Reactive-Process Programming
and
Distributed Discrete-Event Simulation

Thesis by

Wen-King Su

In Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

California Institute of Technology

Pasadena, California

1990

(Submitted October 31, 1989)

© 1990

Wen-King Su

All rights Reserved

Acknowledgments

Many thanks

To my thesis advisor, Dr. Charles L. Seitz, whose care and dedication made it all possible.

To my committee members, Dr. Charles L. Seitz, Dr. Mani Chandy, Dr. Alain Martin, Dr. Brad Sturtevant, and Dr. Eric Van de Velde, for their careful review and analysis of my research.

To our technical editor, Dian De Sha, who spent glorious days and nights tracking and hunting the blunders and blemishes in my writing.

To our operations manager, Arlene DesJardins, who takes care of every little day-to-day detail and makes the department feel like a nice big family.

To my peers, Bill Athas, Bill Dally, John Ngai, and Craig Steele, for their help and advice.

To my junior co-workers, Nanette Boden, Charles Flaig, Glenn Lewis, Mike Pertel, and Jakov Seizovic, for their feedback and support.

To our system managers, Don Speck, Chris Lee, and Joe Beckenbach, for keeping our machines running smoothly.

To our guests from abroad, Sven Mattisson and Lena Peterson, for their enthusiasm and friendship.

To my advisors at UC Davis, Dr. Wen C. Lin of EE/CS and Dr. George E. Bruening of BioChem, for my enlightenments.

To my buddies from UC Davis, Glenn Saito and John Bakos, for their help in my college years.

To my teachers and counselor at Casa Roble High School, Mr. Gomez, Dr. Smithson, Mr. Hoffman, Mr. Scalatta, Mr. Pickard, Mr. Hellen, Mrs. Sproul, and Mrs. Cruzen, who worked to keep me involved in school.

To Xerox Corporation for supporting this work through a Xerox special-opportunity fellowship.

To my parents, who endured many difficult times to bring me here and to raise me in this land of opportunity.

And to Freedom and Liberty;

sacred to our very heart and soul, yet sadly denied to so many.

The research described in this thesis 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.

Abstract

The same forces that spurred the development of multicomputers — the demand for better performance and economy — are driving the evolution of multicomputers in the direction of more abundant and less expensive computing nodes — the direction of *fine-grain* multicomputers. This evolution in multicomputer architecture derives from advances in integrated circuit, packaging, and message-routing technologies, and carries far-reaching implications in programming and applications. This thesis pursues that trend with a balanced treatment of multicomputer programming and applications. First, a reactive-process programming system — Reactive-C — is investigated; then, a model application — discrete-event simulation — is developed; finally, a number of logic-circuit simulators written in the Reactive-C notation are evaluated.

One difficulty in multicomputer applications is the inefficiency of many distributed algorithms compared to their sequential counterparts. When better formulations are developed, they often scale poorly with increasing numbers of nodes, and their beneficial effects eventually vanish when many nodes are used. However, rules for programming are quite different when nodes are plentiful and cheap: The primary concern is to utilize all of the concurrency available in an application, rather than to utilize all of the computing cycles available in a machine. We have shown in our research that it is possible to extract the maximum concurrency of a simulation subject, even one as difficult as a logic circuit, when one simulation element is assigned to each node. Despite the initial inefficiency of a straightforward algorithm, as the the number of nodes increases, the computation time decreases linearly until there are only a few elements in each node. We conclude by suggesting a technique to further increase the available concurrency when there are many more nodes than simulation elements.

Contents

List of Figures ix

List of Program Listings xiii

1 Introduction 1

1.1 Motivation 1

1.2 History 2

1.3 Outline 5

2 Reactive-Process Programming 7

2.1 Definition of a Reactive Process 7

2.2 Reactive-C Programming System 9

3 Reactive-Process Layers 18

3.1 Simple Layers 18

3.1.1 The bottom layer (`b`-layer) 18

3.1.2 The length-carrying layer (`l`-layer) 22

3.1.3 The non-blocking-receive layer (`nb`-layer) 24

3.1.4 Handler layering 26

3.2 Message Type 28

3.3 Discretion on Receive 30

3.3.1 Discretion using `b`-layer functions 30

3.3.2 The RPC-discretion layer (`r`-layer) 34

3.3.3 The CSP-discretion layer (`csp`-layer) 37

3.3.4 A more general type-discretion layer (`t`-layer) 39

3.4 Other Layers 40

3.4.1 A flow-controlling layer (`f`-layer) 40

3.4.2 The CK primitives 42

3.4.3 The RK primitives (`x`-primitives) 44

3.5 Layering on Light-Weight Processes 45

4 **Cosmic Environment** **47**

4.1 The Cosmic Environment Specification 47

4.2 Our Cosmic Environment Implementation 49

4.2.1 Structure of our CE implementation 50

4.2.2 Cosmic Environment exterior 53

4.2.3 Cosmic Environment processes 54

4.2.4 Program compilation 56

4.2.5 Spawning programs 57

4.2.6 Data representation and conversion 57

5 **Model of Simulation** **61**

5.1 Mathematical Framework and Analysis 61

5.1.1 Systems and elements 61

5.1.2 States and time 63

5.1.3 Knots and progress 64

5.1.4 Rules of thumb — sufficient conditions for progress 66

5.1.5 Non-existence of necessary and sufficient progress conditions . . . 67

5.1.5.1 Simulation and Boolean satisfiability 67

5.1.5.2 Simulation and simultaneous equations 67

5.2 Operational Framework 69

5.2.1 Breaking a simulation into smaller slices 69

5.2.2 Slices and knots 71

5.2.3 Implementation considerations 72

5.3 The Generic Simulator Model and Its Derivatives 73

5.3.1 Message-driven simulation 74

5.3.2 Concurrent event-driven simulation 75

5.3.3 Sequential simulator 76
5.3.4 Concurrent backtracking simulators 78
5.3.5 Branch-and-bound simulators 79
5.3.6 Time-driven simulators 81
5.3.7 Summary 82

6 Logic-Circuit Simulator Experiments **84**

6.1 Why Logic Circuits? 85
6.2 CMB-Variant Simulator 87
6.2.1 The element simulators 87
6.2.2 The simulator message system 94
6.2.3 The variants 97
6.2.4 Variant algorithms 99
6.2.5 Instrumentation 101
6.2.6 Experimental results 103
6.3 Sequential Simulator 107
6.3.1 Sequential simulator mechanism 107
6.3.2 Hazards in sequential simulators 110
6.3.3 Instrumentation 112
6.3.4 Big multiplier results 113
6.3.5 Small multiplier results 115
6.3.6 Circuit topology *vs.* activity level 118
6.3.7 Hybrid possibilities 120

