Submicron Systems Architecture Project. Previous semiannual technical reports and other technical reports covering parts of the project in detail are listed following these summaries, and can be ordered from the Caltech Computer Science Library.

1.2 Objectives

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

1.3 Highlights

- Mosaic prototype approaching completion (2.1).
- Delivery of 2nd-generation multicomputers (2.2)
- Programming with composition (3.3)
- First asynchronous microprocessor (4.1).
- Fast self-timed mesh routing chips (4.2).

2. Architecture Experiments

2.1 Mosaic Project

Chuck Seitz, Nanette J. Boden, Jordan Holt, Jakov Seizovic, Don Speck, Wen-King Su, Steve Taylor, Tony Wittry

The Mosaic C is an experimental fine-grain multicomputer, currently in development. Each Mosaic node is a single VLSI chip containing a 16-bit processor, a three-dimensional mesh router with each of its channels operating at 160Mb/s, a packet interface, at least 8KB of RAM, and a ROM that holds self-test and bootstrap code. These nodes are arrayed logically and physically in a three-dimensional mesh. We are working toward building a 16K-node (32×32×16) Mosaic prototype, together with the system software and programming tools required to develop application programs.

The Mosaic can be programmed using the same reactive-process model that is used for the medium-grain multicomputers that our group has developed. However, the small memory in each node dictates that programs be formulated with concurrent processes that are quite small. The Cantor programming system supports this style of reactive-process programming by a combination of language, compiler, and runtime support. The programmer is responsible only for expressing the computing problem as a concurrent program. The resources of the target concurrent machine are managed entirely by the programming system.

The Mosaic project includes many subtasks, which are listed below together with their current status:

Design, layout, and verification of the single-chip Mosaic node. The design and layout of the Mosaic C chip are now complete, and are going through extensive switch-level simulation tests, including the simulation of multiple nodes (see section 4.3). We expect to send a memoryless version of the node element to fabrication in about two weeks as a final check of the processor, packet interface, and router sections. These chips will be connected to external RAM and ROM to provide functional node elements for software development and host interfaces. Fabrication of the first chips in 1.2μm CMOS technology with RAM and ROM is anticipated in June 1989; quantity fabrication is anticipated in September 1989.

Internal self-test and bootstrap code. Since the Mosaic C is a programmable computing element, devoting a portion of the bootstrap ROM to self-testing greatly simplifies the logistics of producing these chips in significant quantity. The bootstrap and self-test code has been designed and is currently being written. The code will be tested using the ROM connected to the memoryless Mosaic C elements. Additional tests to the channels, which must be accomplished by the fabricator's automatic test equipment, are also being written.

Packaging. A preliminary packaging design based on TAB-packaged Mosaic

C chips was completed following a visit to Hewlett-Packard NID to understand their TAB packaging capabilities. The manufacturing and replacement unit contains eight nodes in a logical 2×2×2 submesh on a circuit-card module whose physical dimensions are approximately 2.5×5inches^2. These modules have stacking connectors that provide 160 pins on both the top and bottom, and are confined by pressure between motherboards to provide a three-dimensional connection structure that can be disassembled and reassembled for repair.

Cantor runtime system. A complete Cantor runtime system was written in Mosaic assembly code, and is now running correctly with a suite of small test programs under a Mosaic simulator on our medium-grain multicomputers (see section 3.1). This system provides the low-level implementation of message and process-creation primitives, and normally will be loaded as part of the Mosaic system initialization. The evolution of the Cantor programming language and the experience gained by use are two factors that are expected to affect continuing refinements to this system.

Cantor language, compiler, and application studies. A definition of a version of Cantor (3.0) with functions and limited message discretion was proposed in January 1989 by William C. Athas of UT Austin. We have been studying the changes in the runtime support that will be required by these improvements. In the interim, the definition and compiler implementation of Cantor 2.2 remain in use for application development.

Host interfaces and displays. The three-dimensional mesh structure of the Mosaic allows a very large bandwidth around the mesh edges. In order to initiate and interact with computations within the Mosaic, we must provide interfaces between the Mosaic message network and conventional computers and networks. One approach being studied is to use a memoryless Mosaic with a two-ported external memory as a convenient interface to workstation computers. Another external connection that is desired is a display interface. An elegant method that uses one 32×32 plane of a Mosaic as a rendering engine, frame buffer, and output video-conversion system has been developed. The detailed design of the video output generator that attaches to one edge of this 32×32 plane is now under way.

2.2 Second-Generation Medium-Grain Multicomputers*

Chuck Seitz, Joe Bechenbach, Christopher Lee, Jakov Seizovic, Craig Steele, Wen-King Su

A 16-node Intel iPSC/2 was delivered in November 1988, and a 16-node Symult Series 2010, a second-generation medium-grain multicomputer developed as a

* This segment of our research is sponsored jointly by DARPA and by grants from Intel Scientific Computers (Beaverton, Oregon) and Symult Systems (Monrovia, California).

joint project between our research project and Symult Systems, Inc. (formerly Ametek Computer Research Division), was delivered in December 1988. Both of these systems have been used extensively for programming system developments, applications, and benchmarks. We have encountered very few system problems in running existing Cosmic-C application programs on either the Symult Series 2010 or Intel iPSC/2.

Application programs typical of those that were written for first-generation multicomputers run 8–10 times faster per node on the Symult Series 2010 and on the Intel iPSC/2 than on first-generation machines, such as the Intel iPSC/1. Applications involving latency-sensitive non-local message traffic exhibit more dramatic improvements, particularly on the Series 2010, due to cut-through message routing being included in the hardware of these second-generation multicomputers.

Delivery of a 64-node Series 2010 is expected on 31 March 1989, and our 16-node Series 2010 will be returned briefly to Symult to be upgraded to 32 nodes and retrofitted with some hardware improvements to the mesh termination and host interfaces. The 32-node Series 2010 will continue as our principal programming-system-development machine. The 64-node Series 2010 and the 16-node iPSC/2 will be made available to outside users through the Caltech Concurrent Supercomputing Facilities. Outside users will include researchers at Caltech, as well as those associated with the Rice-Caltech-Argonne-Los Alamos (NSF Science and Technology) Center for Research in Parallel Computation. These systems will also be available for use by researchers in the DARPA community; DARPA researchers should contact Chuck Seitz (`chuck@vlsi.caltech.edu`) to make arrangements for access.

We expect to expand both the Intel iPSC/2 and Symult Series 2010 to larger configurations by the early part of CY90.

Copies of the Cosmic Environment system have been distributed to 13 additional sites during this period, bringing the total copies distributed directly from the project to over 160.

An effort has been started to implement major extensions of the Cosmic Environment host runtime system and the Reactive Kernel node operating system. The new CE will be based internally on reactive programming, and will allow a more distributed management of a set of network-connected multicomputers. The extended RK will support global operations across sets of cohort processes, including barrier synchronization, sum, min, max, parallel prefix, and rank. Another extension will be the support of distributed data structures, such as sets and ordered sets. These new features will be implemented at the RK handler level, where the message latency is only a fraction of that at the protected user level. The implementation of these algorithms at the handler level permits the performance of global and distributed-data-structure operations in times that do not greatly exceed those of user-level operations dealing with single messages.

Our Caltech project continues to work with both Intel and Symult on the architectural design, message-routing methods and chips, and system software for medium-grain multicomputers. We expect to see additional major advances in the performance and programmability of these systems over the next two years. In addition, we continue to develop applications in VLSI design and analysis tools, and in other areas in which the programming of these multicomputer systems presents particular difficulties or opportunities.

2.3 Cosmic Cube Project

Wen-King Su, Jakov Seizovic, Chuck Seitz

The Cosmic Cubes that were built in our project in 1983 and the Intel iPSC/1 d7 that was contributed to the project in 1985 continue to operate very reliably. Overall usage has decreased somewhat with the appearance of the second-generation multicomputers, but the iPSC/1 continues to be used fairly heavily within the research group for discrete event simulations, and by Caltech students and faculty in Aeronautics for supersonic-flow computations.

Neither the 64-node or 8-node Cosmic Cubes exhibited any hard failures in this five-month period. The two original Cosmic Cubes have now logged 3.8 million node-hours with only four hard failures, three of which were chip failures in nodes, and one a power-supply failure. A node MTBF in excess of 1,000,000 hours is probable based on this reliability experience.

3. Concurrent Computation

3.1 Cantor

Nanette J. Boden, Chuck Seitz

Programming Fine-Grain Multicomputers

The experiments we reported previously in application programming using Cantor 2.0 and 2.2 have suggested a series of changes to the Cantor language. William C. Athas, who led the development of Cantor while he was a graduate student and post-doc in the project, and who is now at UT Austin, has incorporated these ideas into the definition of a new version of Cantor (3.0). The principal structural changes are the introduction of limited discretion in receiving messages according to type, and in the approach to implementing functions.

In developing the Cantor programming system for the Mosaic, we mean to allow for these changes so that we may change to Cantor 3.0 as soon as a new compiler is produced.

Cantor for the Mosaic

Development of Cantor runtime support for the Mosaic multicomputer has progressed significantly during the last five months. Initially, we defined a Cantor Abstract Machine (CAM) that represents an idealized machine for executing Cantor code. The CAM instruction set includes single instructions that encapsulate complicated Cantor operations, such as process creation and message passing. By design, the implementation of these operations can be varied within native code generators for experimenting with different strategies. With the Mosaic, for example, we use a macro-assembler that translates the implementation for each CAM instruction into Mosaic instructions.

The definition of the first version of the Cantor runtime system for the Mosaic consisted chiefly of freezing efficient implementations for process creation and message passing, and expressing them with Mosaic instructions. In the case of process creation, a software cache of available reference values is maintained on each node so that processes can be created with low latency. These reference values are later bound to actual processes by special *creator* processes located on each node that allocate memory for new processes. Receiving a message on the Mosaic is implemented by having the runtime system determine the destination process, and then run that process to absorb the message. The runtime system also communicates with the runtime systems on other nodes to manage resources within the node, *eg*, sending requests for more reference values to fill the software cache.

To evaluate different runtime system prototypes, we developed a Mosaic simulator that runs on existing medium-grain multicomputers, including the Cosmic

Cubes, Intel iPSCs, and the Symult 2010. A host program distributes the Mosaic code for a Cantor program to each simulated Mosaic node, and initiates computation by instantiating the *main* process of the Cantor program. Program output is achieved by instantiating a *console* process and passing its reference in messages.

Currently, our simulator is working on a test suite of simple Cantor programs. In the future, we plan to incorporate some of the more recent Cantor innovations, *eg*, functions and limited message discretion, into the simulator and into the runtime system. We are also planning experiments to evaluate different strategies for code distribution and memory allocation throughout Mosaic nodes.

3.2 Concurrent Logic Programming

Stephen Taylor

A commercially supported concurrent logic programming system was ported to our Symult Series 2010 multicomputer, and is available for all users of our project's multicomputers.

This system is composed of a compiler for the language *Strand*, and an environment for program development. The language provides an abstract message-passing framework for use in a variety of symbolic and system integration tasks. The system is also operational on Intel iPSC systems, networks of Suns, Mecho Transputer surfaces, PC Plug-in Transputer cards, Encore/Sequent shared memory machines, BBN Butterfly, and Atari personal machines. The system was used for a graduate course in compiler techniques this quarter, and will be used in a graduate course on concurrent programming in this coming quarter. It is also being used to study various applications in the *composition* research described in the following section of this report. Finally, a textbook describing the ideas embodied in the Strand system was recently completed, and will be published by Prentice-Hall in July 1989.

3.3 Programming with Composition

Mani Chandy, Stephen Taylor

We are interested in developing a notation for specifying concurrent algorithms and programs. Our goals are to support formal reasoning about program correctness and to provide efficient implementations of symbolic, numeric, and operating system codes. We have chosen *program composition* as a central notion due to its prevalence in both semantic models and program design methodologies.

During the past six months, we have considered the basic components of such a notation. Our conclusion is that there are four composition operators of importance. These operators are defined on program units; the method by which these units are implemented is relatively unimportant. It is natural to expect the notation to allow existing codes (written in Fortran, C, Lisp, Ada, *etc*) to be reused on

multicomputers. Moreover, the composition of these units will have a formal semantic characterization. To explore the utility of the notation, we are currently focussing on the hand compilation of non-trivial application codes. If performance results indicate that the notation is sufficiently efficient, we plan to build a compiler targeted to multicomputer architectures.

In the area of numeric computing we are studying a large fluid-flow problem developed in the department of Applied Mathematics at Caltech. This Fortran application computes the transition from a two-dimensional Taylor Vortex to three-dimensional wavy-vortex flow. Central to the application is a relaxation algorithm that employs a multigrid method. After benchmarking, we discovered that more than 70% of the execution time for the application was spent in the relaxation algorithm; thus, we decided to focus on this algorithm. Unfortunately, we arrived at a somewhat negative conclusion: The original algorithm was based on a sequential line-iteration scheme that afforded no opportunity for concurrent execution. As a result, we have converted the original code to use a point Gaussian relaxation algorithm; this appears more suitable. We are currently in the process of debugging a concurrent formulation of the algorithm.

In the area of symbolic computing we are studying a large automated reasoning program in conjunction with the Aerospace Corporation in Los Angeles. This program has been used extensively for checking the correctness of hardware specifications and Ada programs. A central component of the program is a *congruence closure* algorithm used for maintaining equality assertions. We began *this research* by investigating the opportunities for executing portions of this algorithm concurrently. This, again, led us to a somewhat negative conclusion: The granularity of typical invocations of the algorithm is too low to benefit from concurrent execution. We are now investigating a new algorithm that overlaps the execution of multiple equality assertions. Since a large number of these occur in a typical proof, we believe this to be a more suitable direction.

Finally, we are also interested in working with DNA sequencing programs, but have not yet made substantial progress in this area.

It should be understood that the objective of these application efforts is to test the utility of the program-composition notation, rather than to develop the applications themselves.

3.4 Variants of the Chandy-Misra-Bryant Distributed Discrete-Event Simulation Algorithm

Wen-King Su, Chuck Seitz

During the past five months, additional simulations using the new logic simulator have been made, and a revision of the paper "Variants of the Chandy-Misra-Bryant Distributed Discrete-event Simulation Algorithm" (included as an appendix to this report) was written for publication in the 1989 SCS Eastern Multi-Conference. A

test version of the hybrid simulator has been implemented on top of the concurrent CMB variant simulators. Results from this prelimiary investigation are promising, and a new, more efficient version of the hybrid simulator is currently being written.

3.5 Distributed Snapshots

Mani Chandy

One of the fundamental problems in distributed systems appears trivial: Record the state of the system. The problem is, however, quite difficult because distributed systems do not have a single system-wide clock. If there were a clock, all processes could record their local states at a predetermined time. The problem of recording global states of distributed systems is at the core of a large number of problems in distributed systems, including deadlock detection, termination detection, and resource management. The paper, "The Essence of Distributed Snapshots," submitted to the *ACM Transactions on Computer Systems*, and included as an appendix to this report, presents necessary and sufficient conditions for a collection of local snapshots (recordings of local states) to be a global snapshot. The paper shows that many distributed algorithms can be developed in a systematic and straightforward manner from these conditions.

4. VLSI Design

4.1 The Design of the First Asynchronous Microprocessor

Alain J. Martin, Steven M. Burns, T. K. Lee, Drazen Borkovic, Pieter J. Hazewindus

We have completed the design of an entirely asynchronous (self-timed, delay-insensitive) microprocessor. It is a 16-bit, RISC-like architecture with independent instruction and data memories. It has 16 registers, 4 buses, an ALU, and two adders. The size is about 20,000 transistors. Two versions have been fabricated: one in 2μm MOSIS SCMOS, and one in 1.6μm MOSIS SCMOS. (On the 2μm version, only 12 registers were implemented in order to fit the chip into the 84-pin $6600\mu m \times 4600\mu m$ pad frame.)

With the exception of *isochronic forks* (see the paper included as an appendix to this report), the chips are entirely delay-insensitive, *ie*, their correct operation is independent of any assumption on delays in operators and wires except that the delays be finite. The circuits use neither clocks nor knowledge about delays.

The only exception to the design method is the interface with the memories. In the absence of available memories with self-timed interfaces, we have simulated the completion signal from the memories with an external delay. For testing purposes, the delay on the instruction memory interface is variable.

In spite of the presence of several floating n-wells, the 2μm version runs at 12 MIPS. The 1.6μm version runs at 18 MIPS. (Those performance figures are based on measurements from sequences of ALU instructions without carry. They do not take advantage of the overlap between ALU and memory instructions.) Those performance results are quite encouraging given that the design is very conservative: It uses static gates, dual-rail encoding of data, completion trees, *etc.*

Only two of the 12 2μm chips passed all tests, but 34 out of the 50 1.6μm chips were found to be entirely functional. However, within a certain range of values for the instruction memory delay, the 1.6μm version is not entirely functional. We cannot yet explain this phenomenon.

We have tested the chips under a wide range of VDD voltage values. At room temperature, the 2μm version is functional in a voltage range from 7V down to 0.35V! And it reaches 15 MIPS at 7V. We have also tested the chips cooled in liquid nitrogen. The 2μm version reaches 20 MIPS at 5V and 30 MIPS at 12V. The 1.6μm version reaches 30 MIPS at 5V. Of course, these measurements are made without adjusting any clocks (there are none), but simply by connecting the processor to a memory containing a test program and observing the rate of instruction execution. The results are summarized in Figure 1. The power consumption is 145mW at 5V, and 6.7mW at 2V. Figure 2 shows that the optimal *power-delay product* is obtained at 2V at room temperature.

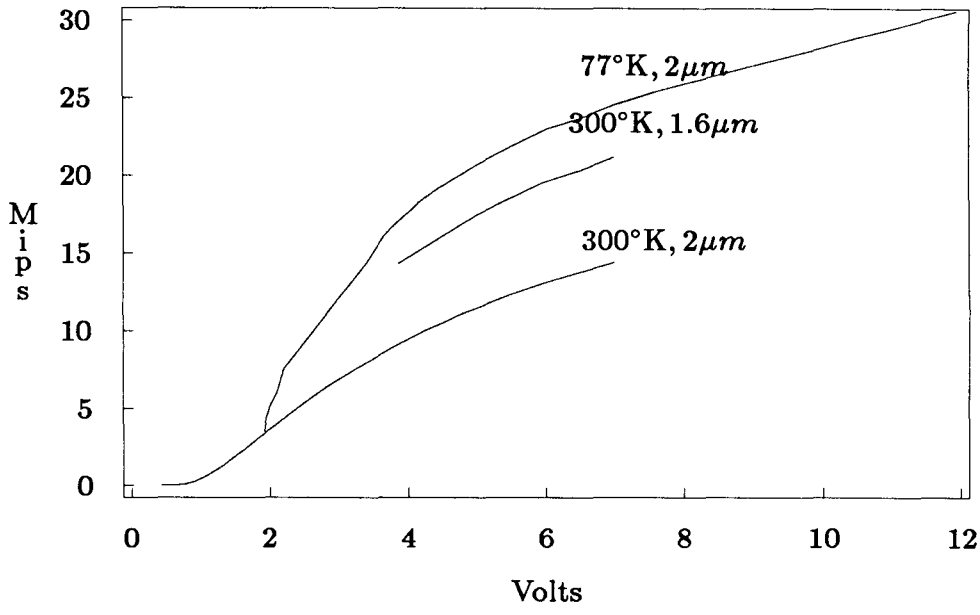


Figure 1: MIPS as a function of VDD

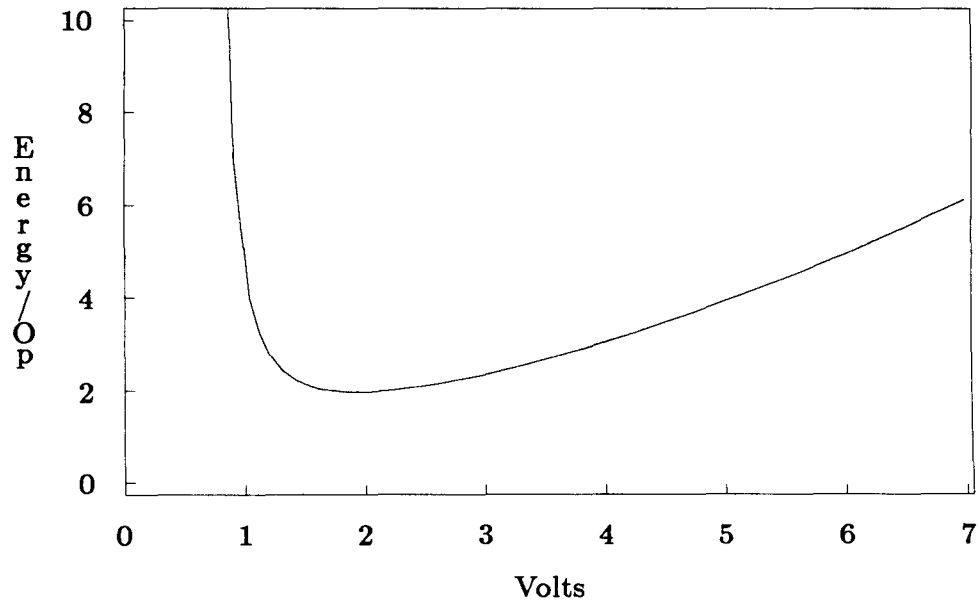


Figure 2: Power-delay product as a function of VDD

4.2 Fast Self-Timed Mesh Routing Chips

Chuck Seitz

The latest mesh-routing-chip (MRC) design, the FMRC2.1 design, was sent to MOSIS for 1.6μm SCMOS fabrication on 7 November 1988. This chip is a revision of FMRC2.0 that corrects a timing error in the latching of a routing decision. A Spice simulation indicated that that the revision corrected a timing error of approximately 0.7ns to a timing margin of about 1.0ns (about 50% of the difference between two short delay paths; hence, not as risky as it may sound). The maximal throughput predicted both by Spice and by tau-model calculations was 60MB/s.

These chips were returned from fabrication on 10 January 1989, and were found to operate correctly under a nearly exhaustive functional screening, and at a maximum throughput of 56MB/s. The yield on this run was 44/50. One of the chips had a cracked package, and two had bonding shorts; hence, the fabrication yield was actually 44/47.

Batches of 20 good chips were sent both to Intel Scientific Computers (as GFE on their DARPA contract) and to Symult Systems, and both companies have verified that these chips operate correctly in their test fixtures or systems.

The FMRC2.1 chip employs a design method that is *not* entirely delay-insensitive (see previous section). The circuit exhibits races *within* modules, but these modules have self-timed interfaces to other modules. Previous MRCs, entirely pin-for-pin compatible, employed the same delay-insensitive style as the asynchronous processor reported in the previous section, and required nearly twice the silicon area to operate half as fast as the FMRC2.1.

Hence, we conjecture that we shall see the same phenomenon with self-timed designs that is apparent with conventional designs; namely, that chips with relatively few cell types, such as memories and MRCs, will profitably employ circuit-level optimizations. Such optimizations are relatively less profitable and manageable in more complex chip designs, such as processors.