7 Hybrid Simulators **122**

7.1 Coordinated Sequential Simulator (Hybrid-1) 122
7.1.1 The algorithm 122
7.1.2 Sorting with a different key 123

7.1.3 The simulator mechanism . . . 126

7.1.4 The simulator output . . . 128

7.1.5 Expectation . . . 128

7.1.6 Experimental results . . . 129

7.2 Progressive Hybrid Simulator (Hybrid-2) . . . 133

7.2.1 The mechanism . . . 134

7.2.2 Experimental results . . . 137

8 Additional Performance Results **141**

8.1 2-D Clock Network . . . 142

8.1.1 Description . . . 142

8.1.2 Sweep-mode results . . . 144

8.1.3 Real-mode results . . . 145

8.2 Tree-Ring Example . . . 152

8.2.1 Description . . . 152

8.2.2 Simulation results . . . 153

8.3 FIFO Loop . . . 160

8.3.1 Description . . . 160

8.3.2 Simulation results . . . 160

9 Summary **167**

9.1 Economy and Performance of a Multicomputer . . . 167

9.2 Overhead and Latency . . . 171

9.3 Fine-Grain Multicomputer Programming . . . 173

9.4 The Next Frontier . . . 174

10 Bibliography **175**

List of Figures

1.1 Block diagram of a multicomputer 1

2.1 Possible behavior of a reactive process 8

2.2 Representation of a process 9

2.3 Operation of a Reactive-C kernel 10

2.4 Specification of the factorial process 12

2.5 The divide step 13

2.6 The combine step 14

3.1 Mapping a binary tree to a multicomputer 21

3.2 Process structure comparison 21

3.3 Structure of a 1-layer message buffer 22

3.4 An example of a FIFO queue 28

3.5 Expansion steps in the merge-sort program 30

3.6 Giving away a list for the third time (stack grows up) 33

3.7 Getting an out-of-sequence reply 33

3.8 Structure of a channel in a channel-based CSP implementation 38

3.9 Control flow for heavy-weight processes 45

3.10 Control flow for light-weight processes 45

4.1 Elements of a computation 47

4.2 A process group 50

4.3 Partitioning into two parts 50

4.4 A multicomputer shared by two users 51

4.5 Host message-system implementation 52

4.6 Cosmic Environment with unified resource management 52

5.1 Representation of a system 61

5.2 Representation of a system composed of elements 61

5.3 Closing a system into a closed graph 62

5.4 Arc source and destination 62

5.5 Element inputs and outputs 62

5.6 Arcs a_{0-4} form a path of length 5 63

5.7 Arcs a_{0-4} form a circuit of length 5 63

5.8 Example of a knot-containing system 64

5.9 A circuit to evaluate satisfiability of a set of clauses 67

5.10 Mapping equations into physical system 68

5.11 Element-simulator operation for an element with a non-zero delay 70

5.12 Element-simulator operation for an element with a zero delay 70

5.13 A system that contains all three types of slices 71

5.14 Representation of an arc 73

5.15 Replacing tape by messages 74

5.16 Example of deadlock in an event-driven simulation 76

5.17 Model of a sequential simulator 77

5.18 A researcher submitting a grant 79

5.19 Comparison between three simulators 80

5.20 An example of a continuous system 82

6.1 A logic circuit whose behavior is different from its Boolean network 84

6.2 A number of circuit simulators and their relationship 85

6.3 Domain of the generic simulator model 87

6.4 Process structure and a simple example of connectivity 88

6.5 A sample circuit and a possible mapping to a multicomputer 95

6.6 Structure of a sweep-mode simulation 102

6.7 Structure of a real-mode simulation 102

6.8 Three phases of the oscillating multiplier 104

6.9 A 1376-gate multiplier, sweep-mode 104

6.10 A circuit containing a dynamic hazard condition 110

6.11 A 1376-gate multiplier for 40μs on an iPSC/2 113
6.12 A 1376-gate multiplier for 40μs on an iPSC/1 114
6.13 Combining the iPSC/2 and iPSC/1 graphs with sequential timing aligned . . 115
6.14 A 1376-gate multiplier for 100μs on a Symult 2010 116
6.15 A 116-gate multiplier for 100μs on an iPSC/1 117
6.16 A 116-gate multiplier for 100μs on an iPSC/2 117
6.17 A 116-gate multiplier for 400μs on a Symult 2010 117
6.18 Effect of increased latency on simulation performance 118
6.19 A 1376-gate multiplier for 100μs on a Symult 2010 — fast oscillation 119
6.20 Modified Laffer Curve 120
7.1 An event that invalidates another event 125
7.2 Layering in the hybrid-1 simulator 126
7.3 Expected performance of the hybrid-1 simulator 129
7.4 A 1376-gate multiplier for 100μs on a Symult 2010 130
7.5 A 1376-gate multiplier for 100μs on a Symult 2010 with random placement . . 131
7.6 A faster oscillating 1376-gate multiplier for 100μs on a Symult 2010 132
7.7 A 1376-gate multiplier for 100μs on a Symult 2010 137
7.8 A 1376-gate multiplier for 100μs on a Symult 2010 with random placement . . 138
7.9 A faster-oscillating 1376-gate multiplier for 100μs on a Symult 2010 139
7.10 A 116-gate multiplier for 400μs on a Symult 2010 140
8.1 A FIFO consisting of 4 units 142
8.2 A C-element FIFO consisting of 4 units 143
8.3 A 3×4 array of self-oscillating FIFO units 143
8.4 Sweep-mode CMB-variant simulation of an 1841-gate clock network 144
8.5 An 1841-gate clock network for 50μs on a Symult 2010 146
8.6 An 1841-gate clock network for 50μs on a Symult 2010 148
8.7 A 473-gate clock network for 200μs on a Symult 2010 149