4.3 Mosaic C Chip

Jakov Seizovic, Jordan Holt, Chuck Seitz, Don Speck, Wen-King Su, Tony Wittry

During the past few months, work on the Mosaic chip has predominantly consisted of a series of extensive switch-level simulations. Using COSMOS instead of MOSSIM, we were able to decrease the simulation time by a factor of ten, with a negligible additional cost in setup (compile) time. The simulation of a memoryless version of Mosaic chip, consisting of about 26K transistors, takes slightly over a second of real time per clock cycle when running on a SUN 3/260. This has enabled us to simulate fairly long sequences of instructions from the Cantor runtime system at the switch-simulation level.

Having completed simulations of all of the logic parts of the Mosaic chip, *ie*, processor, packet interface, router, and bus arbiter, independently as well as together, we are entering the final phase of switch-level simulations, where multiple Mosaic chips will be represented as processes under CE/RK, and run on the multicomputers operated by the project, as well as on workstations.

We are planning to send the first version of a Mosaic chip to fabrication on a 2μ MOSIS run within a couple of weeks.

4.4 New CMOS PLAs

Jakov Seizovic, Chuck Seitz

A NOR-NOR precharged PLA has been designed to replace the NAND-NOR precharged PLA that we have used extensively since 1985. Both the delay and precharge time of this NOR-NOR PLA are linear in the number of inputs, a significant improvement compared to the NAND-NOR PLA, in which the delay is quadratic, and precharge time is cubic. This PLA has replaced the two NAND-NOR PLAs in the Mosaic C packet interface and the hybrid static/precharge NAND-NOR PLA in the Mosaic processor, and accordingly has saved us a lot of time and trouble in the Mosaic design.

4.5 CIF-flogger

Glenn Lewis, Chuck Seitz

CIF-flogger is a multicomputer program for flattening CIF files, rasterizing the geometry, and performing parallel operations on the geometry in strips. It runs under the CE/RK system, and hence, on most available multicomputers, including the Intel iPSC/2 and Symult Series 2010.

CIF-flogger currently supports the following operations on the chip geometry:

- parsing the CIF specification file (produced by Magic)
- flattening and rasterizing the hierarchical design geometry
- recognizing transistor geometry
- global connected-component labeling
- bloat, shrink, and logical mask layer operations
- creating new CIF for a processed design

Plans for CIF-flogger include:

- general CIF-reading capability
- circuit extraction

- well-plug checking
- design-rule checking

Initial timings indicate that CIF-flogger provides these operations in a matter of a few seconds for 100K-transistor chips. CIF-flogger is intended to be a useful tool for chip designers and foundries to verify that a design passes "syntactical" checks before it is fabricated, thus saving both time and money.

4.6 Adaptive Routing in Multicomputer Networks

John Y. Ngai, Chuck Seitz

As we are wrapping up our theoretical investigation of multicomputer adaptive routing, our recent efforts have been concentrated in two areas:

(1) The first of a series of publications will appear in the 1989 ACM Symposium on Parallel Algorithms and Architectures, to be held in Sante Fe, New Mexico this June. (A copy of this paper is included at the end of the report.)

(2) We have been searching for practical implementation ideas for replacing the existing oblivious router in the Mosaic with an adaptive router. A low-latency header encoding and modification scheme that we have dubbed the "sign-first one-shy code" has been devised for an adaptive router with a relatively narrow flit width. The details of these implementation ideas can be found in a forthcoming PhD thesis.

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

Technical Reports

28 March 1989

Prices include postage and help to defray our printing and mailing costs.

Publication Order Form

To order reports fill out the last page of this publication form. *Prepayment* is required for all materials. Purchase orders will not be accepted. All foreign orders must be paid by international money order or by check drawn on a U.S. bank in U.S. currency, payable to CALTECH.

____CS-TR-89-03 $3.00 *Feature-oriented Image Enhancement with Shock Filters, I*
Rudin, Leonid I with Stanley Osher

____CS-TR-89-02 $3.00 *Design of an Asynchronous Microprocessor*,
Martin, Alain J

____CS-TR-89-01 $4.00 *Programming in VLSI From Communicating Processes to Delay-insensitive Circuits*,
Martin, Alain J

____CS-TR-88-22 $2.00 *Variants of the Chandy-Misra-Bryant Distributed Discrete-Event Simulation Algorithm*,
Su, Wen-King and Charles L Seitz

____CS-TR-88-21 $3.00 *Winner-Take-All Networks of O(N) Complexity*,
Lazzaro, John, with S Ryckebusch, M A Mahowald and C A Mead

____CS-TR-88-20 $7.00 *Neural Network Design and the Complexity of Learning*,
Judd, J. Stephen

____CS-TR-88-19 $5.00 *Controlling Rigid Bodies with Dynamic Constraints*,
Barzel, Ronen

____CS-TR-88-18 $3.00 *Submicron Systems Architecture Project*,
ARPA Semiannual Technical Report

____CS-TR-88-17 $3.00 *Constrained Differential Optimization for Neural Networks*,
Platt, John C and Alan H Barr

____CS-TR-88-16 $3.00 *Programming Parallel Computers*,
Chandy, K Mani

____CS-TR-88-15 $13.00 *Applications of Surface Networks to Sampling Problems in Computer Graphics*, PhD Thesis
Von Herzen, Brian

____CS-TR-88-14 $2.00 *Syntax-directed Translation of Concurrent Programs into Self-timed Circuits*
Burns, Steven M and Alain J Martin

____CS-TR-88-13 $2.00 *A Message-Passing Model for Highly Concurrent Computation*,
Martin, Alain J

____CS-TR-88-12 $4.00 *A Comparison of Strict and Non-strict Semantics for Lists*, MS Thesis
Burch, Jerry R

____CS-TR-88-11 $5.00 *A Study of Fine-Grain Programming Using Cantor*, MS Thesis
Boden, Nanette J

____CS-TR-88-10 $3.00 *The Reactive Kernel*, MS Thesis
Seizovic, Jacov

____CS-TR-88-07 $3.00 *The Hexagonal Resistive Network and the Circular Approximation*,
Feinstein, David I

____CS-TR-88-06 $3.00 *Theorems on Computations of Distributed Systems*,
Chandy, K Mani

____CS-TR-88-05 $3.00 *Submicron Systems Architecture*
ARPA Semiannual Technical Report

____CS-TR-88-04 $3.00 *Cochlear Hydrodynamics Demystified*
Lyon, Richard F and Carver A Mead

____CS-TR-88-03 $4.00 *PS: Polygon Streams: A Distributed Architecture for Incremental Computation Applied to Graphic*
MS Thesis
Gupta, Rajiv

____CS-TR-88-02 $4.00 *Automated Compilation of Concurrent Programs into Self-timed Circuits*, MS Thesis
Steven M Burns

____CS-TR-88-01 $3.00 *C Programmer's Abbreviated Guide to Multicomputer Programming*,
Seitz, Charles, Jakov Seizovic and Wen-King Su

____5258:TR:88 $3.00 *Submicron Systems Architecture*
ARPA Semiannual Technical Report

____5256:TR:87 $2.00 *Synthesis Method for Self-timed VLSI Circuits*,
Martin, Alain
current supply only: see *Proc. ICCD'87: 1987 IEEE Int'l. Conf. on Computer Design*, 224-229, C

____5253:TR:88 $2.00 *Synthesis of Self-Timed Circuits by Program Transformation*,
Burns, Steven M and Alain J Martin

____5251:TR:87 $2.00 *Conditional Knowledge as a Basis for Distributed Simulation*,
Chandy, K. Mani and Jay Misra

____5250:TR:87 $10.00 *Images, Numerical Analysis of Singularities and Shock Filters*, PhD Thesis
Rudin, Leonid Iakov

____5249:TR:87 $6.00 *Logic from Programming Language Semantics*, PhD Thesis
Choo, Young-il

____5247:TR:87 $6.00 *VLSI Concurrent Computation for Music Synthesis*, PhD Thesis
Wawrzynek, John

____5246:TR:87 $3.00 *Framework for Adaptive Routing*
Ngai, John Y and Charles L. Seitz

____5244:TR:87 $3.00 *Multicomputers*
Athas, William C and Charles L Seitz

____5243:TR:87 $5.00 *Resource-Bounded Category and Measure in Exponential Complexity Classes*, PhD Thesis
Lutz, Jack H

____5242:TR:87 $8.00 *Fine Grain Concurrent Computations*, PhD Thesis
Athas, William C.

____5241:TR:87 $3.00 *VLSI Mesh Routing Systems*, MS Thesis
Flaig, Charles M

____5240:TR:87 $2.00 *Submicron Systems Architecture*
ARPA Semiannual Technical Report

____5239:TR:87 $3.00 *Trace Theory and Systolic Computations*
Rem, Martin

____5238:TR:87 $7.00 *Incorporating Time in the New World of Computing System*, MS Thesis
Poh, Hean Lee

____5236:TR:86 $4.00 *Approach to Concurrent Semantics Using Complete Traces*, MS Thesis
Van Horn, Kevin S.

____5235:TR:86 $4.00 *Submicron Systems Architecture*
ARPA Semiannual Technical Report

____5234:TR:86 $3.00 *High Performance Implementation of Prolog*
Newton, Michael O

____5233:TR:86 $3.00 *Some Results on Kolmogorov-Chaitin Complexity*, MS Thesis
Schweizer, David Lawrence

____5232:TR:86 $4.00 *Cantor User Report*
Athas, W.C. and C. L. Seitz

____5230:TR:86 $24.00 *Monte Carlo Methods for 2-D Compaction*, PhD Thesis
Mosteller, R.C.

____5229:TR:86 $4.00 *anaLOG - A Functional Simulator for VLSI Neural Systems*, MS Thesis
Lazzaro, John

____5228:TR:86 $3.00 *On Performance of k-ary n-cube Interconection Networks*,
Dally, Wm. J

____5227:TR:86 $18.00 *Parallel Execution Model for Logic Programming*, PhD Thesis
Li, Pey-yun Peggy

____5223:TR:86 $15.00 *Integrated Optical Motion Detection*, PhD Thesis
Tanner, John E.

____5221:TR:86 $3.00 *Sync Model: A Parallel Execution Method for Logic Programming*
Li, Pey-yun Peggy and Alain J. Martin
current supply only: see *Proc SLP'86 3rd IEEE Symp on Logic Programming Sept '86*

____5220:TR:86 $4.00 *Submicron Systems Architecture*
ARPA Semiannual Technical Report

____5215:TR:86 $2.00 *How to Get a Large Natural Language System into a Personal Computer*,
Thompson, Bozena H. and Frederick B. Thompson

____5214:TR:86 $2.00 *ASK is Transportable in Half a Dozen Ways*,
Thompson, Bozena H. and Frederick B. Thompson

____5212:TR:86 $2.00 *On Seitz' Arbiter*,
Martin, Alain J

____5210:TR:86 $2.00 *Compiling Communicating Processes into Delay-Insensitive VLSI Circuits*,
Martin, Alain
current supply only: see *Distributed Computing* v 1 no 4 (1986)

____5207:TR:86 $2.00 *Complete and Infinite Traces: A Descriptive Model of Computing Agents*,
van Horn, Kevin

____5205:TR:85 $2.00 *Two Theorems on Time Bounded Kolmogrov-Chaitin Complexity*,
Schweizer, David and Yaser Abu-Mostafa

____5204:TR:85 $3.00 *An Inverse Limit Construction of a Domain of Infinite Lists*,
Choo, Young-Il

____5202:TR:85 $15.00 *Submicron Systems Architecture*,
ARPA Semiannual Technical Report

____5200:TR:85 $18.00 *ANIMAC: A Multiprocessor Architecture for Real-Time Computer Animation*, PhD thesis
Whelan, Dan

____5198:TR:85 $8.00 *Neural Networks, Pattern Recognition and Fingerprint Hallucination*, PhD thesis
Mjolsness, Eric

____5197:TR:85 $7.00 *Sequential Threshold Circuits*, MS thesis
Platt, John

____5195:TR:85 $3.00 *New Generalization of Dekker's Algorithm for Mutual Exclusion*,
Martin, Alain J
current supply only: see *Information Processing Letters*, **23**, 295-297 (1986)

____5194:TR:85 $5.00 *Sneptree - A Versatile Interconnection Network*,
Li, Pey-yun Peggy and Alain J Martin

____5193:TR:85 $2.00 *Delay-insensitive Fair Arbiter*
Martin, Alain J

____5190:TR:85 $3.00 *Concurrency Algebra and Petri Nets*,
Choo, Young-il

____5189:TR:85 $10.00 *Hierarchical Composition of VLSI Circuits*, PhD Thesis
Whitney, Telle

____5185:TR:85 $11.00 *Combining Computation with Geometry*, PhD Thesis
Lien, Sheue-Ling

____5184:TR:85 $7.00 *Placement of Communicating Processes on Multiprocessor Networks*, MS Thesis
Steele, Craig

____5179:TR:85 $3.00 *Sampling Deformed, Intersecting Surfaces with Quadtrees*, MS Thesis,
Von Herzen, Brian P.

____5178:TR:85 $9.00 *Submicron Systems Architecture*,
ARPA Semiannual Technical Report

____5174:TR:85 $7.00 *Balanced Cube: A Concurrent Data Structure*,
Dally, William J and Charles L Seitz

____5172:TR:85 $6.00 *Combined Logical and Functional Programming Language*,
Newton, Michael

____5168:TR:84 $3.00 *Object Oriented Architecture*,
Dally, Bill and Jim Kajiya

____5165:TR:84 $4.00 *Customizing One's Own Interface Using English as Primary Language*,
Thompson, B H and Frederick B Thompson

____5164:TR:84 $13.00 *ASK French - A French Natural Language Syntax*, MS Thesis
Sanouillet, Remy

____5160:TR:84 $7.00 *Submicron Systems Architecture*,
ARPA Semiannual Technical Report

____5158:TR:84 $6.00 *VLSI Architecture for Sound Synthesis*,
Wawrzynek, John and Carver Mead

____5157:TR:84 $15.00 *Bit-Serial Reed-Solomon Decoders in VLSI*, PhD Thesis
Whiting, Douglas

____5147:TR:84 $4.00 *Networks of Machines for Distributed Recursive Computations*,
Martin, Alain and Jan van de Snepscheut

____5143:TR:84 $5.00 *General Interconnect Problem*, MS Thesis
Ngai, John

____5140:TR:84 $5.00 *Hierarchy of Graph Isomorphism Testing*, MS Thesis
Chen, Wen-Chi

____5139:TR:84 $4.00 *HEX: A Hierarchical Circuit Extractor*, MS Thesis
Oyang, Yen-Jen

____5137:TR:84 $7.00 *Dialogue Designing Dialogue System*, PhD Thesis
Ho, Tai-Ping

____5136:TR:84 $5.00 *Heterogeneous Data Base Access*, PhD Thesis
Papachristidis, Alex

____5135:TR:84 $7.00 *Toward Concurrent Arithmetic*, MS Thesis
Chiang, Chao-Lin

____5134:TR:84 $2.00 *Using Logic Programming for Compiling APL*, MS Thesis
Derby, Howard

____5133:TR:84 $13.00 *Hierarchical Timing Simulation Model for Digital Integrated Circuits and Systems*, PhD Thesi
Lin, Tsu-mu

____5132:TR:84 $10.00 *Switch Level Fault Simulation of MOS Digital Circuits*, MS Thesis
Schuster, Mike

____5129:TR:84 $5.00 *Design of the MOSAIC Processor*, MS Thesis
Lutz, Chris

____5128:TM:84 $3.00 *Linguistic Analysis of Natural Language Communication with Computers*,
Thompson, Bozena H

____5125:TR:84 $6.00 *Supermesh*, MS Thesis
Su, Wen-king

____5123:TR:84 $14.00 *Mossim Simulation Engine Architecture and Design*,
Dally, Bill

____5122:TR:84 $8.00 *Submicron Systems Architecture*,
ARPA Semiannual Technical Report

____5114:TM:84 $3.00 *ASK As Window to the World*,
Thompson, Bozena, and Fred Thompson

____5112:TR:83 $22.00 *Parallel Machines for Computer Graphics*, PhD Thesis
Ulner, Michael

____5106:TM:83 $1.00 *Ray Tracing Parametric Patches*,
Kajiya, James T

____5104:TR:83 $9.00 *Graph Model and the Embedding of MOS Circuits*, MS Thesis
Ng, Tak-Kwong

____5094:TR:83 $2.00 *Stochastic Estimation of Channel Routing Track Demand*,
Ngai, John

____5092:TM:83 $2.00 *Residue Arithmetic and VLSI*,
Chiang, Chao-Lin and Lennart Johnsson

____5091:TR:83 $2.00 *Race Detection in MOS Circuits by Ternary Simulation*,
Bryant, Randal E

____5090:TR:83 $9.00 *Space-Time Algorithms: Semantics and Methodology*, PhD Thesis
Chen, Marina Chien-mei

____5089:TR:83 $10.00 *Signal Delay in General RC Networks with Application to Timing Simulation of Digital Integrated Circuits*,
Lin, Tzu-Mu and Carver A Mead

____5086:TR:83 $4.00 *VLSI Combinator Reduction Engine*, MS Thesis
Athas, William C Jr

____5082:TR:83 $10.00 *Hardware Support for Advanced Data Management Systems*, PhD Thesis
Neches, Philip

____5081:TR:83 $4.00 *RTsim - A Register Transfer Simulator*, MS Thesis
Lam, Jimmy
current supply only: see *Acta Informatica* **20**, 301-313, (1983)

____5074:TR:83 $10.00 *Robust Sentence Analysis and Habitability*,
Trawick, David

____5073:TR:83 $12.00 *Automated Performance Optimization of Custom Integrated Circuits*, PhD Thesis
Trimberger, Steve

____5065:TR:82 $3.00 *Switch Level Model and Simulator for MOS Digital Systems*,
Bryant, Randal E

____5054:TM:82 $3.00 *Introducing ASK, A Simple Knowledgeable System*, Conf on App'l Natural Language Processing
Thompson, Bozena H and Frederick B Thompson

____5051:TM:82 $2.00 *Knowledgeable Contexts for User Interaction*, Proc Nat'l Computer Conference
Thompson, Bozena, Frederick B Thompson, and Tai-Ping Ho

____5035:TR:82 $9.00 *Type Inference in a Declarationless, Object-Oriented Language*, MS Thesis
Holstege, Eric

____5034:TR:82 $12.00 *Hybrid Processing*, PhD Thesis
Carroll, Chris

____5033:TR:82 $4.00 *MOSSIM II: A Switch-Level Simulator for MOS LSI User's Manual*,
Schuster, Mike, Randal Bryant and Doug Whiting

____5029:TM:82 $4.00 *POOH User's Manual*,
Whitney, Telle

____5018:TM:82 $2.00 *Filtering High Quality Text for Display on Raster Scan Devices*,
Kajiya, Jim and Mike Ullner

____5017:TM:82 $2.00 *Ray Tracing Parametric Patches*,
Kajiya, Jim

____5015:TR:82 $15.00 *VLSI Computational Structures Applied to Fingerprint Image Analysis*,
Megdal, Barry

____5014:TR:82 $15.00 *Extension of Object-Oriented Languages to a Homogeneous, Concurrent Architecture*, PhD Thesis
Lang, Charles R Jr

____5012:TM:82 $2.00 *Switch-Level Modeling of MOS Digital Circuits*,
Bryant, Randal

____5000:TR:82 $6.00 *Self-Timed Chip Set for Multiprocessor Communication*, MS Thesis
Whiting, Douglas

____4684:TR:82 $3.00 *Characterization of Deadlock Free Resource Contentions*,
Chen, Marina, Martin Rem, and Ronald Graham

____4655:TR:81 $20.00 *Proc Second Caltech Conf on VLSI*,
Seitz, Charles, ed.

____3760:TR:80 $10.00 *Tree Machine: A Highly Concurrent Computing Environment*, PhD Thesis
Browning, Sally

____3759:TR:80 $10.00 *Homogeneous Machine*, PhD Thesis
Locanthi, Bart

____3710:TR:80 $10.00 *Understanding Hierarchical Design*, PhD Thesis
Rowson, James

____3340:TR:79 $26.00 *Proc. Caltech Conference on VLSI (1979)*,
Seitz, Charles, ed

____2276:TM:78 $12.00 *Language Processor and a Sample Language*,
Ayres, Ron

Caltech Computer Science Technical Reports

Please PRINT your name, address and amount enclosed below:

name _____

Address _____

City _____ State _____ Zip _____ Country _____

Amount enclosed $ _____

_____ Please check here if you wish to be included on our mailing list

_____ Please check here for any change of address

_____ Please check here if you would prefer to have future publications lists sent to your e-mail address.

E-mail address _____

Return this form to: Computer Science Library, 256-80, Caltech, Pasadena CA 91125

_____89-03	_____5250	_____5210	_____5143	_____5074
_____89-02	_____5249	_____5207	_____5140	_____5073
_____89-01	_____5247	_____5205	_____5139	_____5065
_____88-22	_____5246	_____5204	_____5137	_____5054
_____88-21	_____5244	_____5202	_____5136	_____5051
_____88-20	_____5243	_____5200	_____5135	_____5035
_____88-19	_____5242	_____5198	_____5134	_____5034
_____88-18	_____5241	_____5197	_____5133	_____5033
_____88-17	_____5240	_____5195	_____5132	_____5029
_____88-16	_____5239	_____5194	_____5129	_____5018
_____88-15	_____5238	_____5193	_____5128	_____5017
_____88-14	_____5236	_____5190	_____5125	_____5015
_____88-13	_____5235	_____5189	_____5123	_____5014
_____88-12	_____5234	_____5185	_____5122	_____5012
_____88-11	_____5233	_____5184	_____5114	_____5000
_____88-10	_____5232	_____5179	_____5112	_____4684
_____88-07	_____5230	_____5178	_____5106	_____4655
_____88-06	_____5229	_____5174	_____5104	_____3760
_____88-05	_____5228	_____5172	_____5094	_____3759
_____88-04	_____5227	_____5168	_____5092	_____3710
_____88-03	_____5223	_____5165	_____5091	_____3340
_____88-01	_____5221	_____5164	_____5090	_____2276
_____5258	_____5220	_____5160	_____5089	_____....
_____5256	_____5215	_____5158	_____5086	_____....
_____5253	_____5214	_____5157	_____5082	_____....
_____5251	_____5212	_____5147	_____5081	_____....

The Design of an Asynchronous Microprocessor

Alain J. Martin, Steven M. Burns, T.K. Lee,
Drazen Borkovic, Pieter J. Hazewindus

California Institute of Technology
Pasadena CA 91125, USA

to appear in *Proc. Decennial Caltech Conference on VLSI*, 20-22 March, 1989, MIT Press
Caltech-CS-TR-89-2

1 Introduction

Prejudices are as tenacious in science and engineering as in any other human activity. One of the most firmly held prejudices in digital VLSI design is that asynchronous circuits—a.k.a. self-timed or delay-insensitive circuits—are necessarily slow and wasteful in area and logic. Whereas asynchronous techniques would be appropriate for control, they would be inadequate for data paths because of the cost of dual-rail encoding of data, the cost of generating completion signals for write operations on registers, and the difficulty of designing self-timed buses.

Because a general-purpose microprocessor contains a complex data path, a corollary of the previous opinion is that it is impossible to design an efficient asynchronous microprocessor. Since we have been developing a design method for asynchronous circuits that gives excellent results, and since the above objections to large-scale *data* path designs are genuine but untested, we decided to "pick up the gauntlet" and design a complete processor.

The design of an asynchronous microprocessor poses new challenges and opens new avenues to the computer architect. Hence, the experiment unavoidably developed a dual purpose: We are refining an already well-tested design method, and we are starting a new series of experiments in asynchronous architectures. (As far as we know, this is the first entirely asynchronous microprocessor ever built.) The results we are reporting have a different implication depending on whether they are related to the first or second goal of the experiment. Whereas we are convinced that our design methods have reached maturity, we are quite aware that asynchronous techniques may influence the computer architects in completely new ways that this first design is just starting to explore.

In order to focus the experiment on asynchronous circuit design, we have intentionally excluded optimizations at the high and low ends of the design process. The instruction set is straightforward and no assumption has been made on the code produced by the compiler. No special electrical optimizations other than transistor sizing have been applied; the circuit techniques rarely go beyond those taught in a graduate-level VLSI class, and, apart from the memory interfaces, the circuits are ***delay-insensitive***. Hence, any performance is to be attributed to the design method and to the inherent advantages of asynchronous design.

A circuit is delay-insensitive when its correct operation is independent of any assumption on delays in operators and wires except that the delays be finite. Such circuits do not use a clock signal or knowledge about delays: Sequencing is enforced entirely by communication mechanisms.

The class of entirely delay-insensitive circuits is very limited. Different asynchronous techniques distinguish themselves in the choice of the compromises to delay-insensitivity. ***Speed-independent*** techniques assume that delays in gates are arbitrary, but there are no delays in wires. ***Self-timed*** techniques assume that a circuit can be decomposed into ***equipotential*** regions inside which delays in wires are negligible[11].

In our method, certain local forks are introduced to distribute a variable as inputs of several gates. We assume that the difference between the delays in the branches of such forks are short compared to delays in other gates. We call such forks ***isochronic***[6], [8].

The general method—a complete description of which can be found in the referenced papers [2], [5], [6], [7], [8]—is based on program transformations. The circuit is first designed as a set of concurrent programs. Each program is then compiled (manually or automatically) into a circuit by applying a series of program transformations. Control and data path are first designed separately and then combined in a mechanical way. This important ***divide-and-conquer*** technique is a main innovation of the method.

2 Preliminary Results

As of this writing, the first design is complete, and has been scheduled for fabrication in $2\mu m$ MOSIS SCMOS. The chip was functionally simulated using COSMOS [1], and was found to be functionally correct.

The architecture is a 16-bit processor with offset and a simple instruction set of the RISC type [4]. The data path contains twelve 16-bit registers, four buses, an ALU, and two adders. The chip contains 20,000 transistors and fits within a 5500λ by 3500λ area. We are using an 84-pin $6600\mu m \times 4600\mu m$ frame. An estimate of the critical path suggests processor performance of approximately 15MIPS in $2\mu m$ SCMOS. (A slightly improved $1.6\mu m$ SCMOS version is also being fabricated.)

This experiment, the most challenging one we have conducted so far, promised to be an important test for our method. The results obtained so far have been very encouraging.

The technique for separating control and data path has been extended with a novel asynchronous bus design, and is now robust and general.

The handshaking protocol between circuit elements has also been modified so that half of a protocol sequence overlaps subsequent actions. This protocol makes it possible to "hide" half of delays of the completion trees, the tree of gates that combine the completion signals from the asynchronous elements. In addition, at most two completion trees are in sequence on any path. Thus, completion tree delays are not a serious disadvantage of asynchronous design.

Instruction pipelining has been approached as a concurrent programming problem: Starting with a sequential program for the processor, concurrency is introduced through a series of program transformations. However, although the transformations are guided by the intent to overlap the important phases—fetch, decode, execute—of instruction execution, they are neither mechanical nor unique. The designer decides how to decompose a program into several concurrent ones. We do not claim that our solution in this first design is in any way optimal.

3 Specification of the Processor as a Sequential Program

The instruction set is deliberately not innovative. It is a conventional 16-bit-word instruction set of the ***load-store*** type. The processor uses two separate memories for instructions and data. There are three types of instructions: ALU, memory, and program-counter (***pc***). All ALU instructions operate on registers; memory instructions involve a register and a data memory word. Certain instructions use the following word as ***offset***. (See Table 1 in Appendix 2.)

```
*[FETCH : i, pc := imem[pc], pc + 1;
          [offset(i.op) → offset, pc := imem[pc], pc + 1;
          ||¬offset(i.op) → skip
          ];
  EXECUTE : [alu(i.op) → ⟨reg[i.z], f⟩ := aluf(reg[i.x], reg[i.y], i.op, f)
            |ld(i.op) → reg[i.z] := dmem[reg[i.x] + reg[i.y]]
            |st(i.op) → dmem[reg[i.x] + reg[i.y]] := reg[i.z]
            |ldx(i.op) → reg[i.z] := dmem[offset + reg[i.y]]
            |stx(i.op) → dmem[offset + reg[i.y]] := reg[i.z]
            |lda(i.op) → reg[i.z] := offset + reg[i.y]
            |stpc(i.op) → reg[i.z] := pc
            |jmp(i.op) → pc := reg[i.y]
            |brch(i.op) → [cond(f, i.cc) → pc := pc + offset
                          |¬cond(f, i.cc) → skip
                          ]
            ]
].
```

Figure 1: Sequential program describing the processor

The only important omissions, those of an interrupt mechanism and communication ports, are ones we found to be unnecessary distractions in a first design.

The sequential program describing the processor is a non-terminating loop, each step of which is a *FETCH* phase followed by an *EXECUTE* phase. The complete sequential program for the processor is shown in Figure 1. (The notation, which is an extension of the one we have used in previous work, is described in Appendix 1.) Variable i, which contains the instruction currently being executed, is described in the PASCAL record notation as a structured variable consisting of

several fields. All instructions contain an ***op*** field for the ***opcode***. The parameter fields depend on the types of the instructions, which are found in Table 2 in Appendix 2. The most common ones, those for ALU, load, and store instructions, consist of the three parameters, x, y, and z. Variable ***cc*** contains the condition code field of the branch instruction, and f contains the ***flags*** generated by the execution of an ***alu*** instruction.

The two memories are the arrays ***imem*** and ***dmem***. The index to ***imem*** is the program-counter variable, ***pc***. The general-purpose registers are described as the array ***reg***[0...15]. (Only twelve registers are implemented in the first chip.) Register ***reg***[0] is special: It always contains the value zero.

4 Decomposition into Concurrent Processes

We decompose the previous program into a set of concurrent processes that communicate and synchronize using communication commands on channels. A restricted form of shared variables is allowed. The control channels ***Xs***, ***Ys***, ***ZAs***, ***ZWs***, ***ZRs***, and the bus ***ZA*** are one-to-many; the buses ***X***, ***Y***, ***ZM*** are many-to-many; the other channels are one-to-one. But all channels are used by only two processes at a time. The structure of processes and channels is shown in Figure 2. The final program is shown in Figures 3 and 4.

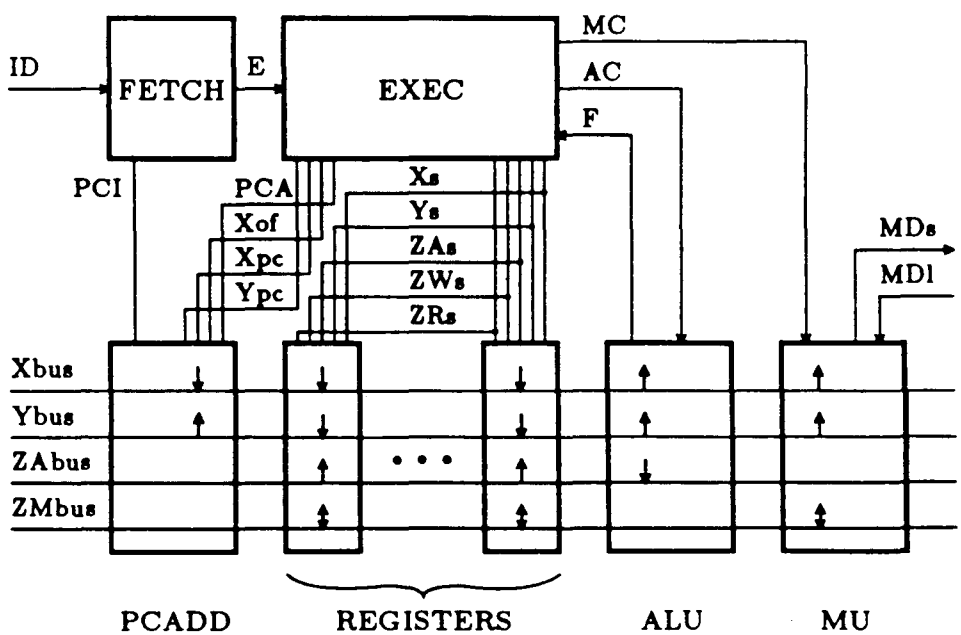


Figure 2: Process and channel structure

$$
\begin{aligned}
IMEM &\equiv *[ID!imem[pc]] \\
FETCH &\equiv *[PCI1; ID?i; PCI2; \\
&\quad [offset(i.op) \rightarrow PCI1; ID?offset; PCI2 \\
&\quad \|\neg offset(i.op) \rightarrow skip \\
&\quad]; E1!i; E2 \\
&\quad] \\
PCADD &\equiv (*[[\overline{PCI1} \rightarrow PCI1; y := pc + 1; PCI2; pc := y \\
&\quad \|\overline{PCA1} \rightarrow PCA1; y := pc + offset; PCA2; pc := y \\
&\quad \|\overline{Xpc} \rightarrow X!pc \bullet Xpc \\
&\quad \|\overline{Ypc} \rightarrow Y?pc \bullet Ypc \\
&\quad]] \\
&\quad \|*[[\overline{Xof} \rightarrow X!offset \bullet Xof]] \\
&\quad) \\
EXEC &\equiv *[E1?i; \\
&\quad [alu(i.op) \rightarrow E2; Xs \bullet Ys \bullet AC!i.op \bullet ZAs \\
&\quad \|ld(i.op) \rightarrow E2; Xs \bullet Ys \bullet MC1 \bullet ZRs \\
&\quad \|st(i.op) \rightarrow E2; Xs \bullet Ys \bullet MC2 \bullet ZWs \\
&\quad \|ldx(i.op) \rightarrow Xof \bullet Ys \bullet MC1 \bullet ZRs; E2 \\
&\quad \|stx(i.op) \rightarrow Xof \bullet Ys \bullet MC2 \bullet ZWs; E2 \\
&\quad \|lda(i.op) \rightarrow Xof \bullet Ys \bullet MC3 \bullet ZRs; E2 \\
&\quad \|stpc(i.op) \rightarrow Xpc \bullet Ys \bullet AC!add \bullet ZAs; E2 \\
&\quad \|jmp(i.op) \rightarrow Ypc \bullet Ys; E2 \\
&\quad \|brch(i.op) \rightarrow F?f; [cond(f, i.cc) \rightarrow PCA1; PCA2 \\
&\qquad\qquad\qquad\qquad \|\neg cond(f, i.cc) \rightarrow skip \\
&\qquad\qquad\qquad\qquad]; E2 \\
&\quad]
\end{aligned}
$$

Figure 3: The final program, first part

$$
\begin{aligned}
ALU \equiv *[[&\overline{AC} \rightarrow AC?op \bullet X?x \bullet Y?y; \\
&\qquad \langle z, f \rangle := aluf(x, y, op, f); ZA!z \\
&[]\overline{F} \rightarrow F!f \\
&]] \\
MU \equiv *[[&\overline{MC1} \rightarrow X?x \bullet Y?y \bullet MC1; ma := x + y; MDl?w; ZM!w \\
&[]\overline{MC2} \rightarrow X?x \bullet Y?y \bullet MC2 \bullet ZM?w; ma := x + y; MDs!w \\
&[]\overline{MC3} \rightarrow X?x \bullet Y?y \bullet MC3; ma := x + y; ZM!ma \\
&]] \\
DMEM \equiv *[[&\overline{MDl} \rightarrow MDl!dmem[ma] \\
&\quad []\overline{MDs} \rightarrow MDs?dmem[ma] \\
&]] \\
REG[k] \equiv (*[[&\neg bk \wedge k = i.x \wedge \overline{Xs} \rightarrow X!r \bullet Xs]] \\
&\|*[[\neg bk \wedge k = i.y \wedge \overline{Ys} \rightarrow Y!r \bullet Ys]] \\
&\|*[[\neg bk \wedge k = i.z \wedge \overline{ZWs} \rightarrow ZM!r \bullet ZWs]] \\
&\|*[[\neg bk \wedge k = i.z \wedge \overline{ZAs} \rightarrow bk\uparrow; ZAs; ZA?r; bk\downarrow]] \\
&\|*[[\neg bk \wedge k = i.z \wedge \overline{ZRs} \rightarrow bk\uparrow; ZRs; ZM?r; bk\downarrow]] \\
&)
\end{aligned}
$$

Figure 4: The final program, second part

Process *FETCH* fetches the instructions from the instruction memory, and transmits them to process *EXEC* which decodes them. Process *PCADD* updates the address **pc** of the next instruction concurrently with the instruction fetch, and controls the ***offset*** register. The execution of an ALU instruction by process *ALU* can overlap with the execution of a memory instruction by process *MU*. The ***jump*** and ***branch*** instructions are executed by *EXEC*; ***store-pc*** is executed by the ALU as the instruction "add the content of register **r** to the pc and store it." The array *REG*[**k**] of processes implements the register file. Both *MU* and *PCADD* contain their own adder. Processes *IMEM* and *DMEM* describe the instruction memory and data memory, respectively.

Updating the PC

The variable *pc* is updated by process *PCADD*, and is used by *IMEM* as the index of the array *imem* during the *ID* communication—the instruction fetch.

The assignment $pc := pc+1$ is decomposed into $y := pc+1; pc := y$, where y is a local variable of *PCADD* . The overlap of the instruction fetch, *ID*? (either *ID*?*i* or *ID*?*offset*), and the pc increment, $y := pc + 1$, can now occur while *pc* is constant. Action *ID*? is enclosed between the two communication actions *PCI1* and *PCI2*, as follows:

$$PCI1; ID?i; PCI2 .$$

In *PCADD*, $y := pc + 1$ is enclosed between the same two communication actions while the updating of *pc* follows *PCI2*:

$$\overline{PCI1} \rightarrow PCI1; y := pc + 1; PCI2; pc := y .$$

Since the completions of *PCI1* and *PCI2* in *FETCH* coincide with the completion of *PCI1* and *PCI2* in *PCADD*, respectively, the execution of *ID*?*i* in *FETCH* overlaps the execution of $y := pc + 1$ in *PCADD*. *PCI1* and *PCI2* are implemented as the two halves of the same communication handshaking to minimize the overhead.

In order to concentrate all increments of *pc* inside *PCADD*, we use the same technique to delegate the assignment $pc := pc + offset$ (executed by the *EXEC* part in the sequential program) to *PCADD*.

The guarded command $\overline{Xof} \rightarrow Xof!offset$ in *PCADD* has been transformed into a concurrent process since it needs only be mutually exclusive with assignment $y := x + offset$, and this mutual exclusion is enforced by the sequencing between *PCA1*; *PCA2* and *Xof* within *EXEC*.

5 Stalling the Pipeline

When the pc is modified by *EXEC* as part of the execution of a *pc* instruction, (*store-pc*, *jump* or *branch*), fetching the next instruction by *FETCH* is postponed until the correct value of the pc is assigned to *PCADD.pc*.

When the offset is reserved for *MU* by *EXEC*, as part of the execution of some memory instructions, fetching the next instruction, which might be a new offset, is postponed until *MU* has received the

value of the current offset. In the second design, we have refined the protocol to block ***FETCH*** only when the next instruction is a new offset.

Postponing the start of the next cycle in ***FETCH*** is achieved by postponing the completion of the previous cycle, i.e., by postponing the completion of the communication action on channel E. As in the case of the ***PCI*** communication, E is decomposed into two communications, $E1$ and $E2$. Again, $E1$ and $E2$ are implemented as the two halves of the same handshaking protocol.

In ***FETCH***, $E!i$ is replaced with $E1!i; E2$. In ***EXEC***, $E2$ is postponed until after either $Xof?offset$ or a complete execution of a ***pc*** instruction has occurred.

6 Sharing Registers and Buses

A bus is used by two processes at a time, one of which is a register and the other is ***EXEC***, ***MU***, ***ALU***, or ***PCADD***. We therefore decided to introduce enough buses so as not to restrict the concurrent access to different registers. For instance, ***ALU*** writing a result into a register should not prevent ***MU*** from using another register at the same time.

The four buses correspond to the four main concurrent activities involving the registers.

The X bus and the Y bus are used to send the parameters of an ALU operation to the ALU, and to send the parameters of address calculation to the memory unit. We also make opportunistic use of them to transmit the pc and the offset to and from ***PCADD***.

The ZA bus is used to transmit the result of an ALU operation to the registers. The ZM bus is used by the memory unit to transmit data between the data memory and the registers.

We make a virtue out of necessity by turning the restriction that registers can be accessed only through those four buses into a convenient abstraction mechanism. The ALU uses only the X, Y, and ZA ports without having to reference the particular registers that are used in the communications. It is the task of ***EXEC*** to reserve the X, Y, and ZA bus for the proper registers before the ALU uses them.

The same holds for the ***MU*** process, which references only X, Y, and ZM. An additional abstraction is that the X bus is used to send the offset to ***MU***, so that the cases for which the first parameter is $i.x$ or $offset$ are now identical, since both parameters are sent via the X bus.

Exclusive Use of a Bus

Commands ***Xpc***, ***Ypc***, and ***Xof*** are used by ***EXEC*** to select the X and Y buses for communication of ***pc*** and ***offset***. Commands ***Xs***, ***Ys***, and ***ZAs*** are used by ***EXEC*** to select the X, Y, and ZA buses, respectively, for a register that has to communicate with the ALU as part of the execution of an ALU instruction.

Two commands are needed to select the ZM bus: ***ZWs*** if the bus is to be used for writing to the data memory, and ***ZRs*** if the bus is to be used for reading from the data memory.

Let us first solve the problem of the mutual exclusion among the different uses of a bus. As long as we have only one ALU and one memory unit, no conflict is possible on the ZA and ZM buses, since only the ALU uses the ZA bus, and only the memory unit uses the ZM bus. But the X and Y buses are used concurrently by the ALU, the memory unit, and the pc unit.

We achieve mutual exclusion on different uses of the X bus as follows. (The same argument holds for Y.) The completion of an X communication is made to coincide with the completion of one of the selection actions ***Xs***, ***Xof***, ***Xpc***; and the occurrences of these selection actions exclude each other in time inside ***EXEC*** since they appear in different guarded commands.

This coincidence is implemented by the ***bullet*** ($\bullet$) command : For arbitrary communication commands U and V inside the same process, $U \bullet V$ guarantees that the two actions are completed at the same time. We then say that the two actions coincide. The use of the bullets ***X!pc*** $\bullet$ ***Xpc*** and ***X!offset*** $\bullet$ ***Xof*** inside ***PCADD*** , and ***X!r*** $\bullet$ ***Xs*** inside the registers enforce the coinidence of X with ***Xpc***, ***Xof***, and ***Xs***, respectively. The bullets in ***EXEC***, ***ALU***, and ***MU*** have been introduced for reasons of efficiency: Sequencing is avoided.

7 Register Selection

Command ***Xs*** in ***EXEC*** selects the X bus for the particular register whose index k is equal to the field $i.x$ of the instruction i being decoded by ***EXEC***, and analogously for commands ***Ys***, ***ZAs***, ***ZRs***, and ***ZWs***.

Each register process ***REG***$[k]$, for $0 \leq k < 16$, consists of five elementary processes, one for each selection command. The register that is selected by command ***Xs*** is the one that passes the test $k = i.x$. This implementation requires that the variable $i.x$ be shared by all

registers and *EXEC*. An alternative solution that does not require shared variables uses demultiplexer processes. (The implementations of the two solutions are almost identical.)

The semicolons in the last two guarded commands of $REG[k]$ are introduced to pipeline the computation of the result of an ALU instruction or memory instruction with the decoding of the next instruction.

Mutual Exclusion on Registers

A register may be used in several arguments (x, y, or z) of the same instruction, and also as an argument in two successive instructions whose executions may overlap. We therefore have to address the issue of the concurrent uses of the same register. Two concurrent actions on the same register are allowed when they are both read actions.

Concurrency within an instruction is not a problem: X and Y communications on the same register may overlap, since they are both read actions, and Z cannot overlap with either X or Y because of the sequencing inside *ALU* and *MU*.

Concurrency in the access to a register during two consecutive overlapping instructions (one instruction is an ALU and the other is a memory instruction) can be a problem: Writing a result into a register (a ZA or a ZR action) in the first instruction can overlap with another action on the same register in the second instruction. But, because the selection of the z register for the first instruction takes place before the selection of the registers for the second instruction, we can use this ordering to impose the same ordering on the different accesses to the same register when a ZA or ZR is involved.

This ordering is implemented as follows: In $REG[k]$, variable bk (initially false) is set to true before the register is selected for ZA or ZR, and it is set back to false only after the register has been actually used. All uses of the register are guarded with the condition $\neg bk$. Hence, all subsequent selections of the register are postponed until the current ZA or ZR is completed.

We must ensure that bk is not set to true before the register is selected for an X or a Y action ***inside the same instruction***, since this would lead to deadlock. We omit this refinement which does not appear in the program of Figures 3 and 4.

8 Implementation

Control Part

The control part of a process is obtained by the following transformations: First, each communication command involving message input or output is replaced with a "bare" communication on the channel; for instance, $C?x$ and $C!x$ would both be replaced with C.

Second, all assignment statements are delegated to subprocesses. Assignment S is replaced with a communication command on a new channel, say Cs, and the subprocess $*[[\overline{Cs} \rightarrow S \bullet Cs]]$ is introduced. After these transformations, the control part of each process consists only of boolean expressions in conditionals and of communication commands. Thus, the next step is to implement each communication command with a ***handshaking protocol.***

Handshaking Protocols

Consider the matching pair of actions $X!u$ and $X?v$ in processes A and B respectively. We first implement the bare communication on channel X. The channel is implemented by the two handshake wires $(xo \ \underline{w} \ yi)$ and $(yo \ \underline{w} \ xi)$ as indicated on Figure 5.(a). As usual, we use a four-phase, or "return-to-zero" handshaking protocol. Such a protocol is not symmetrical: All communications in one process are implemented as ***active*** and all communications in the other process as ***passive***.

We have shown in [7] and [8] that the implementation of an input action is significantly simpler when combined with an active protocol than with a passive one. Therefore all input actions are implemented as active and all output actions as passive. (In the case of output, the implementation of communication is the same for active and passive protocols.)

The standard active and passive implementations are:

$$[yi]; yo\uparrow; [\neg yi]; yo\downarrow \quad \textbf{(passive)}$$

$$xo\uparrow; [xi]; xo\downarrow; [\neg xi] \quad \textbf{(active)} \ .$$

(The passive protocol starts with the wait action $[yi]$, i.e., "wait until the input wire is set to true." The active protocol starts with $xo\uparrow$, i.e., "set the output wire to true.")