8.8 A 241-gate clock network for 200μs on a Symult 2010 150

8.9 A 65-gate clock network for 500μs on a Symult 2010 151

8.10 A 12-unit tree ring 152

8.11 A 1-to-2 pulse-distributor circuit 153

8.12 A 1142-gate tree network for 50μs on a Symult 2010 155

8.13 A 1142-gate tree network for 50μs on a Symult 2010 156

8.14 An 857-gate tree network for 70μs on a Symult 2010 157

8.15 An 572-gate tree network for 100μs on a Symult 2010 158

8.16 An 287-gate tree network for 200μs on a Symult 2010 159

8.17 Circuit for one latch 160

8.18 Sweep-mode CMB-variant simulation of an 1067-gate FIFO loop 161

8.19 An 1067-gate FIFO loop for 100μs on a Symult 2010 163

8.20 An 1067-gate FIFO loop for 100μs on a Symult 2010 164

8.21 A 459-gate FIFO loop for 100μs on a Symult 2010 165

8.22 A 155-gate FIFO loop for 200μs on a Symult 2010 166

9.1 Two idealized multicomputer evolution paths 167

9.2 Multicomputer cost space 169

9.3 Intersection with A plane 169

9.4 Intersection with B-plane 170

9.5 Two idealized multicomputer evolution paths in the path space 171

List of Program Listings

2.1 Kernel of Reactive-C programming environment . . . 10

2.2 Reactive-C factorial program . . . 11

2.3 Factorial main program . . . 16

3.1 Heavy-weight factorial program . . . 19

3.2 Program fragments for mapping a binary tree to a multicomputer . . . 20

3.3 The carrier program for building FIFO . . . 29

3.4 The merge-sort program . . . 32

3.5 An incorrect implementation of the C read function . . . 34

3.6 A correct implementation of the C read function . . . 37

4.1 Three representations of π in double-precision floating-point-number format . 58

4.2 Three layouts of a structure, in order of increasing byte address . . . 59

6.1 Structure of a FRAGMENT . . . 89

6.2 An inverter in a CMB-variant simulator . . . 89

6.3 An XOR-gate in a CMB-variant simulator . . . 90

6.4 An OR-gate in a CMB-variant simulator . . . 92

6.5 CMB-variant QUEUE_FRAGMENT function . . . 93

6.6 CMB-variant TRIM_FRAGMENT function . . . 94

6.7 CMB-variant OUTPUT function . . . 96

6.8 CMB-variant main loop . . . 96

6.9 CMB-variant indefinitely-lazy main loop . . . 98

6.10 CMB-variant demand-driven main loop . . . 100

6.11 CMB-variant main loop as a light-weight process . . . 101

6.12 Sequential-simulator event structure . . . 107

6.13 An inverter in sequential simulator . . . 108

6.14 The SEND_EVENT function in sequential simulator . . . 108

6.15 An OR-gate in sequential simulator . . . 109

6.16 Sequential-simulator main loop as a light-weight process 110

6.17 A SEND_EVENT function that reduces glitches 112

7.1 Hybrid-1 main loop 127

7.2 Hybrid-1 embedded message system 127

7.3 Generic logic-gate handler for hybrid-2 134

7.4 Hybrid-2 main loop 136

Chapter 1 Introduction

Section 1.1 Motivation

Advances in applications, programming methods, and computer architectures are inextricably intertwined. Architectures and programming methods develop in response to demands from applications; they also give rise to new applications. Simulation is an application that contributes to and benefits from the development of faster and more economical computers. *Discrete-event simulation* can produce a broad variety of interaction patterns and timing relationships; it is, therefore, a *model application* for the study of *multicomputers* and *reactive-process programming*. This research is a study of both reactive-process programming and distributed discrete-event simulation on multicomputers.

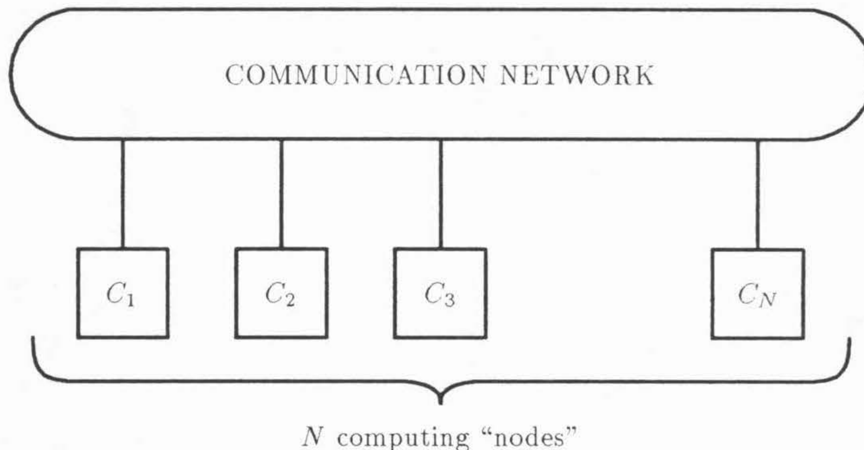

Figure 1.1 Block diagram of a multicomputer.

A multicomputer (Figure 1.1) is composed of a collection of *node* computers connected to each other via a message-passing network. Multicomputers can be divided into three categories by their node size:

Category	Node Size	Memory per Node	N	Examples
Coarse-grain	cabinet	$\approx$ 64MB	4–64	Network of supercomputers
Medium-grain	circuit-board	$\approx$ 2MB	16–4096	iPSC, NCUBE, Symult 2010
Fine-grain	chip	$\approx$ 16KB	1024–65536	Mosaic

Each node has its own private memory that is not directly accessible by other nodes, and each node can contain multiple *processes*. Processes on different nodes run asynchronously;

processes within a single node are interleaved to produce the same effect as if they were in different nodes. Communication between processes is performed via message passing.

Section 1.2 History

Simulation and programming have long influenced each other. Although one can argue that *every* computation is, in fact, a simulation of some physical or abstract process, the first effort to provide a programming system for discrete-event simulation was the development of *Simula* [6], which was based on the *Algol* programming language. Discrete-event simulation operates on a system of components (*physical processes*) that interact by discrete actions. Structured languages such as Algol permit the modular representation of these components. As such languages became available, discrete-event simulation techniques began to emerge from the traditional event-list-oriented simulation techniques. Each Simula module contains its own set of private data and procedures, and is, in effect, a process that interacts with others to perform a simulation.

Although it was initially conceived as a simulation language, Simula became a general-purpose, object-oriented, multiple-process programming language. The assimilation of object-oriented and multiple-process programming concepts led to the development of *CSP* [8], *Smalltalk* [7], and other systems that are more closely identified with programming. Although Smalltalk was created to make programming simple, its programming model also gave it the potential for concurrent operation of its objects. CSP was created to study and unify diverse distributed programming constructs by using concurrent processes and synchronous messages. Smalltalk and Simula are both object-oriented systems; CSP includes the concept of independent, interacting processes without the distraction of such object-oriented concepts as *inheritance*.

Multicomputer implementations for variants of Simula [9] and Smalltalk [11] were shown to be feasible and useful. *Occam* [10], a CSP variant with static interprocess communication graphs, provided a programming system for *transputer*-based multicomputers. However,

most commercial multicomputers do not use language derivatives as their basic programming system because the concepts of multiple-process programming also appear in operating systems. Interprocess synchronization and communication capabilities became common in such popular operating systems as UNIX. Although UNIX began as a system with simple file locks and data streams, it evolved into one in which both servers and clients abound, and whose processes are capable of complex interaction with other processes either on the same machine or on other machines via computer networks. Thus, when *medium-grain multicomputers* with PC-sized nodes became available, the conventional process model of multiprogramming operating systems was used.

These machines use generic, sequential programming languages, such as *C*, *Fortran*, *Lisp*, and *Pascal*. Codes written in these languages compile into independent programs that are run in the nodes as processes. These processes interact with each other by calling library functions that send and receive messages. The model of a conventional operating system is chosen because the sequential programming languages are adequate for most applications, and also because object-oriented languages and others, such as *Lisp* and *Prolog*, can be implemented easily on such systems. Program objects are represented by processes and embedded processes.

Early experiments in distributed discrete-event simulation were done by Mani Chandy and Jay Misra [13], and independently by Randy Bryant [12]. These approaches were seen as variants of event-list-based sequential simulation algorithms, in which synchronization is accomplished by message-passing. Although the degree of synchronization that exists in most sequential simulators can be relaxed when a simulation is distributed, extra work (or *overhead*) is required to maintain the necessary synchronization. Such simulators are called *conservative simulators*, because the processes do not perform speculative computations.

The speculative (*optimistic*) approach was developed by David Jefferson [14] to improve the performance of simulations for the medium-grain multicomputers. His research on the *Time Warp* simulator resulted in a general-purpose programming system called the *Virtual*

Time System. The idea was to save the state of a process whenever the process encounters a synchronization point; then, instead of blocking the process until the synchronization is complete, to have the process select a possible outcome and continue to execute. When the synchronization is finally complete, if the outcome differs from the selected outcome, the process and all those that it has since affected are *rolled back*, and process execution restarts at the synchronization point.

Methods for reducing overhead were studied intensively because nodes in a medium-grain multicomputer are few and expensive. However, as multicomputers evolve toward their next incarnation — the fine-grain multicomputers — nodes become abundant and cheap. With a myriad of single-chip nodes, fine-grain multicomputers promise significantly better cost-*vs.*-performance ratios and total computing capacity than do the medium-grain multicomputers. The *Mosaic C*, currently being developed at Caltech, is an example of a fine-grain multicomputer. While each node of the Mosaic C contains a 16-bit CPU, a message router, and only 16 Kbytes of RAM, the entire Mosaic C will contain 16K nodes.

A number of fine-grain, reactive-process-based programming languages have been developed in anticipation of the fine-grain multicomputer. Among them is the *Cantor* notation, which most strongly influenced the programming methods used in this research. (Cantor is being developed by W.C. Athas [15] using a model similar to the *Actor* notation [1].) Reactive-process programming systems are similar to CSP, but impose additional constraints on the operation of the processes in order to simplify the operating systems of the fine-grain multicomputers. Cantor also allows us to express programs in finely divided objects that are distributed over many small nodes.

The inversion of the cost ratio between the processor and the memory forms a new set of ground rules for multicomputer programming. The shifting focus has strong implications for programming in general: The memory, rather than the processor, is now the scarce commodity. Programming techniques that buy speed by using a large number of idle memory cells are no longer favorable, but ones that buy speed by using idle processors are. Instead

of trying to have something useful happen in every available CPU cycle in the machine, application writers should now focus on extracting as much concurrency as possible from the application.

In this experiment, the concept of fine-grain, reactive-process programming influenced simulation. The overhead that prompted the development of optimistic approaches for medium-grain multicomputers was recast in a more benign role. Having this overhead merely required the use of a larger number of inexpensive processors in the multicomputer, and did not reduce the amount of concurrency that could be extracted from the system being simulated. A programming system similar to Cantor was developed for this research, and a number of conservative simulators suitable for fine-grain multicomputers were developed.

Section 1.3 Outline

Since this research is a study of both programming and simulation, this thesis is divided into two major parts: Chapters 2 through 4 deal with programming, and Chapters 5 through 8 deal with simulation. The two parts are only loosely interdependent, and do not reflect the extensive two-way influence that exists between simulation and programming. For example, the *lazy-evaluation model* of simulation guided us in the design of the x-*primitives*, which are the message-handling functions of our reactive-process programming system; and the support mechanisms in the simulator were modeled after the mechanisms of the *Reactive*-C programming system.

Chapter 2 introduces reactive-process programming and the Reactive-C implementation of its basic mechanisms. Reactive-C is merely the ordinary C programming language used with a particular programming discipline. It is useful for exposing the simplicity of reactive-process programming systems — a level of simplicity that is necessary for any programming system for fine-grain multicomputers. It is not the best tool, however, for studying reactive-process programming. Therefore, a slightly higher-level programming system is used in Chapter 3 to demonstrate the generality and simplicity of reactive-process programming.

Chapter 4 describes the *Cosmic Environment*, a programming environment that embodies the reactive-process programming discipline.

The discussion of simulation begins in Chapter 5 with the model of simulation. The subject system being simulated is recursively defined to be a collection of interacting systems or elements, and elements are simulated by a set of simulators that interact by message-passing. The condition for progress is discussed in detail, a generic simulator is described, and the derivation of a variety of simulators is shown. Chapter 6 describes a direct implementation of the generic simulator using the Reactive-C notation. Logic circuits are the subject of choice, because they are diverse and because they expose properties of the simulators by imposing few processing requirements of their own. The performance we observed is shown to be that which was expected: The time required for a simulation decreases linearly as the number of computing nodes increases. Comparing the performance to the sequential simulator shows that the overhead does not interfere with the ability to utilize the concurrency available in the system. Chapter 7 introduces new simulators that do not have an overhead when only one node is used. However, the speed increase is no longer linear: Performance converges to that of the previous simulator as more nodes are used. Although only one test circuit was used throughout these two chapters, additional results on a few other circuits are presented in Chapter 8. The results are all similar, even though the circuits being simulated are quite different.

Finally, Chapter 9 defends the rationale for simulation on fine-grain multicomputers, and discusses some of its implications on programming and simulation.

Chapter 2 Reactive-Process Programming

Reactive-process programming is a discipline in which processes are inactive until they are triggered by inputs. When suitable inputs are present, a process and its inputs will *react* in a single atomic action in which the inputs are consumed. Reactive-process programs can be written in specifically designed notations such as Cantor; they can also be written in vanilla notations such as C. Although Cantor hides many rough edges to make programming simpler, C is perhaps better in exposing the mechanics of reactive-process programming. We will use C for our discussion. and assume that readers are familiar with C.

A reactive-process program can be written as a simple combination of data structure and function, as a full-fledged heavy-weight process with its own process context, or as a complex multi-tasking operating system. The diversity arises from a small and elegant set of properties that allows reactive-process programming systems with very different capabilities to be built on top of one another in a consistent manner. Since the tailoring of a programming system to specific requirements is made simple, an application no longer has to be twisted around the system; instead, the system can be crafted to suit the intrinsic needs of the application.