We introduce an alternative active implementation, called ***lazy active***:

$$[\neg xi]; xo\uparrow; [xi]; xo\downarrow \quad \textbf{(lazy active)} \ .$$

The lazy active protocol differs from the active one in that the last wait action $[\neg xi]$ is postponed until the beginning of the next communication. The difference is important when data communication is involved.

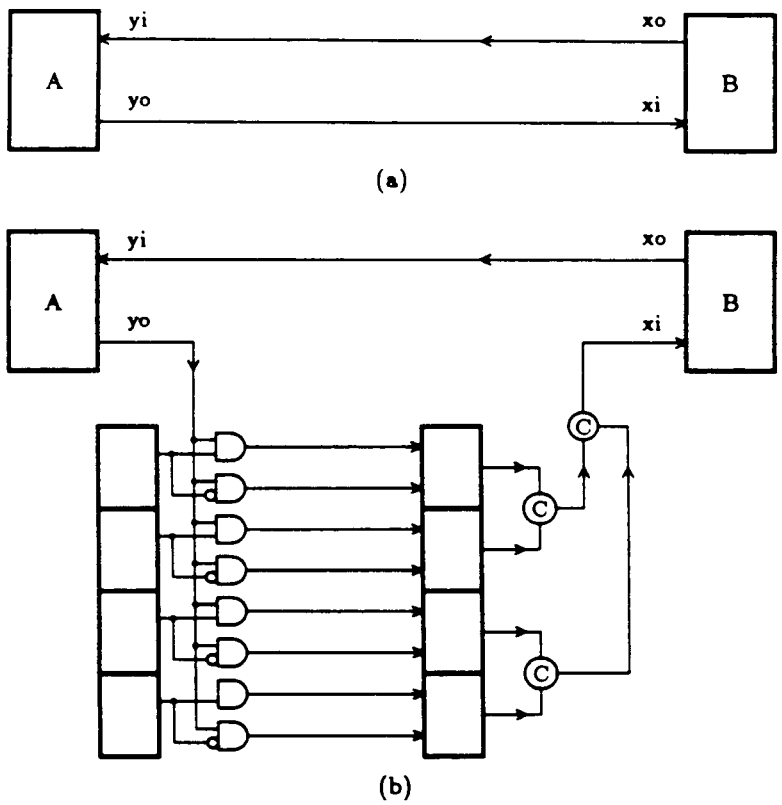


Figure 5: Implementation of communication

Figure 5.(b) shows how the data path is combined with the control. The bits of the communication channel between the two registers (the "data wires") are dual-rail encoded. Wire $(yo \underline{w} xi)$ is "cut open," yo is used to assigned the values of the bits of u to the dual-rail data wires, and xi is set to true when all bits of v have been set to the values of the data wires. Each cell of a register contains an acknowledge wire that is set to true when the bit of the cell has been set to a valid value of the two data wires, and reset to false when the data wires are both

reset to false. Let $vack_i$ be the acknowledge of bit v_i, xi is set and reset as:

$$vack_0 \wedge vack_1 \ldots \wedge vack_{15} \mapsto xi\uparrow$$
$$\neg vack_0 \wedge \neg vack_1 \ldots \wedge \neg vack_{15} \mapsto xi\downarrow$$

Since a 16-input C-element would be prohibitively slow to implement, the implementation is a tree of smaller C-elements, which we call a ***completion tree***. Figure 5.(b) shows a tree of binary C-elements. In the actual processor, we use a two-level tree of 4-input C-elements.

When data is transmitted via a bus, and when the completion tree is large, the gain of using a lazy-active protocol can be very important, since half of the data transmission delays and half of the completion-tree delays can overlap with the rest of the computation. Therefore, all input actions are implemented as lazy active.

The case when data is transmitted from process A to process B via a bus is only slightly more complicated. No arbitration is necessary: A and B are allowed to communicate via a bus only after the bus has been reserved for these two processes. The chief problem in implementing the buses is the distributed implementation of large multi-input OR-gates.

The lazy-active protocol cannot be used when an input action is probed—such as action $AC?op$ in the ALU—because the probe requires a passive protocol. For those cases, we have designed a special protocol that requires two control wires.

9 ALU

ALU control

In the ALU process, variable z is not needed to store the result of an ALU operation: the result can be put directly on the ZA bus. The first guarded command of the ALU process can be rewritten:

$$\overline{AC} \rightarrow AC?op \bullet X?x \bullet Y?y; \langle ZA, f\rangle := aluf(x, y, op, f).$$

Hence, the control part is simply:

$$\begin{aligned} *[&[\overline{AC} \rightarrow AC \bullet X \bullet Y; AL \\ &[]\overline{F} \rightarrow F \\ &]]. \end{aligned}$$

(The assignment to f is omitted.) Communication command AL is the call of the subprocess evaluating $aluf$. The handshaking protocol of AL is passive because it includes an output action on the ZA bus: $[ali]; alo\uparrow; [\neg ali]; alo\downarrow$. Hence, $alo\uparrow$ is the "go" signal for the ALU computation proper.

The first guarded command has the structure of a canonical stage of the pipeline. Parameters are simultaneously received on a set of ports, and the result is sent on another port as in:

$$*[L?x; R!f(x)].$$

Such a process is called a ***buffer***. Since L is implemented as lazy active, and R as passive, it is a ***lazy-active/passive buffer***. In the second design, where we have decomposed both the ALU and the memory processes into two processes in order to improve the pipeline, each stage of the pipeline is a lazy-active/passive buffer.

ALU data path

The output Z of the subprocess is dual-rail encoded. When the subprocess is called, variables x, y, and op have stable and valid values. Moreover, the content of op has been encoded in a KPG ("kill, propagate, generate") form which is used to produce the carry-out for each bit, and also for the result. The length of the carry chain is variable, which is an advantage in a fully asynchronous execution.

Since the carry-out of each bit is inverted relative to the carry-in, we alternate the logic encoding of the stages in the carry chain: A carry-in that has a true value when high generates a carry-out that has a true value when low, and vice-versa for the next stage. With this coding, only one CMOS gate delay is incurred per stage. Although the acknowledge from the ZA bus is used as completion signal, a completion tree is needed at the output of the subprocess for the computation of the flags.

The elapsed time between the activation of the ALU subprocess by $alo\uparrow$ and the appearance of the results on the output Z depends on the number of stages in the carry chain. Add, substract, and other logical functions typically take between 13 and 25ns in 2μm SCMOS.

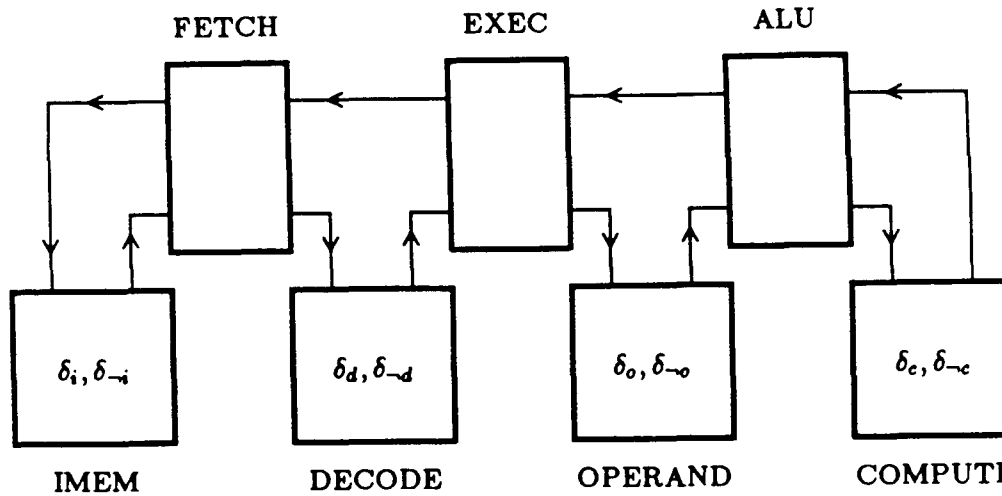


Figure 6: Abstract Pipeline for ALU Instructions

10 Performance

In this processor, an instruction is executed in a varying amount of time, depending in part on the type of instruction and the values of its operands, and on the sequence surrounding the instruction. Because of this data dependence, an analysis of the "real" performance of the processor, i.e., the performance of the processor when executing "real" programs, is quite complex and most probably must be determined by simulation. The performance analysis can be simplified by assuming an infinite sequence of identical instructions with typical operand values. (The results obtained through this analysis do not include the potential benefits of interleaving ALU and memory instructions.) Here, we analyze the performance of the processor executing an infinite sequence of ALU instructions.

In this case, the processor can be viewed as the three-stage pipeline shown in Figure 6. By assuming the ALU operations are performed on distinct registers, the register locking mechanism need not be introduced and the control for the ***EXEC*** process and the ***ALU*** process reduces to lazy-active/passive buffers. The ***fetch*** process is complicated by the increment of the **pc**, but if the instruction memory is assumed to be slower than the increment, control for this process also reduces to a lazy-active/passive buffer. By first assuming negligible control delays compared with datapath delays (denoted δ_D and $\delta_{\neg D}$ for the upgoing and downgoing propagation delays of datapath unit D, respectively),

the cycle time, c_P, of each process P is determined by the datapath delays that must be sequenced. A lazy-active/passive buffer sequences only the upgoing transitions of the two datapath units and, separately, the upgoing and downgoing transitions of the individual units, resulting in cycle time $\max(\delta_{D1} + \delta_{D2}, \delta_{D1} + \delta_{\neg D1}, \delta_{D2} + \delta_{\neg D2})$.

Since each process in the pipeline is a lazy-active/passive buffer, and since the throughput of the pipeline is determined by the slowest process:

$$\begin{aligned} c_{FETCH} &= \max(\delta_m + \delta_d, \delta_m + \delta_{\neg m}, \delta_d + \delta_{\neg d}) \\ c_{EXEC} &= \max(\delta_d + \delta_o, \delta_d + \delta_{\neg d}, \delta_o + \delta_{\neg o}) \\ c_{ALU} &= \max(\delta_o + \delta_c, \delta_o + \delta_{\neg o}, \delta_c + \delta_{\neg c}) \\ c_{PROC} &= \max(c_{FETCH}, c_{EXEC}, c_{ALU}) \ . \end{aligned}$$

Timing simulations suggest that the dominant constraints are the memory and decode sequence in the ***FETCH*** process ($\delta_m + \delta_d$), and the operand and compute sequence in the ALU process ($\delta_o + \delta_c$). For the $2\mu m$ SCMOS processor, the delays introduced by the control parts increase the cycle time by **10** to $\mathbf{20}ns$, bringing the cycle time for an infinite stream of ALU instructions up to $\max(\mathbf{35}ns + \delta_m, \mathbf{65}ns)$. We expect the processor to achieve **15** MIPS if the access delay of the instruction memory (δ_m) is no longer than $\mathbf{30}ns$.

11 Correctness by Construction and CAD Tools

Since the method is based on semantics-preserving program transformations, the object code generated by the compilation procedure is correct by construction.

The object code is a set of potentially concurrent ***production rules*** that are constructs of the form $B1 \mapsto x\uparrow$ or $B2 \mapsto x\downarrow$, where $B1$ and $B2$ are mutually exclusive boolean expressions, and $x\uparrow$ and $x\downarrow$ stand for "set x to true" and "set x to false," respectively. The compilation procedure guarantees the absence of hazards by ensuring that the conditions $B1$ and $B2$ are ***stable***, i.e., if $B1$ is true, it remains true until x as been set to true.

If the production rules of the object code can be matched with the production rules that describe the standard cells of a cell library, a standard-cell-layout program can be used to generate a layout corresponding to the object code. We have been using such a standard cell approach in our previous designs, and indeed all chips fabricated in this way have been found to be functional on "first silicon."

However, most of the processor was designed manually. First, since the control section introduces significant overhead, we decided to

compile its object code manually. Second, because the data path was expected to be the critical part with respect to size and because of the difficulty of adjusting the pitch of the different registers automatically, the automatic layout program was used for the control part but not for the data path. This decision was later justified by the fact that, whereas the data path was hardly changed after the first design, the control part went through a series of drastic modifications. We observed that, again, our method for separating control and data path permitted us to implement completely different pipelines by changing the control without significant alterations of the data path.

As usual, the disadvantage of manual compilation was that the design was not shielded from clerical errors at which humans excel.

While the difficult optimization problem that is at the core of a high-performance processor design is probably still beyond automatic compilation technology, the designer should be assisted with CAD tools that perform the mechanical translation steps. Other CAD tools that we found useful include a program that estimates the critical path of a circuit. The program, which was developed by Steve Burns, gives excellent results. It estimates the delays of each path by a simulation of the execution based on the production rules.

Magic was used for the manual layout [10].

12 Conclusion

Although the chips are still in fabrication, we are very satisfied with the preliminary results of the experiment.

First, the chip layout is obviously not large. The control is surprisingly small despite our use of an automatic layout tool; also, the anticipated nightmare of data path layout did not materialize. The register pitch is 80λ, which is quite reasonable given that four buses have to be placed.

Second, the predicted performance is quite remarkable, given that the experiment is a first in two ways: It is our first experience as computer architects, and it is the first asynchronous microprocessor ever built.

Third, the complete design took five persons (one joined in the middle of the project) five months.

Since the choice of an instruction set was not part of the experiment, our design should be judged in two ways: the choice of the concurrent program of Figure 3, and its implementation.

The implementation is satisfactory, but not optimal. The sizing of transistors can be improved and the number of transitions can be decreased, mainly by a better placement of inverters. For instance, the delays due to a completion tree and to the control for a buffer are both about twice their theoretical minimum.

The program of Figure 3 represents the choice of a pipeline, and of synchronization techniques to implement it. We have deliberately chosen a simple pipeline. In particular, the mechanism for stalling, which places part of the decoding in series with the fetch on the critical path, sacrifices efficiency for simplicity. However, performance evaluations show that the pipeline is well-balanced since the different stages have comparable average delays. Improving the critical path by overlapping fetch and decode requires improving the ALU and memory instruction execution stages by pipelining parts of these stages.

The practicality of overlapping ALU and memory instruction executions remains an open issue. It is not clear whether the gain in performance is worth the complexity of the synchronization involved and the requirement of two separate Z buses.

We find the synchronization techniques used to implement the concurrent activities between the different stages of the pipeline particularly elegant and efficient, since the delays incurred in a synchronization can be of arbitrary length and vary from instruction to instruction.

We foresee excellent performances for asynchronous processors as the feature size keeps decreasing. But the designer must be ready to learn and apply new design methods based on concurrent programmming, that are required to exploit asynchronous techniques to their fullest.

Acknowledgment

We are indebted to Bill Athas and Bill Dally for useful discussions in the preliminary stage of the design. Chuck Seitz, Nanette Boden, and Dian De Sha made excellent comments on the manuscript. The first author enjoyed numerous discussions with Chuck Seitz on the general topic of asynchronous design.

The research described in this paper was sponsored by the Defense Advanced Research Projects Agency, DARPA Order numbers 3771 & 6202, and monitored by the Office of Naval Research under contract numbers N00014-79-C-0597 & N00014-87-K-0745.

References

[1] R.E. Bryant, D. Beatty, K. Brace, K. Cho, and T. Sheffler. COSMOS: A Compiled Simulator for MOS Circuits. In ***24th Design Automation Conference***, pp.9–16. ACM and IEEE, 1987.

[2] Steven M. Burns and Alain J. Martin, Syntax-directed Translation of Concurrent Programs into Self-timed Circuits. In J. Allen and F. Leighton (ed), ***Fifth MIT Conference on Advanced Research in VLSI,*** pp 35-40, MIT Press, 1988.

[3] C.A.R. Hoare, Communicating Sequential Processes. ***Comm. ACM*** 21,8, pp 666-677, August, 1978.

[4] Mark Horowitz ***et al.***, MIPS-X: A 20-MIPS Peak, 32-bit Microprocessor with On-Chip Cache. ***IEEE Journal of Solid-State Circuits***,SC-22(5):790-799,October, 1987.

[5] Alain J. Martin, The Design of a Self-timed Circuit for Distributed Mutual Exclusion. In Henry Fuchs (ed), ***1985 Chapel Hill Conf. VLSI***, Computer Science Press, pp 247-260, 1985.

[6] Alain J. Martin, Compiling Communicating Processes into Delay-insensitive VLSI Circuits. ***Distributed Computing***, 1,(4), Springer-Verlag, pp 226-234 1986.

[7] Alain J. Martin, A Synthesis Method for Self-timed VLSI Circuits. ***ICCD 87: 1987 IEEE International Conference on Computer Design***, IEEE Computer Society Press, pp 224-229, 1987.

[8] Alain J. Martin, Programming in VLSI: From Communicating Processes to Delay-insensitive Circuits. In C.A.R. Hoare (ed), ***UT Year of Programming Institute on Concurrent Programming***, Addison-Wesley, Reading MA, 1989.

[9] Carver Mead and Lynn Conway, ***Introduction to VLSI Systems***, Addison-Wesley, Reading MA, 1980.

[10] J. K. Ousterhout ***et al.***, The Magic VLSI layout system, ***IEEE Design Test Comput.***, 2, (1), pp 19-30, February, 1985.

[11] Charles L. Seitz, System Timing, Chapter 7 in Mead & Conway, ***Introduction to VLSI Systems***, Addison-Wesley, Reading MA, 1980.

Appendix 1: Notation

The program notation, which is inspired by C.A.R. Hoare's CSP [3], is briefly described.

$b\uparrow$ stands for $b :=$ **true**, $b\downarrow$ stands for $b :=$ **false**.

The execution of the ***selection*** command $[G_1 \rightarrow S_1 [] \ldots [] G_n \rightarrow S_n]$, where G_1 through G_n are boolean expressions, and S_1 through S_n are program parts, (G_i is called a "guard," and $G_i \rightarrow S_i$ a "guarded command") amounts to the execution of an arbitrary S_i for which G_i holds. If $\neg(G_1 \vee \ldots \vee G_n)$ holds, the execution of the command is suspended until $(G_1 \vee \ldots \vee G_n)$ holds.

The execution of the ***repetition*** command $*[G_1 \rightarrow S_1 [] \ldots [] G_n \rightarrow S_n]$, where G_1 through G_n are boolean expressions, and S_1 through S_n are program parts, amounts to repeatedly selecting an arbitrary S_i for which G_i holds and executing S_i. If $\neg(G_1 \vee \ldots \vee G_n)$ holds, the repetition terminates.

For communication actions X and Y, "$X \bullet Y$" stands for the coincident execution of X and Y, i.e., the completions of the two actions coincide.

$[G]$ where G is a boolean expression, stands for $[G \rightarrow \textbf{skip}]$, and thus for "wait until G holds."

(Hence, "$[G]$; S" and $[G \rightarrow S]$ are equivalent.)

$*[S]$ stands for $*[\textbf{true} \rightarrow S]$, and thus for "repeat S forever."

From (iii) and (iv), the operational description of the statement $*[[G_1 \rightarrow S_1 [] \ldots [] G_n \rightarrow S_n]]$ is "repeat forever: wait until some G_i holds; execute an S_i for which G_i holds."

Communication commands: Let two processes, ***p1*** and ***p2***, share a channel with port X in ***p1*** and port Y in ***p2***. (In the processes of Figure 3, the same name is used for all the ports of the same channel.) If the channel is used only for synchronization between the processes, the name of the port is sufficient to identify a commnication on this port. If the communication is used for input and output of messages, the CSP notation is used: $X!u$ outputs message u, and $X?v$ inputs message v.

At any time, the number of completed X-actions in ***p1*** equals the number of completed Y-actions in ***p2***. In other words, the completion of the nth X-action "coincides" with the completion of the n-th Y-action. If, for example, ***p1*** reaches the nth X-action before ***p2*** reaches the nth Y-action, the completion of X is suspended until ***p2*** reaches Y. The X-action is then said to be ***pending***. When, thereafter, ***p2*** reaches Y, both X and Y are completed. It is possible (and even advantageous) to define communication actions as coincident and yet implement the actions in completely asynchronous ways. For an explanation, see [8].

Probe: Since we need a mechanism to select a set of pending communication actions for execution, we provide a general boolean command on ports, called the ***probe***. In process $p1$, the probe command $\overline{X}$ has the same value as the predicate "Y is pending in $p2$."

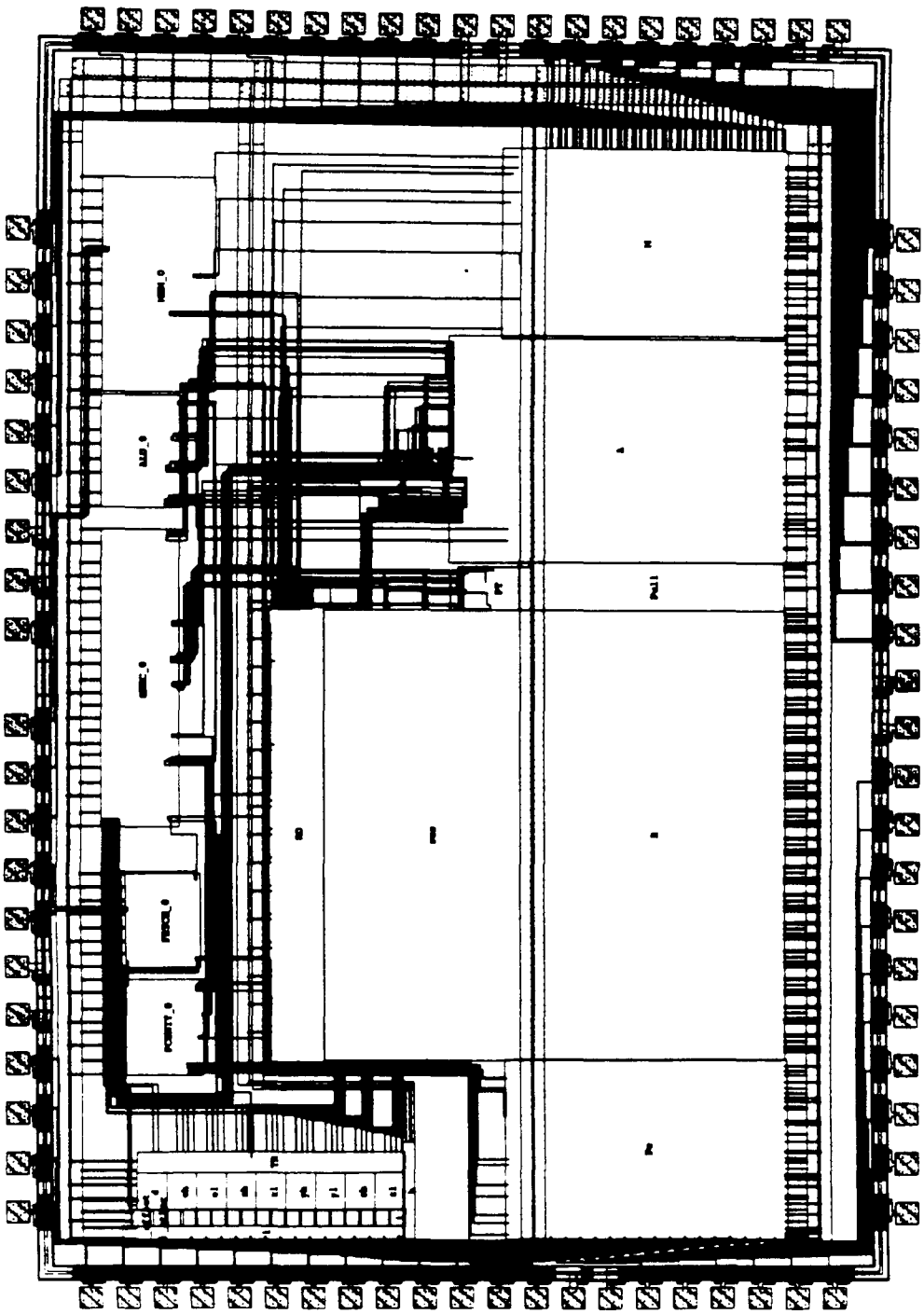
Appendix 2: Instruction Set

ALU	op rx ry rz	rz,f := rx op ry
MEM	op rx ry rz	rz := mem[rx+ry] (for load)
		mem[rx+ry] := rz (for store)
MEMOFF	op ao ry rz	rz := mem[ry + offset] (for load)
	offset	mem[ry + offset] := rz (for store)
		rz := ry + offset (for load address)
BRANCH	op ao — cc	if cond(f,cc) then pc := pc + offset
	offset	
JUMP	op ao ry —	pc := ry
STPC	op ao — rz	rz := pc

Table 1: Instruction Types

inst	n_3 $b_{15}b_{14}b_{13}b_{12}$	n_2 $b_{11}b_{10}b_9b_8$	n_1 $b_7b_6b_5b_4$	n_0 $b_3b_2b_1b_0$
alu	0011	rx	ry	rz
	0100	rx	ry	rz
	⋮			
	1111	rx	ry	rz
ld	0010	rx	ry	rz
st	0001	rx	ry	rz
ldx	0000	0000	ry	rz
stx	0000	0001	ry	rz
lda	0000	0010	ry	rz
brc	0000	0011	—	cc
jmp	0000	0100	ry	—
stpc	0000	0101	—	rz

Table 2: Opcode Assignments



Variants of the Chandy-Misra-Bryant Distributed Discrete-Event Simulation Algorithm

Wen-King Su and Charles L. Seitz
Department of Computer Science
California Institute of Technology

Caltech-CS-TR-88-22

1. Introduction

We have been using variants of the Chandy-Misra-Bryant (CMB) distributed discrete-event simulation algorithm [1,2,3] since 1986 for a variety of simulation tasks [4]. The simulation programs run on multicomputers [5] (message-passing concurrent computers), such as the Cosmic Cube, Intel iPSC, and Ametek Series 2010. The excellent performance of these simulators led us to investigate a family of variants of the basic CMB algorithm, including lazy message-sending, demand-driven operation with backward demand messages, and adaptive adjustment of the parameters that control the laziness.

These studies were also motivated by our interest in scheduling strategies for reactive (message-driven) multiprocess programs [5,6,7], which are semantically similar to discrete-event (event-driven) simulators. The simulator itself is implemented in the reactive programming environment that we have developed for multicomputers: the Cosmic Environment and the Reactive Kernel [8].

We performed the studies reported here using logic networks. Logic simulation is expected to stress a distributed simulator, and is itself of practical interest. It is easy to construct examples of logic networks with a diversity of behaviors and structural difficulties, such as large fan-in and fan-out. Low-level logic elements such as logic gates exhibit responses in which an input event may or may not influence the outputs, depending on the internal state of the element and on the states of other inputs; yet, they require very little computation to simulate their behavior. Thus, the performance results shown later in this paper involve practically no computation other than the distributed simulation itself.

This paper is a brief and preliminary report of the simulation algorithms and performance results. A more definitive report will be found in the first author's forthcoming PhD thesis.

The research described in this paper was sponsored in part by the Defense Advanced Research Projects Agency, DARPA Order number 6202, and monitored by the Office of Naval Research under contract number N00014-87-K-0745; and in part by grants from Intel Scientific Computers and Ametek Computer Research Division.

2. The CMB Simulation Framework

As usual, the system to be simulated is modeled as a set of communicating elements. A CMB simulator can be implemented by coding the behavior of elements in processes that communicate by messages. A message conveys both a time interval and any events within this interval. A process reacts to the receipt of an input message by updating its internal state, and, if outputs can be advanced in time, by sending messages to connected processes. These messages may include *null messages* that convey no events (changes in the state information), but serve only to advance the simulation time.

It is easy to show that such a simulator is correct [3], in the sense that it computes a possible behavior of the system being simulated. A sufficient condition for freedom from deadlock in this eager message-sending mode is that there is a positive delay in every circuit in the graph of element vertices and communication arcs. Intuitively, it is the delay of the elements being simulated that permits the element simulators to compute the outputs over an interval that is later than the time of the inputs, so that time advances. Simulation time is determined locally, and may get as far out of step at different elements as their causal relationships permit.

This conservative (also known as pessimistic) type of simulator is a concurrent program that exploits the concurrency inherent in the system being simulated. In practice, just as with other concurrent programs, if the number of concurrently runnable processes substantially exceeds the number of processors, one can achieve high utilization of concurrent resources. The speculative (also known as optimistic) type of simulator attempts to exploit additional concurrency by computing beyond the interval during which inputs are defined, at the risk of having to roll back if the speculations prove incorrect. Such approaches are attractive for simulating systems whose inherent concurrency is insufficient to keep concurrent resources busy, and in which speculations can be made with high confidence. Our studies have concentrated on conservative variants of the CMB algorithm.

The design of distributed simulation programs is also influenced by a characteristic of the element simulators. In practice, an element simulator may or may not take as long to process a null message as an event-containing message. For the simulation of some systems, the processing of an event-containing message might involve a lengthy simulation of a physical process, whereas the processing of a null message might be very fast. Such simulations do not seriously stress the distributed-simulation aspect of the computation. However, for the simulation of systems of extremely simple elements, such as logic gates, the time required to compute the output of the gate is so small that it is comparable to the time required to process a null message.

Due to our interest in understanding the limits of event-driven distributed simulation, and the implications for scheduling strategies for message-driven multiprocess programs, our studies have concentrated on the case in which the time required to process null messages is comparable to the time required to process event-containing messages. It is straightforward to extrapolate the performance results for this difficult case to situations in which null-message processing is relatively fast.

The principal trouble with naive implementations of conservative CMB distributed simulation programs in any situation in which processing null messages is as costly as processing event-containing messages is that the volume of null messages may greatly exceed the number of event-containing messages. This difficultly is most evident when simulating systems with many short-delay circuits that have relatively low levels of activity.

In distributing the simulation, we seek to reduce the time required to complete the computation; however, we have an immediate problem if the element simulators must perform many more message-processing operations in the distributed simulation than they would perform event-processing operations in a sequential simulation. The centralized regulation of the advance of time achieved through the ordered event list maintained by sequential simulation programs allows these simulators to invoke element routines only once for each input event. The null messages inflate not only the volume of messages the system must handle, but also the computational load. Thus, if we are going to compete with the best sequential simulators, we must reduce the volume of null messages.

3. Indefinite Lazy Message Sending

To reduce the volume of messages, we use various strategies to defer sending outputs in the hope that the information can be packed into fewer messages. For example, one of the most obvious schemes is to defer sending null messages, so that a series of null messages and an event-containing message can be combined to form a single message that spans a longer interval. Since output events are often triggered only by input events, deferring the delivery of preceeding null messages is less likely to hamper the progress of the destination element than deferring the delivery of event-containing messages.

The first problem that must be addressed in employing such strategies is deadlock. When element simulators defer sending output messages, they may cyclically deny themselves input messages, leading to deadlock. All of our simulators have employed a technique of *indefinite lazy message sending* to permit arbitrary strategies for deferring message sending while still avoiding deadlock. The following is an idealized inner loop of the simulator, shown in the C programming language:

```
while(1)
  if (p = xrecv())
     simulate_and_optionally_send_messages(p);
  else
     take_other_action();
```

The function `xrecv` returns a pointer, `p`, that points to a message for the simulation process if a message has been received. The simulator then dispatches to the appropriate element simulator, and may either send or queue the outputs that the element simulator produces. If there is no message in the node's receive queue, the pointer returned is a NULL (0) pointer. In this case, the simulator takes other

action to break any possible deadlock. For a source-driven simulator, it selects a queued output to send as a message. For a demand-driven simulator, it selects a blocked element, and sends a *demand* message to its predecessor to request that queued outputs be sent. A deadlock in deferring messages cannot occur without "starving" a node of messages. When this situation is detected by `xrecv` returning a NULL pointer, the resulting action breaks the potential deadlock.

Within this indefinite lazy message-sending framework, we can experiment with *any* scheme for deferring and combining messages without concern for deadlock. A message is free to carry any number of events, and an element is free to defer message sending on any basis.

4. Variant Algorithms

We have experimented with many CMB variants; in the interests of comprehension, we will describe the operation and report the performance of six that are representative of the range of possibilities that we have studied:

A *Eager message sending:* This basic form of CMB serves as a baseline for comparison against the variants.

B *Eager events, lazy null messages:* Null outputs are queued. Event outputs, combined with any queued null outputs, are sent immediately. When `xrecv` returns a NULL pointer, the null output that extends to the earliest time is sent as a null message.

C *Indefinite-lazy, single-event:* All output from element simulators is queued. The output queues may contain multiple events. Messages are sent only when `xrecv` returns a NULL pointer. The output queue that extends to the earliest time is selected to generate a message up to the first event, if any, or a null message to the end of the interval.

D *Indefinite-lazy, multiple-event:* This scheme is a slight variation on *C*, motivated by characteristics of multicomputer message systems that make it economical to pack multiple events into fewer messages. All output from element simulators is queued. The output queues may contain multiple events. When `xrecv` returns a NULL pointer, the output queue that extends to the earliest time is selected to generate a message up to the *last* queued event, if any, or a null message to the end of the interval. However, to allow a direct comparison with sequential simulators, events are processed singly.

E *Demand-driven:* Although we usually think of simulation as source driven from inputs, one can equally well organize the simulation as demand driven from outputs. In the pure demand-driven form, all output from element simulators is queued. When `xsend` returns a NULL pointer, the input that lags furthest behind selects the destination for a demand message. Upon receipt of a demand message, if the output queue is not empty, the simulator sends all the information in the output queue; if the output queue is empty, the simulator generates another demand message to the source of lagging input to this element.

F Demand-driven, adaptive: Demand messages single out critical paths in a simulation. In an adaptive form of demand-driven simulation, a threshold is associated with each communication path. Outputs of element simulators are queued only up to the threshold; when the threshold is exceeded, the contents of the queue are sent as a message. Demand messages operate as in E, but also cause the threshold to be decreased. In the cases shown below, the threshold is halved. The simulator is accordingly able to adapt itself to the characteristics of the system being simulated.

Although these variants are described here in terms of message passing, the same variants also appear as different scheduling strategies in shared-memory implementations.

5. Experimental Method

In common with other highly evolved message-passing programs, the simulator is implemented with one simulation process per multicomputer node (or, in the Cosmic Environment, with one simulation process per host computer or per processor in a multiprocessor).

Basis of comparison: Although execution time is one of the most natural bases of comparison between any two programs that perform the same function, and is used below to illustrate the performance of our distributed simulators on different commercial multicomputers, execution time on these concurrent computers depends both on the algorithm and on the characteristics of the particular computer. When we wish to isolate the characteristics of the algorithm from those of the computer, the instrumented simulator operates as a simulator within a simulator. Execution time is then measured in a unit called a *sweep* [5, 6], which corresponds here to a fixed time required to call an element once. The time required for other operations, such as sending a message, can be set to a particular number of sweeps. Normally, a message sent by one node in one sweep is available in the destination node at the next sweep. However, to test the sensitivity of the algorithms to message latency, we can also set the latency to larger values.

Instrumentation: The simulator is a reactive program written in C, and is instrumented to function in two operational modes. In the *sweep mode*, a multicomputer-emulation program runs a simulation of a multicomputer; this in turn runs the reactive simulators. Time is measured in sweep units; on each sweep, each node is allowed to make one element call. In the *real mode*, the simulator runs directly on the multicomputer. There is one copy of the simulator process in each node, and each simulator process runs a subset of the elements as embedded reactive processes. Each node runs at its own pace, and execution time is measured with UNIX's real-time clock.

6. Experimental Results

Performance measurements have been made on a variety of logic networks, including those that are representative of networks found in computers and VLSI chips, and

those that are designed specifically to test or to stress the simulator. Six different network types, each in several sizes up to 4000 logic gates, have been the principal vehicles for these experiments. A larger variation in performance is observed among networks with different characteristics than between algorithm variants.

Multiplier example: The parallel multiplier is a good example of an ordinary logic network. The 14×14 multiplier used in several experiments employs 1376 logic gates to generate the 28-bit product of two 14-bit binary inputs. The multiplier network contains only limited concurrency, and does not contain tight circuits that give the simulator artificial performance boosts or troubles, depending on element distribution. It also contains moderately high fan-out in the multiplier and multiplicand lines; this puts pressure on the message system. In all fairness, the distributed simulation of this multiplier network is not expected to do too badly nor too well.

For the simulation, the most-significant bit of the product is connected back to the multiplier input via an inverting delay. The delay is such that the multiplier reaches a stable state before the multiplier input changes. The multiplicand input is set to a value that causes the circuit to oscillate. A trace of the product outputs shows that the simulator and the circuit are running correctly.

Measurements in the sweep mode: The plot in Figure 1 portrays in a log-log format the sweep count in the sweep mode versus the number of nodes, N, for the simulation of the 14×14 multiplier network under all six CMB variants. It is not useful to continue the plot beyond 2^{11} nodes, since at this point there are as many nodes as simulated gates. The placement of elements in nodes for these trials is balanced but random.

Each horizontal division represents a factor of two in resources; each vertical division represents a factor of two in sweep count or time. We have found this format (*cf* [5]) for portraying the performance of concurrent programs to be more useful than "speedup" graphs, for two reasons. First, we can observe the factor by which the execution time is reduced as resources are increased over very wide ranges. Second, since the ordinate is a physical measure, time or sweep count, we can compare different algorithms directly. For example, in addition to the plots of the sweep counts of the CMB variants, the heavy horizontal line represents the number of sweeps a sequential simulator requires for this same simulation.

The first remarkable characteristic of these performance measurements is that they are so similar across this class of variant algorithms. Algorithms A, E, and F produce more messages than B, C, and D, but in this mode in which messages are free but element invocations are expensive, there is little difference between the variants. The performance under sweep-mode execution exposes the intrinsic characteristics of the algorithm, and is not related to such multicomputer characteristics as the relationship between node computing time and message latency.

The gross characteristics of these curves are similar to those of other concurrent programs [5], and are quite understandable and predictable.

We observe at $\log_2 N{=}0$ (1 node) that all of the CMB variants are somewhat inefficient in comparison with the sequential event-driven simulator. For this

multiplier example, the null messages inflate the number of element invocations by a factor of 2–5 times; this is consistent with the 1–2.5-octave increase in sweep count over that of the sequential simulator. The null messages also inflate the concurrency over that which is intrinsic to the system being simulated. We shall refer to this inflation in the number of element invocations as the *overhead* of distributing the simulation. If the time required to process a null message were smaller than the time required to process an event-containing message, the overhead would be reduced proportionately.

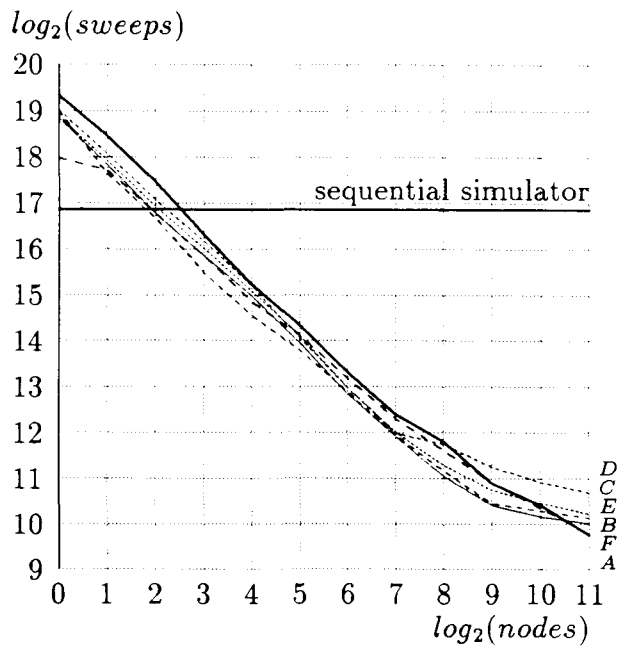


Fig 1: A 1376-gate multiplier, sweep mode

The performance is then divided roughly into two regimes, the first regime being one of near-linear speedup in N for the first 7–8 octaves, and the second regime being one of diminishing returns in N as the computing time approaches an asymptotic mimimum value. In the linear speedup regime, these simulators nearly halve the sweep count with each doubling of resources until limiting effects are reached. Load balance is assured by the weak law of large numbers when there are many elements per node. While each node has a sufficiently large pool of work, node utilization remains high. The simulators approach asymptotic minimal time as they exhaust the available concurrency in the system being simulated. The gradual "knee" of the curve originates from progressively less-effective statistical load balancing as the number of elements per node diminishes with larger N.

Additional statistics have been collected to measure other effects. For example, in the linear-speedup regime, when there are many logic elements per node, the simulators are quite insensitive to message latency. When there are few elements per node, the performance begins to deterioriate as message latency is increased. These

effects will be evident in the measurements performed on real multicomputers.

Measurements on real multicomputers: The results of simulating the same 1376-gate multiplier network on a 16-node iPSC/2 is shown in Figure 2, and on a 128-node iPSC/1 for variants B, C, and D is shown in Figure 3. The iPSC/2 is $\approx$6 times faster per node than the iPSC/1, so the time scales do not correspond. This simulation will not run on an iPSC/1 for $N < 4$ because the data and message queues for an increased number of logic elements per node will not fit in the node memory. Due to the same limitations of the iPSC/1 message system, neither the demand-driven nor the eager-message-sending simulation variants will run in most machine sizes. This choice of performance data is dictated by the desire to show performance results over the largest range of N possible with the machines that are currently operated by our research group. Results essentially identical to those shown in Figure 2 are also obtained on a 16-node Ametek Series 2010.

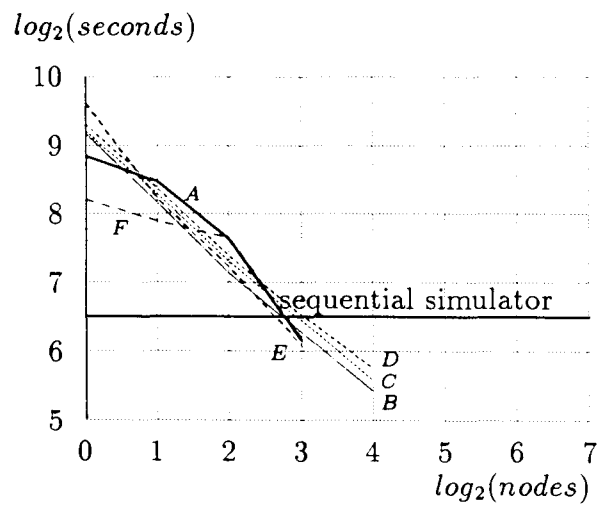


Fig 2: A 1376-gate multiplier for 40μs on an iPSC/2

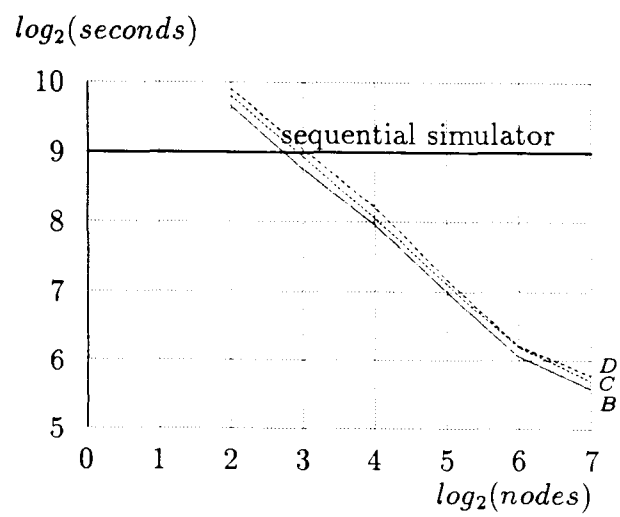


Fig 3: A 1376-gate multiplier for 40μs on an iPSC/1

The simulation of this network for $2^0 \leq N \leq 2^7$ is in the relatively uninteresting (but useful) linear-speedup regime, with some limiting effects starting to be seen in Figure 3 at $N=2^7$. The number of gates being simulated per node is sufficiently high to keep the node utilization high and the sensitivity to message latency low.

In order to exhibit the performance results in the more interesting (but less useful) diminishing-returns regime, we have scaled the network down to a 4-bit multiplier with 116 logic gates. The peformance on an Intel iPSC/2 up to 16 nodes is shown in Figure 4, and on an Intel iPSC/1 up to 128 nodes is shown in Figure 5. This network is small enough to exhibit interesting limiting effects as the simulation is increasingly distributed. The sublinear speedup is due to message latency in inter-node communications, increased null messages as the simulation is increasingly distributed, and load imbalance. The asymptotic time is limited by the message latency rather than by the available concurrency. In particular, Figure 5 shows that the asymptotic execution time of algorithm A, which is not very economical in its use of messages, is more than a factor of two worse than the asymptotic execution time of variants B, C, and D.

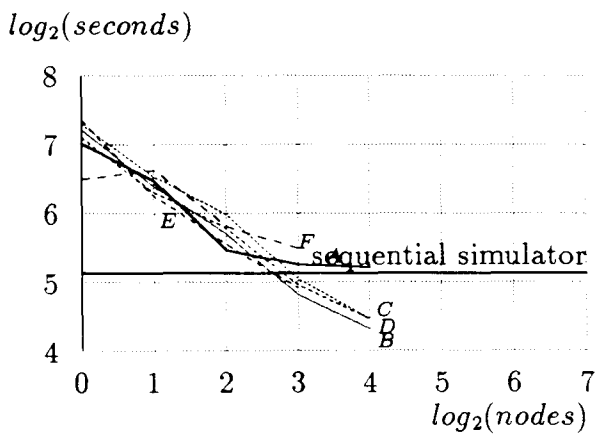


Fig 4: A 116-gate multiplier for 100μs on an iPSC/2

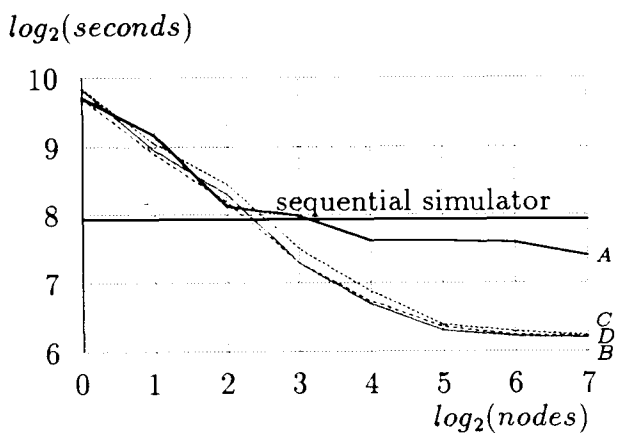


Fig 5: A 116-gate multiplier for 100μs on an iPSC/1

7. Hybrid CMB Variants

Although the CMB variants exhibit good speedup over wide ranges of N, speedup measures only the performance of the algorithm relative to less-distributed instances of itself. In comparison with the sequential simulator, the distributed simulators must pay the overhead of processing null messages. If the elements used in a simulation are such that the time required to process null messages is considerably less than the time to process event-containing messages, these conservative CMB variants will provide excellent performance and efficiency.