In this chapter, we will describe reactive-process programming in its simplest form; the next chapter will be devoted to examples of building more-complex programming systems on top of simpler ones.

Section 2.1 Definition of a Reactive Process

A *reactive process* can be characterized by its two run-states:

Waiting: While a process is waiting, it is completely inert. The process will remain in the waiting state as long as there is no message ready for it to receive; otherwise, the process will be run, taking the earliest-arriving message as its input.

Running: While a process is running, it cannot receive any more messages. A process can run for only a finite period of time before it returns to the waiting state. While a process is running, it can:

a. modify its internal state,

b. send messages,

c. instantiate other processes, or

d. self-destruct.

Message buffers remain attached to a process until they are explicitly released by the process.

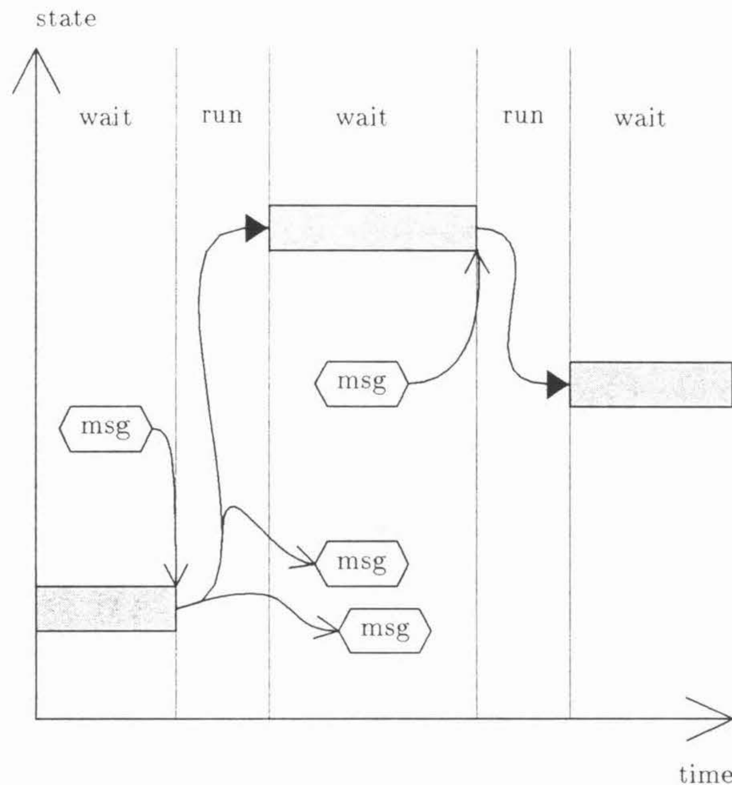

Figure 2.1 Possible behavior of a reactive process.

The reactive-process programming environment has these additional properties:

1. Processes do not exist until they are instantiated.

2. Processes persist until they self-destruct.

3. Each process has a unique process ID.
4. Messages are addressed by the destination-process ID.
5. Message order between any pair of processes is preserved.
6. Messages not immediately consumed are queued.
7. Messages with a valid destination will eventually be delivered.
8. Message buffers are allocated by calling an allocate function.
9. Message buffers can be released by calling either a deallocate or a send function.

Section 2.2 Reactive-C Programming System

Reactive-C is a minimalist implementation of a reactive-process programming environment using the C programming language. As shown in Figure 2.2, a process in Reactive-C is represented by a process structure that includes two pointers: a function pointer and a data pointer. The function pointer references a C function, the current *entry function* of the process. The entry function is called when a process is run.

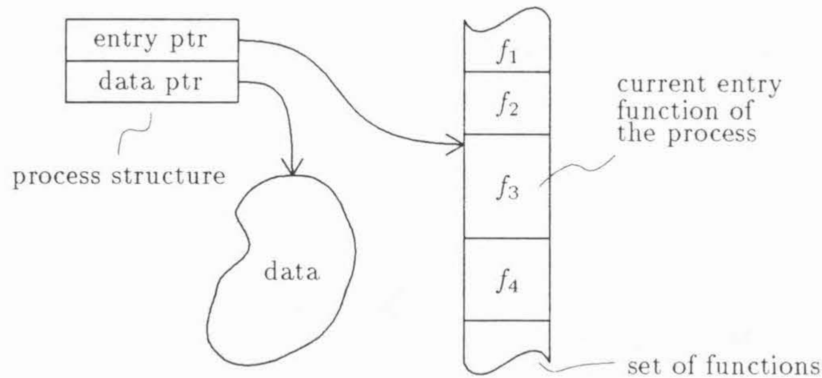

Figure 2.2 Representation of a process.

The data pointer references an arbitrary data structure maintained by the process. Both the data structure and the two pointers are state variables of the process that owns them, and the process can modify them at any time while it is running. When a process starts to run, the triggering message and the process structure are passed to the entry function as function arguments. A process returns to the waiting state by returning from the entry function.

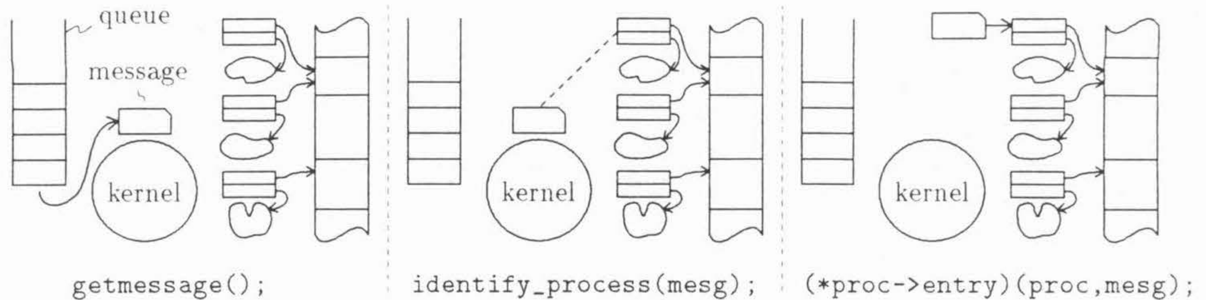

Figure 2.3 Operation of a Reactive-C kernel.

Listing 2.1 is a sample kernel loop of the Reactive-C programming environment. As shown in Figure 2.3, the kernel repeatedly gets a message from the message queue, identifies the receiver, and calls the entry function of the receiving process.

```
1       kernel_loop()
2       {
3           char *mesg;
4           PROC *proc;

6           while(1)
7           {
8               mesg = getmessage();
9               proc = identify_process(mesg);
10              (*proc->entry)(proc,mesg);
11          }
12      }
```

Listing 2.1 Kernel of Reactive-C programming environment.