However, if the time required to process null messages is comparable to the time required to process event-containing messages, as it is for logic simulation, this overhead makes the CMB algorithm and its variants problematic for simulations on parallel computers in which N is small. What might be done to extend the CMB approach into this difficult small-N range?

A component of the overhead that cannot be eliminated within the CMB framework, in which elements are independent processes, is the null messages used to force progress in cycles of idling elements. However, we can take advantage of multiple elements sharing the same node by lumping members of low-latency, low-activity cycles, such as the gates that form a latch, into macro elements, and applying sequential simulation to them internally. The null-message-processing overhead for such cycles is eliminated at the cost of reduced concurrency for their members.

In this type of hybrid CMB variant simulator, all elements in each node are combined into one macro element, which is simulated internally with a conventional, ordered-event-list, sequential simulator. These sequential simulators are tied together externally with one of the CMB variant simulators. Since there is only one macro element per node, the hybrid variants are identical at N=1 to a sequential simulator. As N increases, however, more cycles are partitioned over multiple nodes, and each hybrid variant eventually converges with its corresponding CMB variant.

Measurements in sweep mode: Figure 6 shows the performance results for the CMB variants simulating a ring of 28 self-timed FIFO units. Each FIFO unit contains one FIFO-control cell and eight register cells, implemented with a total of 1067 logic gates. The FIFO ring is 50% full, holding 14 alternating 1- and 0-bytes. The overhead at N=1 is caused by the idling of the cross-coupled NAND latches in the registers and the FIFO controls. The CMB variants show a good speedup with increased N. Except for the initial overhead, the performance of all of the CMB variants is excellent.

Figure 7 shows the simulation results for the same circuit using the hybrid CMB variants with an element-distribution method that tends to place elements of each cycle in the same node.

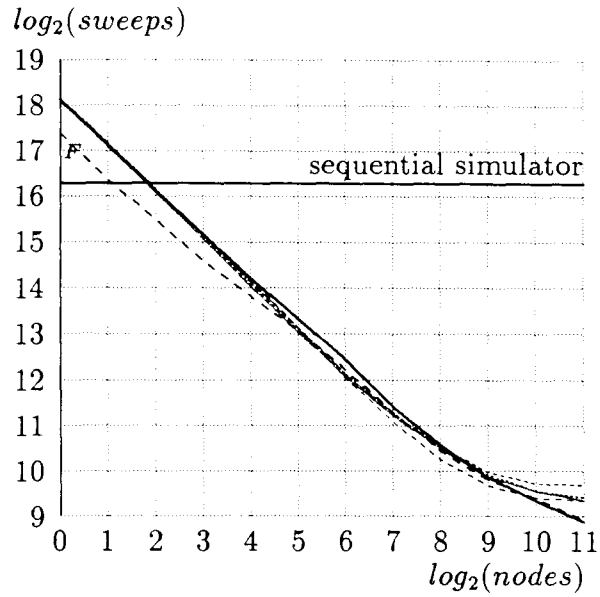


Fig 6: FIFO ring, non-hybrid simulator, emulation mode

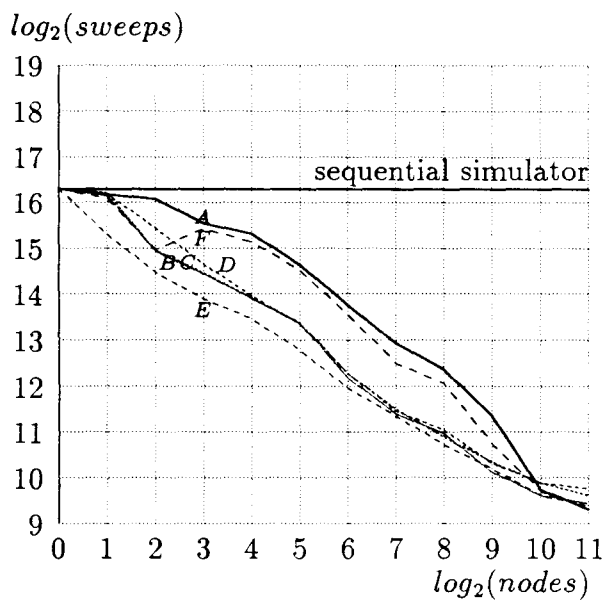


Fig 7: FIFO ring, hybrid simulator, emulation mode

Although the hybrid simulator exhibits a generally decreasing sweep count with increasing N, and extremely good small-N performance for the demand-driven variant E, less desirable behaviors have been observed for the hybrid variants. In particular, if the elements are not properly distributed, or cannot be properly distributed, the simulation time may increase starting at N=2 before starting to decrease. This effect is the result of cycles being broken and scattered over multiple nodes, so that it is the CMB rather than the sequential algorithm that dominates the execution time. Figure

8 illustrates the performance of the simulator for the same circuit used in Figures 6 and 7, but with random placement of the elements across the nodes.

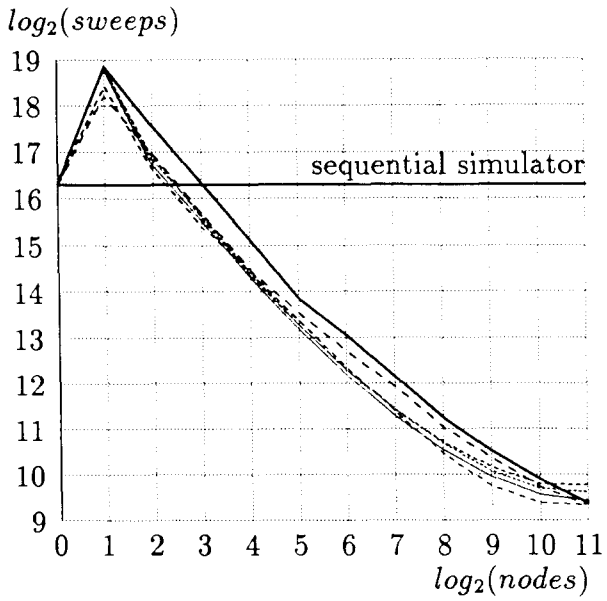


Fig 8: FIFO ring, hybrid simulator, randomized

Some programming short-cuts were used to produce these sweep-mode performance measures for the hybrid variants without implementing a regular sequential simulator; thus, we are not able to include corresponding performance graphs for real multicomputers. However, the instrumentation of the hybrid sweep-mode simulations, together with the performance parameters of second-generation multicomputers such as the Intel iPSC/2 and Ametek Series 2010, indicate that the performance on real multicomputers will be essentially similar to that in the sweep-mode. We are currently implementing distributed simulation programs and instrumentation to run the hybrid CMB variants on real multicomputers.

8. Conclusions

We selected logic simulation for these experiments because we wished to examine the limits of the applicability of the conservative CMB algorithm and its variants. Simulating the behavior of relatively simple elements that have a high degree of connectivity was expected to be a difficult case for distributed simulation. Indeed, the performance results presented here have been much more revealing of the capabilities and limitations of the distributed discrete-event simulation algorithms than earlier simulations that we performed of systems such as multicomputer message networks.

The reader should accordingly be cautious about drawing negative conclusions about the CMB framework from our comparisons of the performance of the CMB variants with the ordered-event-list sequential simulator. For objects of distributed simulation that are less demanding than logic simulation, such as systems in which

processing null messages is much faster than processing event-containing messages, the overhead is proportionately scaled down, and the following general conclusions remain valid:

1. Selected CMB varients exhibit excellent speedup over a wide range of N, limited eventually only by the concurrency of the system being simulated.
2. The CMB variants presented here, all based on the indefinite-lazy-message-sending framework, provide a useful improvement over the basic eager-message-sending CMB algorithm.
3. The hybrid CMB variants offer promise of efficient distributed simulation on small-N concurrent computers.

In some respects, the CMB and sequential algorithms make poor comparison subjects because these two algorithms represent relatively orthogonal optimizations in the basic task of simulation. While the execution time of the sequential simulator is sensitive only to the activity level of the circuit, the execution time for the fully distributed CMB algorithm is sensitive only to the structure of the circuit. In the FIFO-ring example, we can use more data bytes, fewer data bytes, or a different set of data bytes, and shift the sequential simulator's execution time proportionately without significantly changing the CMB variants' curves. Similarly, we can shift the CMB variants' curves without affecting the execution time of the sequential algorithm by varying the delay of the gates in the latches.

The hybrid CMB variants attempt to combine the best aspects of the sequential and CMB algorithms by allowing the sequential simulator to dominate when N is small, and the CMB variants to dominate when N is large. This approach may or may not produce a favorable result, depending on whether the elements can be properly distributed. More research needs to be done in the area of element distribution and its effect on the hybrid variants.

9. Acknowledgment

We very much appreciate the constructive suggestions, ideas, and encouragement that we have received from K. Mani Chandy.

10. References

[1] K. Mani Chandy and Jayadev Misra, "Asynchronous Distributed Simulation Via a Sequence of Parallel Computations," *CACM* 24(4), pp 198–205, April 1981.

[2] Randal E. Bryant, "Simulation of Packet Communication Architecture Computer Systems," MIT-LCS-TR-188, Massachusetts Institute of Technology, 1977.

[3] Jayadev Misra, "Distributed Discrete-Event Simulation," *Computing Surveys* 18(1), pp 39–65, March 1986.

[4] "Submicron Systems Architecture," Semiannual reports to DARPA, Caltech Computer Science Technical Reports [5220:TR:86] and [5235:TR:86], 1986.

[5] William C. Athas and Charles L. Seitz, "Multicomputers: Message-Passing Concurrent Computers," *IEEE Computer* 21(8), pp 9–24, August 1988.

[6] William C. Athas, "Fine Grain Concurrent Computation," Caltech Computer Science Technical Report (PhD thesis) [5242:TR:87], May 1987.

[7] William J. Dally, *A VLSI Architecture for Concurrent Data Structures,* Kluwer Academic Publishers, 1987.

[8] Charles L. Seitz, Jakov Seizovic, and Wen-King Su, "The C Programmer's Abbreviated Guide to Multicomputer Programming," Caltech-CS-TR-88-1, January 1988.

The Essence of Distributed Snapshots

K. Mani Chandy*
California Institute of Technology

6 March 1989
Caltech-CS-TR-89-5

1 Introduction

A distributed system has no global clock, and it is the absence of a global clock that makes for several interesting problems, one of which is obviously important, but apparently trivial: 'Record the state of the system.' Recording the state of distributed system is called 'taking a global snapshot' after [2]. If there were a clock, taking global snapshots would be straightforward: Each process records its state or at some predetermined time, and the collection of recorded process states is used to construct a system state.

Global snapshots are useful in a variety of situations [2,3,6]. The goal of this paper is to identify the essential properties of global snapshots so as to simplify proofs of global snapshot algorithms and to aid in the design of new algorithms.

2 A Distributed System

2.1 Standard Definitions

We shall first define a distributed system as in [8].

*Supported in part by DARPA-6202, monitored by ONR N00014-87-K-0745

A prefix of a sequence z is an initial subsequence of z. A prefix-closed set of sequences is a set such that every prefix of a sequence in the set is also in the set.

A system is a set of *components*. A component is a set of *events* and a prefix-closed set of sequences of its events called its set of *computations*.

A *projection* of a sequence v on a component is the sequence obtained from v by deleting all events in v that are not events of the component.

A *system computation* is a sequence v of events of components of the system such that the projection of v on each component of the system is a computation of that component.

Let $w.p$ be a computation of component p, all p. Let P be a set of components. An *interleaving* of a set of component computations $\{w.p \,|\, p \in P\}$ is a sequence, v, of events of components in P, such that the projection of v on p is $w.p$, all $p \in P$.

We use (y, z) for the catenation of sequences y and z.

2.2 Processes and Channels

A component of a distributed system is either a *process* or a *channel*. Distinct processes have disjoint sets of events, and distinct channels have disjoint sets of events.

A channel is *used by* exactly two processes. The events of a channel are events of the processes that use the channel. We shall restrict attention to channels that satisfy the following monotonicity condition.

Let c be a channel used by processes q and r. Let u, v be computations of c, where $u.r = v.r$, and $u.q$ is a prefix of $v.q$. Let e be an event on r.

A Monotonicity Property If (u, e) is a computation of c, then (v, e) is also a computation of c.

Explanantion The monotonicity condition implies that the execution of events on one process cannot inhibit the execution of an event on another process. If a channel c is used by processes q and r, and there is a computation of c in which e is executed on process r after q and r have executed computations a and b respectively, then there is a computation

of c in which e is executed after r has executed b, and q has executed *an arbitrary sequence of events following a.*

Example: Bounded First-In-First-Out Buffers Consider a first-in-first-out buffer, with a capacity of N messages ($N > 0$), shared by a single producer process and a single consumer process. Such a buffer is a channel that has the monotonicity property, as shown by the following informal argument.

The producer can append any message to the buffer if the buffer is not full. The consumer can receive a message m from the buffer if the buffer is not empty and m is the message at the head of the buffer queue. If the producer can produce a message after it has produced i messages and the consumer has consumed j messages, then the producer can produce a message after it has produced i messages, and the consumer has consumed more than j messages. Therefore, the monotonicity property holds with r as the producer and q as the consumer.

By a similar argument, additional production does not prevent the consumer from receiving the message at the head of the buffer; the monotonicity property also holds with r as the consumer and q as the producer. Therefore, the channel has the monotonicity property.

Example: Stacks Next consider a channel which is a stack. Let m be the message at the top of the stack, if the stack is not empty. The consumer can execute the event: Pop the stack and consume m. The producer can execute an event: Push a message m' on top of the stack. Such a buffer does not have the monotonicity property because an event of the producer — push m' on the stack — where $m \neq m'$, can prevent the consumer from executing the event: Pop the stack and receive message m. Therefore, an event on one process can inhibit the execution of an event on the other.

Note: Symmetry of Processes One way of defining channels is in terms of causality: one of the processes sends a message on the channel, and the other receives the message, thus there is a causal relationship between the sending and the receiving of the message. The definition of channels used in this paper is symmetric with respect to both processes; the defini-

tion does not employ the concept of causality. Monotonicity appears to be an important property of channels of distributed systems.

3 The Problem

Restrict attention to one given system. Let z be a finite computation of the system. For ease of exposition, assume that all events in z are distinct. (If events are repeated in z, then number the events, so that the pair — event-name, number — is distinct.) Let $z.p$ be the projection of z on a process p. Let $x.p$ be any prefix of $z.p$. Let S be the set of process computations $\{x.p | p \text{ is a process } \}$.

Set, S, is defined to be a global snapshot in z if and only if there exists a system computation y where:

1. y is an interleaving of the set of process computations $z.p$, and

2. every event in S occurs in y before every event that is not in S.

The problem is to determine simple necessary and sufficient conditions for S to be a global snapshot in z.

Motivation for the Problem Set S is a global snapshot in z if and only if there is a system computation that first takes the system to a state in which each process p has executed $x.p$, and then to the state in which each process p has executed $z.p$. Informally, S is a global snapshot in z if and only if it *could have* happened that all events in S were executed before all events that are not in S. If S is a global snapshot in z, then properties about S can be used to deduce properties about the state of the system after z is executed. Therefore, it is helpful to determine simple conditions that guarantee that S is a global snapshot.

4 A Solution

The obvious algorithm to determine if S is a global snapshot in z is as follows: Since z is finite, enumerate all interleavings of $z.p$, and determine if there is one with the desired properties. This approach is computationally

intractable if the number of processes is large. Next, we present a theorem that helps us to design tractable solutions.

4.1 Compatible Computations

Let c be a channel. Let c be used by processes q and r. Let u and v be computations of q and r respectively. Process computations u and v are defined to be *compatible with respect to* c if and only if there exists an interleaving w of u and v such that the projection of w on c is a computation of c.

Informally, u and v are compatible with respect to c if and only if process computations u and v could have occurred in a computation of a system consisting of only the two processes q and r, and the single channel c.

Example: Bounded First-In-First-Out Buffers Let c be a channel that is a first-in-first-out buffer with a capacity of N where $N > 0$, and where the buffer is initially empty. Let u and v be computations of the processes that send and receive (respectively) on the channel. Then u and v are compatible with respect to c if and only if the sequence of messages received along c in v is a prefix of the sequence of messages sent along c in u, and the number of messages sent along c in u exceeds the number of messages received along c in v by at most N.

Let z and $x.p$ be as in the problem definition, i.e., z is a system computation and $x.p$ is a prefix of $z.p$. Let the producer and consumer for c be q and r respectively. Since z is a system computation, the sequence of receives along c in $z.r$ is a prefix of the sequence of sends along c in $z.q$. Therefore, the sequence of receives along c in $x.r$ is a prefix of the sequence of sends along c in $x.q$ if and only if the number of receives along c in $x.r$ is at most the number of sends along c in $x.q$. Therefore $x.q$ and $x.r$ are compatible with respect to c if and only if

$$0 \leq (nsends - nreceives) \leq N$$

where *nsends* and *nreceives* are the numbers of sends and receives along channel c in $x.q$ and $x.r$ respectively.

4.2 The Snapshot Theorem and its Applications

We shall first give the theorem, discuss its consequences, and then prove it. Let z, $z.p$, $x.p$ and S be as given earlier.

Theorem Set, S, is a global snapshot in z if and only if, for each channel, c:
$x.q$ and $x.r$ are compatible with respect to c, where q and r are the processes that use r.

Applications of the Theorem The proof that S is a global snapshot of an arbitrary system reduces to a proof of compatibility of a pair of process computations for each channel. Let us use this fact in developing algorithms for a couple of problems. The following discussion is very brief and informal, because our goal is only to demonstrate the use of the theorem, rather than to give a thorough exposition of the algorithms.

The Snapshot Algorithm We shall develop the algorithm in [2]. Consider a system in which channels are first-in-first-out and the capacity of a channel is arbitrarily large. Initially all channels are empty. We wish to develop a distributed algorithm to record the global state of the system.

Consider a channel c used by processes q and r, where q sends along c, and r receives along c, and initially c is empty. As discussed earlier, process computations $x.q$ and $x.r$ are compatible with respect to c if and only if the number of receives along c in $x.r$ is at most the number of sends along c in $x.q$. Therefore the problem of algorithm design reduces to this: Guarantee that the number of receives before the receiver records its state is at most the number of sends before the sender records its state, and also guarantee that every process records its state eventually.

One way of doing this is as follows: After a process records its state, it sends a special message called a *marker* along each of its outgoing edges. On receiving a marker a process records its state if it has not done so. At least one process (called the initiator) records its state in finite time; if there is a path (a sequence of channels) from the initiator to all other processes then every process records its state in finite time of the intiator.

Logical Time Consider the same system as in the previous paragraph. Let z be a computation of the system. We are required to give each event in z a unique number, called its logical time, such that the set of events with logical times less than n corresponds to a global snapshot in z, for all n. Let $x.p$ be the prefix of $z.p$ consisting of all events with logical times less than n; we require that the set S (defined as before as $\{x.p\}$) be a global snapshot.

As in the previous problem, the problem of algorithm design reduces to this: Guarantee that for each channel c, the number of messages received along c in $x.r$ is at most the number of messages sent along c in $x.q$, where q and r are the processes that send and receive along c, respectively. This is equivalent to: guarantee that logical times of events are such that the event of receiving a message has a higher logical time than the event of sending that message. One way of doing so is in [7]: put a time-stamp t on each message where t is the logical time of the event that sends the message, and give the event that receives a message a logical time that is greater than the time-stamp of the message.

(The goal for logical time in the seminal paper [7] is different from that given here, because it is based on the concept of causality. Our goal here is limited: to demonstrate a use of the snapshot theorem.)

4.3 Proof of The Snapshot Theorem

Snapshot Theorem Let z be a finite computation of the system. Let $x.p$ be a prefix of $z.p$, all p. Let S be the set of process computations $\{x.p | p$ is a process $\}$. Set S is a global snapshot in z if and only if, for each channel c, computations $x.q$ and $x.r$ are compatible with respect to c, where q and r are the processes that use c.

Proof If $x.q$ and $x.r$ are incompatible with respect to c, then there is no interleaving of $x.q$ and $x.r$ that is a computation of c, and hence S is not a global snapshot. Next, we prove that if for each c, $x.q$ and $x.r$ are compatible with respect to c, where q, r use c, then S is a global snapshot. The proof given here is a generalization of the proofs given in [2,5] which are limited to unbounded first-in-first-out channels.

Define sequence y as follows: y is the permutation of z where all events in S occur before all events that are not in S, and apart from this change, the order of events in z is retained in y. We shall prove that S is a global snapshot by proving that y is a system computation.

Let w be a permutation of z. We shall give an algorithm which starts with $w = z$ and that ends with $w = y$, and where the algorithm maintains the invariant: w is a system computation.

The Algorithm Initially $w = z$. While w contains a pair of adjacent elements d and e, where d occurs before e, and d is not in S, and e is in S: interchange the positions of d and e in w.

Proof of Termination We prove that the algorithm terminates in a finite number of steps by using the metric: the number of pairs (f, g), where event f occurs earlier than event g in w, and f is not in S, and g is in S. The algorithm terminates if and only if the metric is zero, in which case $w = y$.

The metric has a finite value initially, and every step decreases it; hence the algorithm terminates in a finite number of steps.

Proof of the Invariant We prove the stronger invariant that $w.p = z.p$, all processes p, and $w.c$ is a computation of c, all channels c, where $w.d$ and $z.d$ are the projections of w and z, respectively, on component d. The invariant holds initially, because $w = z$. Let w' be the same as w except that d and e are interchanged. Our proof obligation is to show that w' satisfies the invariant if w satisfies it.

Since $x.q$ is a prefix of $z.q$ and since $w.q = z.q$, it follows that $x.q$ is a prefix of $w.q$. Therefore, if two adjacent events in w are on the same process, q, and the first of the two events is not in $x.q$, then the second is not in $x.q$ either. Since d is not in S and e is in S, it follows that d and e cannot be on the same process. Let d be on process q and let e be on process r, where $r \neq q$.

Since d and e are on different processes, $w'.p = w.p$, all p, and therefore $w'.p = z.p$.

If d and e are on different channels, then the projections of w and w' on each channel are identical, and hence the invariant holds for w'. Therefore, we need only consider the case where d and e are on the same channel; let this channel be c. Our only remaining proof obligation is to show that $w'.c$ is a computation of c.