Listing 2.2 contains an example of a reactive-process program that computes a factorial in logarithmic time on an arbitrarily large machine.

```
1       typedef struct { REF ID; int HI, LO; } FAC_DATA;

3       fac_1(proc,mesg)
4           RC_PROC *proc; FAC_DATA *mesg;
5       {
6           FAC_DATA *mesg2;
7           int         half;
8
9           if(mesg->HI <= mesg->LO)
10          {
11              rc_send(mesg->ID,mesg);
12              rc_exit();
13
14          } else
15          {
16              half  = (mesg->HI + mesg->LO)/2;
17
18              mesg2 = (FAC_DATA *) rc_malloc(sizeof(FAC_DATA));
19              mesg2->ID = rc_myid();
```

```
20              mesg2->HI =  mesg->HI;
21              mesg2->LO =     half+1;
22              rc_spawn(fac_1,mesg2);
23
24              mesg2 = (FAC_DATA *) rc_malloc(sizeof(FAC_DATA));
25              mesg2->ID = rc_myid();
26              mesg2->HI =       half;
27              mesg2->LO =  mesg->LO;
28              rc_spawn(fac_1,mesg2);
29
30              proc->data  = (char *) mesg;
31              proc->entry =         fac_2;
32          }
33      }

35      fac_2(proc,mesg)
36          RC_PROC *proc; FAC_DATA *mesg;
37      {
38          ((FAC_DATA *)(proc->data))->LO = mesg->LO;
39          rc_free(mesg);
40          proc->entry = fac_3;
41      }

43      fac_3(proc,mesg)
44          RC_PROC *proc; FAC_DATA *mesg;
45      {
46          ((FAC_DATA *)(proc->data))->LO *= mesg->LO;
47          rc_free(mesg);
48          rc_send(((FAC_DATA *)(proc->data))->ID, proc->data);
49          rc_exit();
50      }
```

Listing 2.2 Reactive-C factorial program.

The three functions in Listing 2.2 (`fac_1`, `fac_2`, and `fac_3`) are in a suitable form for entry functions because their arguments are the process structure and the input message, and because they are assured to return in finite time. However, they do not represent actual processes; they are merely message-handling functions for processes that reference them by their entry pointers.

Let a *factorial process* be a process that references any of the three functions. Initially, a factorial process waits for a message whose structure is defined by the `C` data structure called `FAC_DATA`. The message is called a `FAC_DATA` message.

```
1       typedef struct { REF ID; int LO, HI; } FAC_DATA;
```

`ID`: Data structure containing the caller's process ID.
`LO`: Low end of a number range.
`HI`: High end of a number range.

After receiving the message (Figure 2.4), the factorial process computes the product of all integers within the closed interval: `[LO,HI]`. The factorial process stores the product

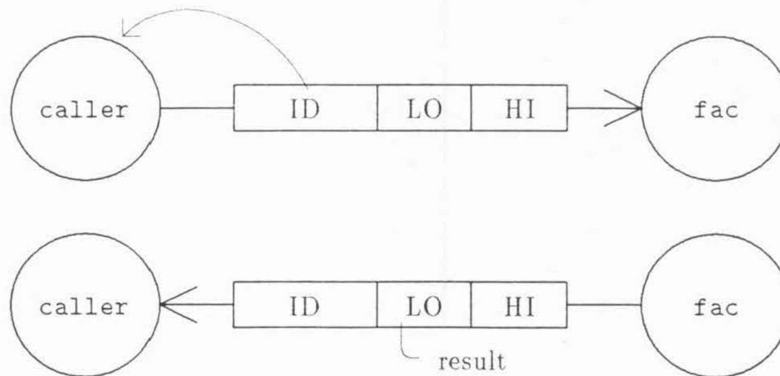

Figure 2.4 Specification of the factorial process.

in the LO field of another FAC_DATA message, which is returned to the requester. Thus, sending a FAC_DATA message with a 1 in the LO field to the factorial process will cause the the factorial of HI to be computed.

To compute the factorial of a value, the requesting process (caller) instantiates a new process whose entry pointer contains the address of the fac_1 function. We shall call this new process the fac_1 process. The factorial is computed by a divide-and-conquer method that iterates using the difference between HI and LO.

```
9              if(mesg->HI <= mesg->LO)
```

When the fac_1 process receives its first message, it compares the two ends of the interval described in the message. If HI equals LO, then there is only one integer in the interval. If HI is 0 (therefore less than LO, which must be 1 at this point), then the factorial of 0 is to be computed. In either case, the correct reply value is equal to the number already contained in LO.

```
11             rc_send(mesg->ID,mesg);
12             rc_exit();
```

Therefore, when LO $\geq$ HI, the message is bounced back to the caller, untouched. The rc_send function called in line 11 causes the message buffer mesg to be sent to the process whose ID is mesg->ID, which is, in this case, the ID of the caller. Since rc_send dissociates the message buffer from the process, the process does not have to release it explicitly before the process is terminated by calling the rc_exit function.

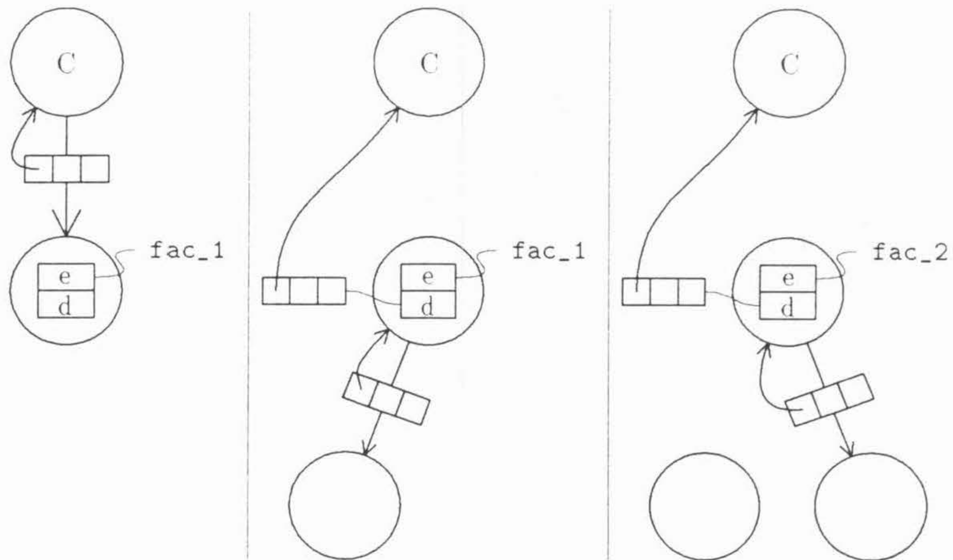

Figure 2.5 The divide step.

```
16              half  = (mesg->HI + mesg->LO)/2;
```