Let t be the prefix of w preceding d in w. Then (t, d, e) is a prefix of w, and (t, e, d) is a prefix of w'.

Since $x.q$ and $x.r$ are compatible with respect to c, there exists an interleaving h of $x.q$ and $x.r$ such that the projection of h on c is a computation of c. Event e is in $x.r$, and therefore is in h. Define u as the prefix of h preceding e. Therefore, (u, e) is a prefix of h, and hence it is a computation of c. Since both $u.r$ and $t.r$ are the prefixes of $x.r$ that precede e, it follows that $u.r = t.r$. Since d is not in $x.q$, it follows that $x.q$ is a prefix of $t.q$. Since $u.q$ is a prefix of $x.q$, it follows from the transitivity of the prefix relation that $u.q$ is a prefix of $t.q$. From the monotonicity property, the projection of (t, e) on c is a computation of c.

Applying the monotonicity property again, the projection of (t, e, d) on c is a computation of c, since the projections of (t, d) and (t, e) on c are computations of c.

Let m be the length of the sequence (t, e, d). We shall prove by induction on k, that for $k \geq m$: $g'.c$ is a computation of c where g' is the prefix of w' of length k.

Base Case: $k = m$. This case has already been proved.

Induction Step: Let f be the $(k+1)$-th event in w. Our proof obligation is to show that the projection of (g', f) on c is a computation of c. Let g be the prefix of w of length k. The projection of (g, f) on c is a computation of c because (g, f) is a prefix of w. From the induction hypothesis, the projection of g' on c is a computation of c. For $k \geq m$: $g.q = g'.q$ and $g.r = g'.r$. If f is on c, the from the monotonicity property of c, the projection of (g', f) on c is a computation of c. If f is not on c, then the projection of (g', f) on c is the same as the projection of g' on c, and the result follows.

5 Partial Snapshots

There are some problems in which a snapshot of some subset of processes and channels is useful, and a global snapshot of all processes and channels is not necessary. We define a *partial* snapshot of a set of processes, Q, in a manner analogous to the definition of a global snapshot. Let z be a system computation. Let $x.p$ be a prefix of $z.p$. Let S be the set of process computations $\{x.q | q \in Q\}$. Set S is defined to be a partial snapshot in z if and only if there exists a system computation y where:

1. y is an interleaving of the set of process computations $z.p$, all processes p, and

2. for each process q in Q, the events in $x.q$ appear in y before the events of q that are not in $x.q$.

A partial snapshot is a global snapshot if Q is the set of all processes.

Next, we shall define a class of problems for which partial snapshots are helpful.

5.1 Termination Problems

Let w be a system computation. Set, Q, is defined to have terminated after w if and only if,

1. for all events e, and all processes q in Q, if $(w.q, e)$ is a computation of q, then e is an event on a channel between q and a process in Q, and

2. for all channels c between processes in Q, there is no event e such that $(w.c, e)$ is a computation of c.

Informally, the first condition says that after a process q has executed $w.q$ it can only execute events on channels connecting it to other processes in Q. The second condition says that there is no extension of a computation of a channel c between processes in Q after w. The two conditions, together, imply that the processes in Q cannot execute events after system computation w.

Example: Full-Buffer Deadlock Consider a system in which each channel is a buffer with a capacity of N, where $N > 0$. A process is either waiting or active. A waiting process is waiting to send a message on any one of a set of full outgoing channels (i.e., channels containing N messages); a waiting process continues to wait until at least one of the channels that it is waiting for becomes not full, and it then sends a message on that channel and becomes active. Waiting processes do not receive messages. A set of processes, Q, is said to be deadlocked if and only if:

1. each channel between processes in Q is full (or equivalently, the number of messages sent on the channel exceeds the number of messages received on the channel by N), and

2. each process in Q is waiting to send messages only along channels to other processes in Q.

The problem is to detect a deadlocked state.

A dual of this problem is obtained by replacing 'full' by 'empty', 'send' by 'receive', and 'outgoing' by 'incoming' in the previous problem.

Next, we give a theorem that shows how partial snapshots may be employed.

5.2 Termination Detection Theorem

Let v be a system computation such that Q terminates after v. If z is a system computation such that for all q in Q, $v.q$ is a prefix of $z.q$, then $v.q = z.q$, for all q in Q.

Proof We prove by induction on the length of prefixes u of z, that $u.q$ is a prefix of $v.q$, for all q in Q. In particular, we prove that $z.q$ is a prefix of $v.q$. Since $v.q$ is a prefix of $z.q$, it follows that $v.q = z.q$.

Base Case u is the empty sequence. The result holds, trivially.

Induction Step Consider a channel c used by processes q and r, where both q and r are in Q. Let $u.r = v.r$ (and $u.q$ is a prefix of $v.q$ from the induction hypothesis). From the monotonicity property, for all events

e on c, if the projection of (v, e) on c is not a computation of c, then the projection of (u, e) on c is not a computation of c. Since Q terminates after v, the projection of (v, e) on c is not a computation of c. Hence, if $u.r = v.r$, for all events e on c, the projection of (u, e) on c is not a computation of c. Since Q terminates after v, the only events on r after $v.r$ are events on channels to other processes in Q. Hence, if $u.r = v.r$, there is no event e on r such that (u, e) is a system computation.

From the arguments of the last paragraph, if (u, e) is a computation of z, then e is on a process r such that $u.r \neq v.r$. Since $u.r$ is a prefix of $v.r$, the length of $u.r$ is less than the length of $v.r$ In this case, $(u.r, e)$ is a prefix of $v.r$, since both $(u.r, e)$ and $v.r$ are prefixes of $z.r$, and the length of $(u.r, e)$ is at most the length of $v.r$.

5.3 Applications of the Theorem

The termination detection theorem tells us that old data $(v.q)$ is current (because $v.q = z.q$) if the old data shows that Q has terminated. This suggests the following class of algorithms for termination detection; this class includes algorithms in [1,4,9].

A Class of Algorithms for Termination Detection The algorithms employ a set of process computations $\{v.q | q \in Q\}$ and have the following specification.

Invariant $v.q$ is a prefix of $z.q$ where z is the system computation up to the current point.

Progress For all q, if the current value of $z.q$ is, say, $y.q$, then eventually $y.q$ is a prefix of $v.q$. (The progress property says that the process computations $v.q$ get updated: eventually, $v.q$ will include the current value, $y.q$, of $z.q$.)

The algorithm determines that Q has terminated if Q terminates after $\{v.q | q \in Q\}$, i.e., if Q terminates after a system computation, y, where $y.q = v.q$, all q in Q.

Correctness The proof of correctness of this class of algorithms is as follows. From the invariant and the theorem, if Q has terminated after $\{v.q|q \in Q\}$, then Q has terminated after z. From the progress property, if Q terminates after z, then eventually $v.q = z.q$, and hence eventually, the algorithm determines that Q has terminated.

Example Next, we give an example of algorithms with the invariant and progress properties given earlier. To detect termination of Q, a token is sent from process to process in Q, in such a manner that the token visits every process in Q repeatedly. The token carries with it a set of process computations $\{v.q|q \in Q\}$. Initially, $v.q$ is the empty sequence. When the token arrives at a process q, it updates this set, by replacing the value of $v.q$ in the set by its current computation, and q determines that Q has terminated if Q terminates after $\{v.q|q \in Q\}$.

Various optimizations are possible in applying this method to detect a specific form of termination. For example, to detect full-buffer deadlock, it is not necessary for the token to carry the entire computation $v.q$; it is sufficient for the token to contain the number of messages sent and received on each channel by q in $v.q$, and the set of processes for which q is waiting.

6 Summary

The paper presents necessary and sufficient conditions for a set of process computations to be a global snapshot. The condition is that for every channel, the computations in the snapshot of the processes that use the channel, are compatible with respect to the channel. The condition is helpful in the development of algorithms.

The paper also presents the concept of partial snapshots and demonstrates its utility.

References

[1] Chandy, K. M.[1987] 'A Theorem on Termination of Distributed Systems', TR-87-09, March 1987, Dept. of Computer Sciences, The University of Texas at Austin, Austin, Texas 78712-1188.

[2] Chandy, K. M., and L. Lamport [1985]. 'Distributed Snapshots: Determining Global States of Distributed Systems,' ACM TOCS, 3:1, February 1985, pp. 63-75.

[3] Chandy, K. M. and J. Misra [1988] *Parallel Program Design: A Foundation*, Addison-Wesley, Reading, Massachusetts, 1988.

[4] Chandy, K. M. and J. Misra [1988] 'On Proofs of Distributed Algorithms with Application to the Problem of Termination Detection', submitted to Distributed Computing [1987].

[5] Dijkstra, E. W. [1985]. 'The Distributed Snapshot of K. M. Chandy and L. Lamport,' in Control Flow and Data Flow, ed. M. Broy, Berlin: Springer-Verlag, 1985, pp. 513-517.

[6] Fischer, M. J., N. D. Griffeth, and N. A. Lynch [1982]. 'Global States of a Distributed System,' IEEE Transactions on Software Engineering, SE-8:3, May 1982, pp. 198-202.

[7] Lamport,L.[1978] 'Time, Clocks and the Ordering of Events in a Distributed System,' C.ACM, 21:7, July 1978, pp 558-565.

[8] Hoare, C. A. R. [1984]. *Communicating Sequential Processes*, London: Prentice-Hall International, 1984.

[9] Raynal, M., J.-M. Helary, C. Jard, and N. Plouzeau [1987]. 'Detection of Stable Properties in Distributed Applications,' in Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing, 1987, pp. 125-136.

To be published in *Proc 1989 ACM Symposium on Parallel Algorithms and Architectures*

A Framework for Adaptive Routing in Multicomputer Networks

John Y. Ngai and Charles L. Seitz
Department of Computer Science, California Institute of Technology
Submitted to the 1989 ACM Symposium on Parallel Algorithms and Architectures*

Introduction

Message-passing concurrent computers, also known as *multicomputers*, such as the Caltech Cosmic Cube [1] and its commercial descendents, consist of many computing nodes that interact with each other by sending and receiving messages over communication channels between the nodes [2]. The communication networks of the second-generation machines, such as the Symult Series 2010 and the Intel iPSC2, employ an *oblivious* wormhole routing technique [3,4] that guarantees deadlock freedom. The message latency of this highly evolved oblivious technique has reached a limit of being capable of delivering, under random traffic, a *stable* maximum sustained throughput of $\approx$ 45 to 50% of the limit set by the network bisection bandwidth. Any further improvements on these networks will require an *adaptive* utilization of available network bandwidth to diffuse local congestions.

In an adaptive multipath routing scheme, message routes are no longer deterministic, but are continuously perturbed by local message loading. It is expected that such an adaptive control can increase the throughput capability towards the bisection bandwidth limit, while maintaining a reasonable network latency. While the potential gain in throughput is at most only a factor of 2 under random traffic, the adaptive approach offers additional advantages, such as the ability to diffuse local congestions in unbalanced traffic, and the potential to exploit inherent path redundancy in richly connected networks to perform *fault-tolerant* routing. The rest of this paper consists of an examination of the various feasibility issues and results concerning the adaptive approach studied by the authors. A much more detailed exposition, including results on performance modeling and fault-tolerant routing, can be found in [5].

*The research described in this paper was sponsored in part by the Defense Advanced Research Projects Agency, DARPA Order number 6202, and monitored by the Office of Naval Research under contract number N00014-87-K-0745, and in part by grants from Intel Scientific Computers and Ametek Computer Research Division.

The Adaptive Cut-Through Model

It is clear that in order for the adaptive multipath scheme to compete favorably with the existing oblivious wormhole technique, it must employ a switching technique akin to *virtual cut-through* [6]. In cut-through switching and its blocking variant, which is used in oblivious wormhole routing, a packet is forwarded immediately upon receipt of enough header information to make a routing decision. The result is a dramatic reduction in the network latency over the conventional store-and-forward switching technique under light to moderate traffic. We now describe a simple cut-through switching model that provides the context for the discussion of issues involved in performing adaptive routing in multicomputer networks. The following definitions develop the notation that will be used throughout the rest of the paper.

Definition 1 A *Multicomputer Network*, M, is a connected undirected graph, $M = G(N, C)$. The vertices of the graph, N, represent the set of computing nodes. The edges of the graph, C, represent the set of *bidirectional* communication channels.

Definition 2 Let $n_i \in N$ be a node of M. The set, $C_i \subseteq C$, is the set of bidirectional channels connecting n_i to its neighbors in M.

Definition 3 The *width*, W, of a channel is the number of data wires across the channel. A *flit*, or *flow control unit*, is the W parallel bits of information transferred in a single cycle. The flit is the unit used to measure the length of a packet.

Definition 4 Given a pair of nodes, n_i and n_j, the set, Q_{ij}, of *routes* joining n_i to n_j is the fixed and predetermined set of directed *acyclic* paths from the source node, n_i, to the destination node, n_j.

Definition 5 For each destination node, n_j, the *profitable* channel set $R_{ij} \subseteq C_i$ is the subset of channels connected to n_i, where $c_k \in R_{ij} \Rightarrow c_k \in q_m \in Q_{ij}$. In other words, forwarding a packet along the routes in Q_{ij} is equivalent to sending it out through a profitable channel in R_{ij}.

Definition 6 For each node $n_i \in N$, the *Routing Relation* $R_i = \{(n_j, c_k) : n_j \in N - \{n_i\}, c_k \in R_{ij}\}$ defines for each possible destination node $n_j \in N$ its corresponding profitable channel set, R_{ij}.

Definition 7 The actual path a packet traverses while in transit in the communication network is referred to as the *trajectory* of the packet. Packet trajectories are identical to the packet routes in oblivious routing schemes but are *non-deterministic* in our adaptive formulation.

We assume the following:

- Long messages are broken into packets that are the logical data entities transferred across the network.
- Packets are of fixed length; *ie*, packet length $= L$, where L is a network-wide constant.
- Complete routing information is included in the header flit of each packet.
- Packets are forwarded in virtual cut-through style.
- A message packet arriving at its destination node is consumed. This is commonly known as the *consumption assumption.*
- A node can generate messages destined to any other node in the network.
- Nodes can produce packets at any rate subject to the constraint of available buffer space in the network, and packets are source queued.
- Each node in the network has complete knowledge of its own routing relation.

Figure 1 presents our view of the structure of a node in a multicomputer network. Conceptually, a node can be partitioned into a computation subsystem, a communication subsystem, and a message interface. For our purpose, the computation subsystem serves as the producer and consumer of the messages routed by the communication subsystem of the node. The message interface consists of dedicated hardware that handles the overhead in sending, receiving, and reassembling of message packets.

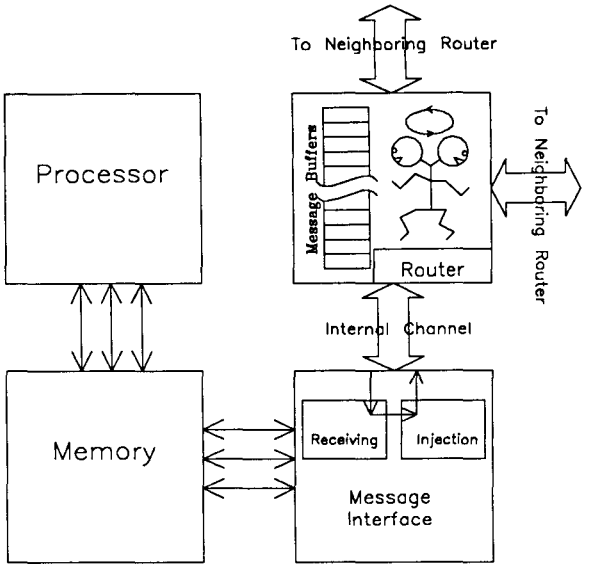


Figure 1: Structure of a node.

Internally, the communication subsystem consists of an adaptive control and a small number of message-packet buffers. Routing decisions are made by the adaptive control, based entirely on locally available information. The bidirectional channel assumption is adopted to allow the network to exploit locality in general message-communication patterns. Furthermore, it assures an identical number of input and output communication channels in each node, irrespective of the underlying network topology. The fixed-packet-length assumption is not essential and can be replaced by a *bounded*-packet-length assumption; *ie*, packet length $\leq L$, without invalidating any of our major results. It is adopted solely to simplify our subsequent exposition.

Communication Deadlock Freedom

In any adaptive routing scheme that allows arbitrary multipath routing, it is necessary to assure freedom from communication deadlock. Communication deadlock is caused generically by the existence of *cyclic* dependencies among communication resources along the message routes. Methods to prevent communication deadlock have been intensively researched and many schemes exist; of these, the methods of structured buffer pools [7] and virtual channels [8] are representative. In essence, all of these methods approach the problem by *re-mapping* any dependency that is potentially cyclic into a corresponding *acyclic* dependency structure. These methods employ restructuring techniques that require information of a global, albeit static, character. In contrast, a very simple technique that is *independent* of network size and topology, through vol-

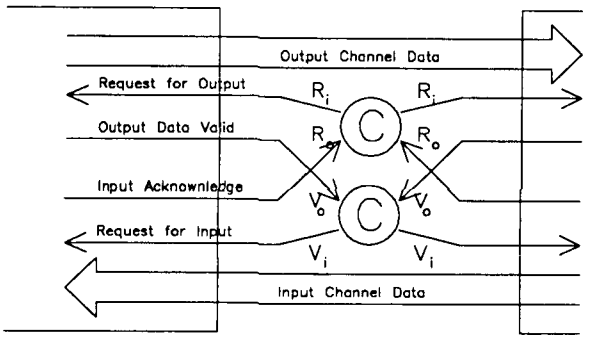


Figure 2: Two-phase protocol signaling.

untary *misrouting*, was suggested in [9] for networks that employ data exchange operations. Such a preemption technique utilizes only local information, and is dynamic in character. It prevents deadlock by *breaking* the potentially cyclic communication dependencies into disjoint paths of unit length. Voluntary misrouting can be applied to assure deadlock freedom in cut-through switching networks, provided the input and output data rates across the channels at each node are tightly *matched*. A simple way is to have all bidirectional channels of the same node operate *coherently* under the protocol described next.

The Coherent Protocol. We now describe the channel data-exchange protocol in detail. It is used to match the transfer rates across all channels of the same node. The protocol employs four control signals per channel, two from each of the communicating partners, and is completely symmetric between the partners. The signaling events for a channel $c \in C$ are:

- R_o — output event to the communicating partner indicating that this node is Ready to accept another input flit from its partner. It also serves as an acknowledgment to its partner for the successful completion of the previous transfer cycle.

- R_i^c — input event from the communicating partner indicating that the partner is Ready to accept another output flit from this node. It is also an acknowledgment from the partner for the successful completion of the previous transfer cycle.

- V_o — output event to the communicating partner indicating that the data flit values currently held at the output channel of this node are Valid and its partner should latch in the held values.

- V_i^c — input event from the communicating partner indicating that the data flit values currently asserted at the input channel of this node are Valid and the node should latch in the held values.

We proceed to define our handshaking protocol across channels of a node $n_k \in N$, in a CSP-like notation [10]:

$$*[\quad R_o;\quad [\forall c \in C_k,\ R_i^c];\quad \textit{apply out data};$$
$$\quad\ V_o;\quad [\forall c \in C_k,\ V_i^c];\quad \textit{latch in data};\quad]$$

Observe that R_o and V_o denote, respectively, the *unique* outgoing Ready and data Valid signaling event to all neighbors of n_k. This enforces the matching of outgoing data rates. On the other hand, the matching of incoming data rates is enforced through the *synchronized wait* for the R_i^c and V_i^c signaling events from all neighbors. The handshaking events R_o, R_i^c interlock with the events V_o, V_i^c to guarantee the stability and strict alternation of each other. The *initial* state of a channel has both directions of the channel ready to accept a new data flit and proceeds thereafter in a *demand-driven* fashion. Figure 2 shows a possible conceptual realization of the protocol under the two-phase signaling convention [11] popular for off-chip communication. Since all the handshaking events defined are local between nearest neighbors, a network following the coherent protocol is *arbitrarily extensible*.

Observe that under cut-through switching, a packet can span many different channels. An outgoing channel occupied by a packet may not be able to assert V_o until after valid data has been asserted by the corresponding incoming channel occupied by the packet, hence, induces matching of data rates across the two occupied channels. The notion of coherency introduced here is a natural way to accommodate such potential dependencies among the various channels of a node under cut-through switching. Another notion that arises naturally is that of a *null flit*. To effect a transfer of data in one direction of a channel while the opposite direction is idle, the receiving partner is required to transmit a null flit in order to satisfy the convention dictated by the exchange protocol.

Deadlock Freedom. We now demonstrate that to assure communication deadlock freedom for networks operating under the coherent protocol, it is sufficient to employ voluntary misrouting to prevent potential buffer overflow. To proceed, observe that routing under the cut-through switching model imposes the following integrity constraints:

1. Packets must always be forwarded to neighbors with their header flits transmitted first. In particular, voluntary misrouting of any internally buffered packet must start from the header flit of the selected packet.

2. Once the flit stream of a packet has been assigned a particular outgoing channel, the assignment must be maintained for the remaining cycles until the entire packet has been transmitted.

These constraints exist because all of the necessary routing information of a packet is encapsulated in the packet header. Interrupting a packet flit stream mid-transfer would render the latter part of the packet undeliverable. To establish deadlock freedom, it is sufficient to show that each node can *independently* complete each transfer cycle and initiate a new one, in a bounded period, without violating the stated constraints. We now show that as long as we have an equal number of input and output channels per node, a condition satisfied readily by our bidirectional channel assumption, we can always satisfy the stated logical requirements, and, hence, assure freedom from communication deadlock.

Theorem 1 Let M denote a coherent multicomputer network where each node has an equal number of input and output channels. If M employs voluntary misrouting to prevent potential buffer overflow, then it is free from deadlock.

Proof. We need to show that buffer overflow can always be prevented by misrouting without violating the cut-through switching integrity constraints. We proceed with a counting argument: Let d denote the number of channels at a node. During a protocol cycle, there may be as many as $n^* \leq d$ new data flits arriving at the input channels. A fraction of these, $0 \leq n' \leq n^*$, are new header flits; the remaining $n^* - n'$ are non-header flits of arriving packets. Of these non-header flits, a fraction of them, $0 \leq n'' \leq n^* - n'$, belong to packets that have already been assigned output channels, and the remaining $n^* - n' - n''$ flits belong to waiting packets that are buffered inside the node. Therefore, the node has at least a total of $n' + (n^* - n' - n'')$ headers flits that are eligible for immediate routing. Hence, in the following cycle, a node can find at least $n' + (n^* - n' - n'') + n'' = n^*$ flits that can be transmitted or misrouted without violating the cut-through switching integrity constraints. This assures that no buffer overflow will occur. The node can always complete its protocol cycles in bounded time; hence, the network is free from deadlock. ■

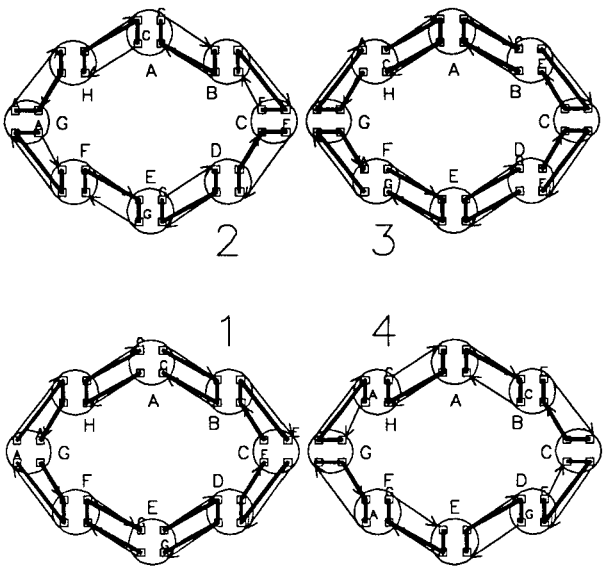


Figure 3: Livelock due to bad assignments.