If HI is greater than LO, the fac_1 process computes a midpoint that divides the interval into two smaller intervals. Two more fac_1 processes are created to work on these two intervals (Figure 2.5). These processes are called the siblings of this process, and an initialization message is sent to each sibling as it is created.

```
18              mesg2 = (FAC_DATA *) rc_malloc(sizeof(FAC_DATA));
```

Message buffers are allocated by the rc_malloc call. The function rc_malloc has the same semantics as the malloc function in C. Depending on the implementation, rc_malloc can be identical to C malloc, can be built on top of C malloc, or can be an entirely different allocator that gets space from a dedicated memory region.

```
19              mesg2->ID = rc_myid();
20              mesg2->HI =  mesg->HI;
21              mesg2->LO =     half+1;
```

After a message buffer has been allocated, it is filled with data to be sent to a sibling. Lines 19–21 are for the sibling that handles the upper half of the interval. The rc_myid function returns the ID of the process. The process becomes the caller of its siblings after its ID has been stored and sent in the ID fields of the initialization messages. The fac_1 process will receive one reply from each of its siblings. When two replies are received, the

process multiplies the values contained in their LO fields and returns the product to its own caller.

```
22              rc_spawn(fac_1,mesg2);
```

Processes are created with the `rc_spawn` function call. At line 22, a new process structure is created, the entry pointer of the new process is initialized to reference the function `fac_1` (first parameter to the `rc_spawn` function), and the message `mesg2` (second parameter to the `rc_spawn`) is sent to the new process as its first input message.

```
30              proc->data  = (char *) mesg;
31              proc->entry =          fac_2;
```

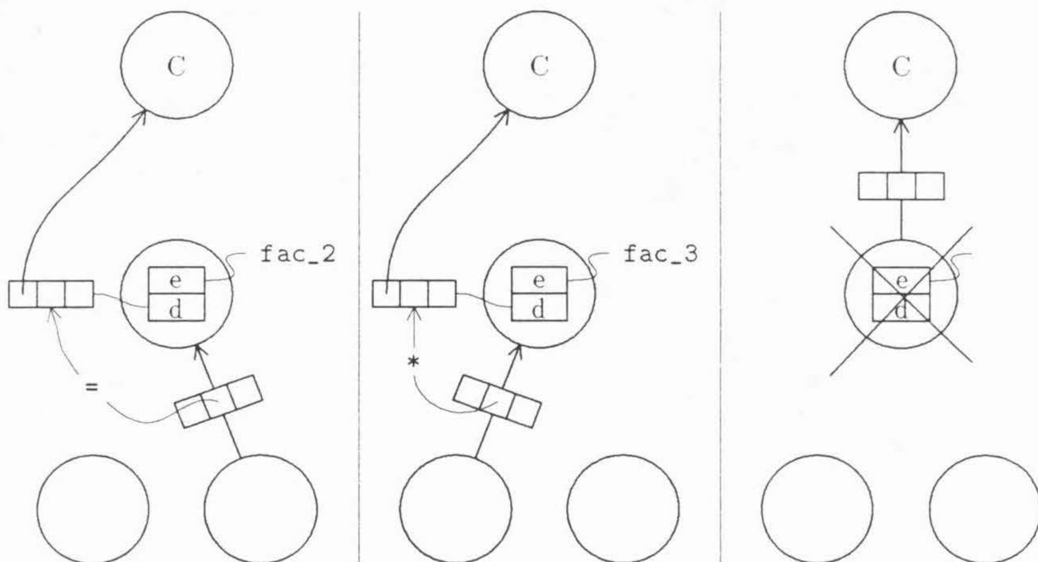

Figure 2.6 The combine step.

The process must now return from the `fac_1` function in order to wait for the replies from its siblings (Figure 2.6). The process sends its reply using the same message buffer that it received, but to prevent losing the reference to that message buffer, it assigns the message buffer into the data pointer of its process structure. Furthermore, since the process is now waiting for a reply message instead of a factorial request message, the entry pointer is changed to reference the function that handles the first reply message. By storing the address of the `fac_2` function into the `entry` field, the `fac_1` process *becomes* a `fac_2` process. The process then returns from the `fac_1` function to indicate that it is going back to the waiting state.

```
35      fac_2(proc,mesg)
36          RC_PROC *proc; FAC_DATA *mesg;
37      {
38          ((FAC_DATA *)(proc->data))->LO = mesg->LO;
39          rc_free(mesg);
40          proc->entry = fac_3;
41      }
```

The fac_2 process waits for the first reply message. When it arrives, its reply value is simply copied into the LO field of the original message buffer, since the process needs a value from each reply before the product can be computed. The reply message buffer from the sibling is no longer needed and is released by calling rc_free. The process then becomes a fac_3 process.

```
46          ((FAC_DATA *)(proc->data))->LO *= mesg->LO;
47          rc_free(mesg);
48          rc_send(((FAC_DATA *)(proc->data))->ID, proc->data);
49          rc_exit();
```

When the fac_3 process gets the second reply message, the returned value is multiplied into the LO field of the original message buffer. The reply message buffer is also freed. The original message buffer, now containing the product of the two reply values, is sent back to the caller. Lastly, the process terminates by calling rc_exit.

Listing 2.3 is a sample program that calls the factorial program. It waits for an input number, computes the factorial of the input number, prints the factorial, and then terminates.

```
1       rc_main(proc,mesg)
2           RC_PROC *proc;
3           char    *mesg;
4       {
5           int          hi;
6           FAC_DATA *mesg2;

8           rc_free(mesg);

10          printf("Enter number: "); scanf("%d",&hi);

12          mesg2 = (FAC_DATA *) rc_malloc(sizeof(FAC_DATA));
13          mesg2->ID = rc_myid();
14          mesg2->HI =         hi;
15          mesg2->LO =          1;
16          rc_spawn(fac_1,mesg2);

18          proc->entry = main_reply;
19      }
```

```
21      main_reply(proc,mesg)
22          RC_PROC *proc; FAC_DATA *mesg;
23      {
24          printf("%d\n",mesg->LO); rc_free(mesg); rc_exit();
25      }
```

Listing 2.3 Factorial main program.

The basic Reactive-C primitives are summarized below:

`char *rc_malloc();`	Allocates a message buffer.
`rc_free();`	Releases a message buffer.
`rc_send();`	Sends and releases a message buffer.
`REF rc_myid();`	Returns the ID of the calling process.
`rc_spawn();`	Instantiates a new process.
`rc_exit();`	Terminates the calling process.

Deliberately omitted from the list is a function that receives a message. In Reactive-C, a message is implicitly requested when a process is created or when a process returns from its entry function. The request is fulfilled when its current entry function is called. The other unusual aspect of the Reactive-C primitives is that `rc_spawn` does not return the ID of the new process; thus, the only direct way for a parent process to get the ID of the sibling is to receive the ID from a message sent by the sibling.