Since the validity of the above proof does not depend on a node's storage capacity, deadlock freedom is established independent of the amount of available buffer space. The simple criterion of having an equal number of input and output channels is sufficient to assure deadlock freedom for a coherent network. In practice, additional buffers are needed in order to inject packets into the network, and to improve the network performance.

Network Progress Assurance

The adoption of voluntary misrouting renders communication deadlock a non-issue. However, misrouting also creates the burden to demonstrate progress in the form of message delivery assurance. In particular, a network can run into a *livelock*. Consider the sequence of routing scenarios depicted in figure 3 for a bidirectional ring consisting of eight nodes and eight packets. Eack of the packets consists of four data flits that span multiple channels and internal buffers. Suppose the nodes employ the following simple, deterministic, packet-to-channel assignment rule: Whenever two incoming packets both request the same outgoing channel, the packet from the clockwise neighbor always wins. Given that, initially, nodes A, C, E, and G each receive two packets destined to nodes that are, respectively, distance two from them in the clockwise direction, after four routing cycles, the packets are all back to where they started! This example illustrates that packets can be forever denied delivery to their destinations even in the absence of communication deadlock.

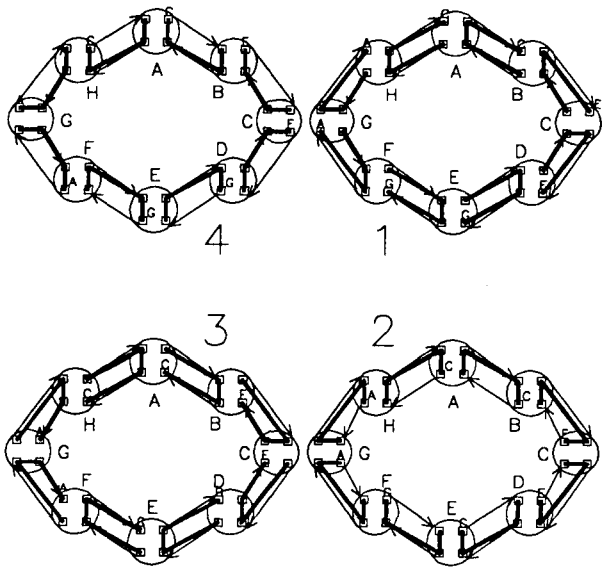


Figure 4: Livelock due to lack of assignments.

Channel-access competitions are, however, not the only type of conflict that can lead to livelock. Consider the situations depicted in figure 4 for the same bidirectional ring network. The traffic patterns are coincidental in such a way that none of the packets will ever have a chance to select its own output channel; rather, at every node, each packet must be forwarded along the only remaining channel, in compliance with the voluntary misrouting discipline, in order to avoid deadlock. It is clear that no matter what assignment strategy one chooses, it is impossible to break this kind of livelock without adding extra buffers per node. In other words, additional measures and resources have to be introduced in order to assure progress, *ie*, delivery of packets, in the network.

Buffering Discipline and Requirement. In order to assure packet delivery in spite of voluntary misrouting, extra buffers are required to store packets temporarily. In particular, sufficient buffers must be provided to allow the adaptive control to give *any* newly arriving packet a chance to escape preemption if so determined by the assignment algorithm. We now demonstrate the existence of such a solution using a bounded number of buffers. We assume the following buffering discipline:

1. Storage is divided into buffers of equal size; each is capable of holding an entire message packet.

2. Each buffer has exactly one input and one output port; this permits simultaneous reading and writing. A good example is a *FIFO* queue of length L.

3. Except as stated below, a buffer can be occupied by only one packet at a time. Oftentimes a packet may not fill its entire buffer, as in case of a partial cut-through. Such a packet occupies both the input and output ports to the buffer.

4. A buffer can be used temporarily to store two packets at a time, if and only if, one of them is leaving through the output port connected to an output channel, and the other is entering through the input port connected to an input channel.

Let b and d denote, respectively, the number of buffers and channels, *ie*, the *degree* at each node. First, we observe that, given the above buffering discipline, we must have $b \geq d$. To see this, assume that $L \gg d$, and consider the following sequence of events at a node with all buffers initially empty: At cycle $t = 0$, a packet P_0 arrives and is forwarded to its requested output channel c^* at cycle $t = 1$. Then, at cycles $t = L-d$ up to $t = L-2$, a total of $d-1$ packets, P_i, $i = 1, \ldots, d-1$, arriving one after another in these $d-1$ consecutive cycles, all requesting the same output channel c^*. Finally, at cycle $t = L+2$, another packet P_d arrives, requesting the same channel c^*. The worst case happens when the assignment algorithm always favors the latest arriving packet requiring it to stay and avoid preemption, and having each occupy a distinct buffer. Given the above arrival sequence, at cycle $t = L+1$, packet P_{d-1} will be forwarded through c^*, which now becomes idle. As a result, each packet from P_1 up to P_d would have to be temporarily stored as it comes in. Since each packet must be allocated to a distinct buffer, we must have $b \geq d$. We now show that having $b = d$ buffers is also sufficient.

Theorem 2 Let M be a coherent network where each node has b packet buffers inside the router operating under the stated assumptions. Then $b = d$ buffers per router is necessary and sufficient to always allow at least one packet, chosen arbitrarily by the assignment algorithm at each node, to escape preemption.

Proof. Necessity follows immediately from the preceding discussion. We proceed to establish sufficiency through a counting argument. Observe that a node is required to consider misrouting of packets in the next cycle only when there are new packets arriving at the current cycle. Figure 5 depicts an accounting of all possible cases of buffer allocation at the end of any such routing cycle. Let n_1 up to n_7 denote, respectively, the number of packets or buffers in each case; and n_0 denote the number of

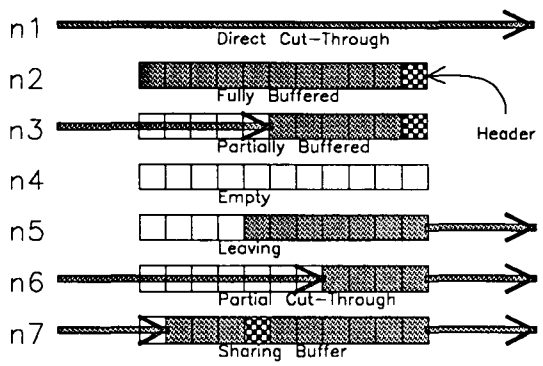


Figure 5: Accounting of buffer allocations.

newly arrived packets. Then, for inputs, we have $n_0+n_1+n_3+n_6+n_7 \leq d$; for outputs, we have $n_1+n_5+n_6+n_7 \leq d$. Let P^* denote the privileged packet chosen by the assignment algorithm to stay behind and avoid misrouting in the following cycle. P^* must be either a newly arrived packet or an already buffered packet. If P^* is a buffered packet, then a newly arriving packet either finds an idle output channel to directly cut through the node; or else we must have $n_1+n_5+n_6+n_7 = d \Rightarrow n_5 \geq n_0+n_3$, which, in turn, implies that there will always be an available buffer ready to accept it. On the other hand, if P^* is a newly arriving packet, then either $n_4+n_5 > 0$, and, hence, there is a buffer ready to accept it; or else we must have $n_2+n_3+n_6+n_7 = b = d$. This, together with the above inequality on inputs, $\Rightarrow n_2 \geq n_0+n_1 \Rightarrow n_2 > 0$. Furthermore, $n_0 > 0 \Rightarrow n_1+n_6+n_7 < d$. In other words, the packet will be able to find at least one buffer with a full idle packet as well as an idle output channel to preempt this idle packet and thus make room for itself. This establishes the sufficiency condition. ■

The trick in allowing the escape of misrouting for any arbitrarily chosen packet is to provide at least a critical, minimum number of buffers that is sufficient to assure either that empty buffers still exist, or that all buffers have been occupied, and, hence, there is some other packet that can be misrouted instead. The particular number required depends on the adopted buffering structure and discipline, and adding more buffers per node will allow the assignment algorithm to operate with more flexibility and perform better. In any case, by having a sufficient number of buffers, competition of profitable channel access is transformed into a competition for the right to stay behind and wait until the winner's profitable channel becomes available, at which time, it will be forwarded. Hence, winners that have been chosen by the assignment algorithm will have the chance to follow the actual paths determined by the routing relations. In a sense, assurance of packet delivery has now been reduced to that of picking *consistent* winners across the network.

Packet-Priority Assignments. An effective scheme for picking consistent winners that is independent of any particular network topology is to resolve the channel-access conflicts according to a *priority* assignment. In particular, the process of forwarding a packet towards its destination can be viewed as a sequence of actions performed to reduce the packet's distance from destination, provided that the set $\mathcal{R} = \{R_i\}$ of routing relations is defined in terms of an underlying metric of the network. In this case, as the result of a channel-access conflict, the winner will be routed along a profitable channel, hence decreasing its distance from the destination. The losers, depending on whether they are misrouted along the remaining unprofitable channels, may or may not increase their distance from destination. Ideally, one would prefer a strict monotonic decrease of distance to destination for each packet routed in the network. As this is impossible under our adaptive model, the alternative is to ensure monotonic decrease over a sequence of exchanges involving multiple packets. This can be achieved by giving higher priority to packets with shorter distances from destination over those with longer distances as follows:

$$P_1 > P_2 \Longleftrightarrow D_1 < D_2$$

where P is a packet's priority and D its distance from destination. We now show that this is sufficient to guarantee livelock freedom.

Theorem 3 A packet-to-channel assignment strategy that observes the defined distance priority, together with the set $\mathcal{R}$ of metric-based routing relations, guarantees livelock freedom in a network.

Proof. At the beginning of a routing cycle, let $D > 0$ be the minimum packet distance from destination. During this cycle, a packet with distance D competes with other packets for channels leading to its destination. If it wins the competition, it will be forwarded along a profitable channel within L cycles. It it loses, it must be to another packet also distance D away from its destination, according to the defined priority. In both cases, the minimum distance is reduced to $< D$ within L cycles. Therefore, D will eventually be reduced to zero, in which case a successful packet delivery occurs and the above argument can be applied again to assure repeated deliveries. This establishes livelock freedom. ■

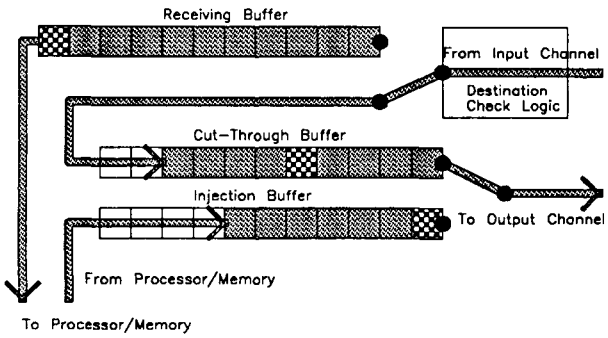


Figure 6: Inside the message interface.

Observe that although the distance priority alone suffices to guarantee global progress in a message network, no corresponding statement can be made concerning each individual packet. This is because it is possible for packets that are far away from their destinations to be repeatedly defeated by newly injected packets that are closer to their respective destinations. A more complex priority scheme that assures delivery of *every* packet can be obtained by augmenting the above simple scheme with *age* information, with higher priorities assigned to older packets:

$$(A_1, D_1) > (A_2, D_2) \iff (A_1 > A_2) \vee ((A_1 = A_2) \wedge (D_1 < D_2))$$

where A is a packet's *age*, that is, the number of routing cycles elapsed since the injection of the packet. Empirical simulation results indicate that the simple distance assignment scheme is sufficient for almost all situations, except under an extremely heavy applied load.

Network-Access Assurance

A different kind of progress assurance that requires demonstration under our adaptive formulation is the ability of a node to inject packets eventually. Because of the requirement to maintain strict balance of input and output data rates, a node located in the center of heavy traffic might be denied access to the network indefinitely. Figure 6 depicts a possible conceptual realization of a message interface. Its operation is similar to the register insertion ring interface described in [12]. It uses two FIFO buffers that can be connected to the output channel towards the network via a switch. Whenever the node has a packet to transmit, it loads the packet into the injection buffer as soon as the buffer becomes empty. When message traffic arrives from the network input channel, it passes through the destination check logic, which redirects any traffic destined to this node to the node memory. Any remaining passing traffic is loaded into the cut-through buffer, which is normally connected to the output channel. Whenever the cut-through buffer becomes empty, the control logic checks to see if there is an output packet waiting for injection. In such case, the switch is toggled so that the output channel is connected to the injection buffer and the injection proceeds. As the output packet is being forwarded, any passing traffic is loaded into the cut-through buffer. The switch connection is flipped back to the cut-through buffer after injection has been finished, and the process repeats. The main interesting property of the message interface for our current discussion is that it provides the mechanism to capture and accumulate interpacket gaps, which need not be contiguous, as empty spaces inside the cut-through buffers. When enough space has been collected, *ie*, the entire packet length, hence, an entire empty buffer, another new packet can be injected into the network. With such a mechanism, the question of assuring eventual packet injection is translated into that of assuring arrival of enough interpacket gaps whenever a node has a packet injection outstanding.

Round-Trip Packets. One simple way to assure network access is to have each packet delivered by the network be returned to its original sender upon arrival at its destination. Since each message interface starts with an empty injection buffer, consumption of its own round-trip packets will always restore its ability to inject the next source-queued packet. More sophisticated versions of such a scheme will use several cut-through buffers, and will demand that packets be returned only if the stock of empty cut-through buffers has been depleted below a predetermined threshold. In this way, the number of round-trip packets can be dramatically reduced when traffic is relatively moderate. Unfortunately, as traffic density increases, the population of round-trip packets also increases, thus further decreasing useful network bandwidth.

Packet-Injection Control. A different scheme that does not incur this overhead is to have the nodes maintain a bounded synchrony with neighbors on the total number of injections. Nodes that fall behind will, in effect, prohibit others from injecting until they catch up. We shall adopt the convention that a node having no packet to inject has a *null* packet queued up; *ie*, during each routing cycle, every node either has a null or real packet ready to inject or else is in the process *of injecting* a real packet. The null-packet convention is required to prevent quiescent nodes that do not have any packet to inject from blocking injections in the

active nodes. Our scheme is to introduce *local synchronization* among neighboring nodes such that the total number of packets injected by a node after each routing cycle will not differ by more than K, a positive constant, from those of its neighbors. We assume that each node explicitly maintains records of the total number of packet injections made by each of its neighbors, measured *relative to that of its own,* and that the information required to update these records in each node is exchanged on separate direct links between the message interfaces among neighbors. A node is allowed to inject its queued packet only if its own number of total injections is fewer than K packet injections ahead of its minimum neighbor. Nodes that are allowed to inject will examine their queued packets. Null packets are always injected by convention, whereas real packets are injected only if the injection mechanism described previously finds at least one empty buffer available to absorb the injection transient. We now show that, with eventual delivery of the packets already injected, this injection synchronization protocol establishes cooperation among the nodes to assure the eventual occurrence of empty cut-through buffers in the message interface for nodes that have real packets waiting for injection as permitted by the protocol.

Lemma 4 A node that has a packet waiting for injection that is permissible under the above injection protocol will eventually inject.

Proof. Observe that, by convention, if the pending packet is null, the node is able to inject immediately, so that the lemma is true vacuously. We now proceed to establish its validity for real packets. Suppose, to the contrary, that a particular node, $n \in N$, is blocked from injection indefinitely because the injection mechanism cannot accumulate sufficient empty buffer space to absorb the injection transient. Our injection protocol then dictates that its neighbors also will be blocked indefinitely from injecting. These, in turn, indefinitely block their neighbors, and so on. Given a finite network, all nodes are eventually blocked from any further injection, and eventually *no* new packet can enter the network. Given the eventual delivery guarantee for packets already injected, ultimately the network will be void of packets; at that point, the input channel to the interface of n will become idle, thus enabling it to resume the accumulation of empty spaces inside the cut-through buffer. Eventually, it will have collected enough spaces to enable the injection of its queued packet into the network. This contradicts the original indefinite blocking assumption of n, hence establishing the validity of the lemma. ■

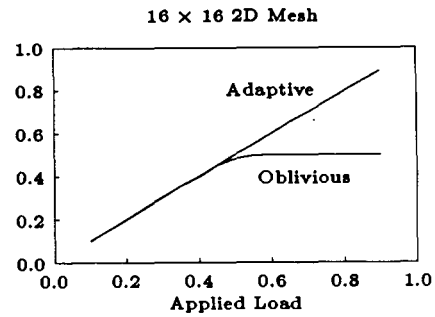


Figure 7: Throughput versus applied load.

We are now ready to show that by following the above injection protocol every individual node will eventually be permitted to inject, and, hence, according to the above lemma, will eventually inject. Specifically, let M be a network, and let T_i denote the total number of packet injections from node $n_i \in N$ since initialization. We now prove that T_i is strictly increasing over time.

Theorem 5 Given the injection protocol and a finite network that is livelock free, the total number of packet injections for each node strictly increases over time.

Proof. During a routing cycle, let $t = \min_{n_i \in N} T_i$ denote the minimum among numbers of packet injections since initialization, taken over all the nodes of the network, and let $S = \{n_i \in N | T_i = t\}$ denote the set of nodes that have recorded the minimum number of packet injections since initialization. Since $K > 0$, according to our protocol, every node $n \in S$ is permitted to inject. Lemma 4 then guarantees eventual injections from all of the nodes in S; hence, t, the minimum number of packet injections per node, is guaranteed to eventually increase over time. This, in turn, guarantees that T_i strictly increases over time, $\forall n_i \in N$. ■

Hence, we are assured of eventual packet injection for each individual node of the network. In other words, the above theorem establishes *fairness* in network access among all the nodes.

Performance Comparisons

An extensive set of simulations was conducted to obtain information concerning the potential gain in performance by switching from the oblivious wormhole to the adaptive cut-through technique. We now summarize very briefly the typical kind of behaviors observed in these simulations. A much more detailed discussion can be found in [5]. Among the

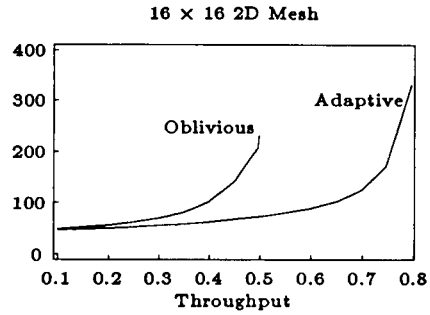


Figure 8: Message latency versus throughput.

various statistics collected, the two most important performance metrics in communication networks are *network throughput* and *message latency*. Figure 7 plots the sustained normalized network throughput versus the normalized applied load of the oblivious and adaptive schemes for a 16×16 2D-mesh network under random traffic. The normalization is performed with respect to the network bisection bandwidth limit. Starting at a very low applied load, the throughput curves of both schemes rise along a unit slope line. The oblivious wormhole curve levels off at $\approx 45 - 50\%$ of normalized throughput but remains *stable* even under increasingly heavy applied load. In contrast, the adaptive cut-through curve keeps rising along the unit slope line until it is out of the range of collected data. It should be pointed out, however, that the increase in throughput obtained is also partly due to the extra silicon area invested in buffer storage, which makes adaptive choices available.

Figure 8 plots the message latency versus normalized throughput for the same 2D-mesh network for a typical message length of 32 flits. The curves shown are typical of latency curves obtained in virtual cut-through switching. Both curves start with latency values close to the ideal at very low throughput, and remain relatively flat until they hit their respective transition points, after which both rise rapidly. The transition points are $\approx 40\%$ and 70%, respectively, for the oblivious and adaptive schemes. In essence, adaptive routing control increases the quantity of routing service, *ie*, network throughput, without sacrificing the quality of the provided service, *ie*, message latency, at the expense of requiring more silicon area.

Summary

Several issues related to adaptive cut-through routing have been addressed in the course of this research, and we did not encounter any insurmountable problem. Rather, the simplicity of these resolution mechanisms gives us hope that the adaptive scheme can be made to improve on the already highly evolved oblivious routing scheme. The discussion in this paper has focused on issues concerning the *feasibility* of the proposed adaptive routing framework. Within this framework, we have also studied and found promising approaches to fault-tolerant routing. Clearly, more work remains to be done. Perhaps the most challenging of all is to realize on *silicon*, the set of ideas outlined in this study.

References

[1] Charles L. Seitz, "The Cosmic Cube," *CACM*, 28(1), January 1985, pp. 22–33.

[2] William C. Athas and Charles L. Seitz., "Multicomputers: Message-Passing Concurrent Computers," *IEEE Computer*, August 1988, pp. 9–24.

[3] William J. Dally and Charles L. Seitz, "The Torus Routing Chip," *Distributed Computing*, 1986(1), pp. 187–196.

[4] Charles M. Flaig, *VLSI Mesh Routing Systems*. Caltech Computer Science Department Technical Report, 5241:TR:87.

[5] John Y. Ngai, *Adaptive Routing in Multicomputer Networks*. Ph.D. Thesis, Computer Science Department, Caltech. To be published.

[6] P. Kermani and L. Kleinrock, "Virtual Cut-Through : A New Computer Communication Switching Technique," *Computer Networks* 3(4) pp. 267–286, Sept. 1979.

[7] P. Merlin, and P. Schweitzer, "Deadlock Avoidance in Store-and-Forward Networks – I : Store-and Forward Deadlock," *IEEE Transactions on Communications*, Vol. COM-28, No. 3, pp. 345–354, March 1980.

[8] William J. Dally and Charles L. Seitz, "Deadlock-Free Message Routing in Multiprocessor Interconnection Networks," *IEEE Transactions on Computers*, Vol. C-36, No. 5, pp. 547–553, May 1987.

[9] A. Borodin, and J. Hopcroft, "Routing, Merging, and Sorting on Parallel Models of Computation," *Journal of Computer and System Sciences*, **30**, pp. 130–145 (1985).

[10] Alain J. Martin, "A Synthesis Method for Self-timed VLSI Circuits," *Proc. 1987 IEEE International Conference on Computer Design: VLSI in Computers & Processors*, IEEE Comp. Soc. Press, pp. 224–229 (1987).

[11] Charles L. Seitz, "System Timing," *Introduction to VLSI Systems*, C. Mead & L. Conway, Addison-Wesley, 1980, Chapter 7.

[12] M. T. Liu, "Distributed Loop Computer Networks," *Advances in Computers*, M. Yovits, Academic Press, pp. 163–221, 1978.