Reactive-C is a minimalist reactive-process programming system. (The kernel code for a single-processor system is only 124 lines long.) Since the parent process can always send its ID to the sibling during spawn, and since the sibling can always send its ID back to its parent via a message, it is not necessary for the spawn function in a minimalist system to return an ID. The goal of Reactive-C is to create a system that is minimal but that is not necessarily easy on the programmer. However, a close relative of the Reactive-C turns out to be well suited for writing event-driven simulators. Another derivative, the *Reactive Kernel*, proves to be very useful in implementing the inner kernel and the handlers of multicomputer operating systems. Details of the Reactive Kernel can be found in the Master's thesis of Jakov Seizovic [5].

Reactive-C is strongly influenced by the Cantor programming language, which is a fine-grain reactive-process programming system in which process spawning uses *futures* to immediately return the sibling ID. The properties and programming paradigms related to fine-grain reactive-process programming are explored in detail the Doctoral thesis of W.C. Athas [15].

In the next chapter, we will focus on the *universality* of reactive-process programming, a property that is best illustrated using full-fledged, coarse-grain reactive processes. Although we will be leaving the Reactive-C environment for now, we should bear in mind that duality exists between a Reactive-C process and its *heavy-weight* counterpart: What is applicable for one is equally applicable for the other. Heavy-weight programs are used for the remainder our discussion because they are simpler to describe.

Universality of a programming system requires the programming system to efficiently support a large variety of other programming systems. *Layering*, or the implementation of new functions on top of basic functions, is the principal means by which universality is achieved.

Chapter 3 Reactive-Process Layers

In contrast to a light-weight Reactive-C process, which has only a function and a data structure, we can generally consider a heavy-weight process to be one that, although its structure is machine dependent, has its own code, data, stack, and thread of control. We can run heavy-weight reactive processes under the Reactive-C programming environment with minimal overhead by using a dedicated, light-weight reactive process, called a *handler*.

In one possible arrangement, the data pointer of a handler references a table containing three segment pointers (for the code, data, and stack segments) and a context structure (containing the frozen records of a suspended heavy-weight process). When a message is received by a handler, the entry function for the handler performs a context switch to resume the execution of the heavy-weight process. When the heavy-weight process calls a receive function, it saves the process context, restores the system context, and returns to the handler. The handler returns from its entry function to request a new message.

In this manner, the combination of the heavy-weight process and its handler appears to the kernel as an ordinary Reactive-C process. The cost of supporting a heavy-weight process under a handler, as opposed to supporting it under the kernel, is no more than one extra level of function call. A handler for a heavy-weight process is an example of layering. A handler that supports multiple heavy-weight processes is used in the Reactive Kernel node operating system for running normal user processes.

Section 3.1 Simple Layers

3.1.1 The bottom layer (b-layer)

As we did for Reactive-C, we shall establish the groundwork for the discussion of universality and layering with an example. Listing 3.1 contains a heavy-weight reactive-process program that computes a factorial in the same manner as the Reactive-C example. We shall refer to the programming system used in this example as the *bottom*, or b-*layer*.

```
1  typedef struct { int pn, pp; int HI, LO; } FAC_DATA;

3  main()
```

```
 4  {
 5      FAC_DATA *data;

 7      {   FAC_DATA *mesg  = (FAC_DATA *) b_recvb();
 8          FAC_DATA *mesg2;
 9          int      half, k;

11          if(mesg->HI <= mesg->LO)
12          {
13              b_send(mesg,mesg->pn,mesg->pp);
14              exit(0);

16          } else
17          {
18              half  = (mesg->HI + mesg->LO)/2;
19              k     = mypid()*nnodes() + mynode();

21              mesg2 = (FAC_DATA *) b_malloc(sizeof(FAC_DATA));
22              mesg2->pn =  mynode();
23              mesg2->pp =   mypid();
24              mesg2->HI =  mesg->HI;
25              mesg2->LO =     half+1;
26              spawn("pfac", (2*k+2)%nnodes(), (2*k+2)/nnodes(), "");
27              b_send(mesg2, (2*k+2)%nnodes(), (2*k+2)/nnodes()    );

29              mesg2 = (FAC_DATA *) b_malloc(sizeof(FAC_DATA));
30              mesg2->pn =  mynode();
31              mesg2->pp =   mypid();
32              mesg2->HI =      half;
33              mesg2->LO =  mesg->LO;
34              spawn("pfac", (2*k+1)%nnodes(), (2*k+1)/nnodes(), "");
35              b_send(mesg2, (2*k+1)%nnodes(), (2*k+1)/nnodes()    );

37              data  =  mesg;
38          }
39      }

41      {   FAC_DATA *mesg  = (FAC_DATA *) b_recvb();

43          data->LO = mesg->LO;
44          b_free(mesg);
45      }

47      {   FAC_DATA *mesg  = (FAC_DATA *) b_recvb();

49          data->LO *= mesg->LO;
50          b_free(mesg);
51          b_send(data,data->pn,data->pp);
52          exit(0);
53      }
54  }
```

Listing 3.1 Heavy-weight factorial program.

A comparison between the Reactive-C example and the b-layer example reveals numerous similarities. The three entry-function candidates are replaced by three program blocks; each block is headed by a line that waits for and receives a message:

```
7       {   FAC_DATA *mesg   = (FAC_DATA *) b_recvb();
```

Instead of messages being passed to it as function arguments, a b-layer process must perform an explicit b_recvb call to get a message. The b_recvb call suspends the process until a message arrives. The message is then returned to the process by the b_recvb function.

```
1  typedef struct { int pn, pp; int HI, LO; } FAC_DATA;
```

A b-layer process is identified by its node and pid pair rather than by just a REF value. There is no reason why it should not use the same single-value representation that Reactive-C uses, except that heavy-weight processes require better control over process placement because they take up a great deal of memory. Thus, wherever ID was used, it is replaced with the node and pid pair.

```
19              k      = mypid()*nnodes() + mynode();

26              spawn("pfac", (2*k+2)%nnodes(), (2*k+2)/nnodes(), "");
27              b_send(mesg2, (2*k+2)%nnodes(), (2*k+2)/nnodes()    );

34              spawn("pfac", (2*k+1)%nnodes(), (2*k+1)/nnodes(), "");
35              b_send(mesg2, (2*k+1)%nnodes(), (2*k+1)/nnodes()    );
```

Listing 3.2 Program fragments for mapping a binary tree to a multicomputer.

Both b_send and spawn need node and pid as their arguments. In order to give a process better control over the placement of its siblings, a process is allowed to define the node and pid of the new processes it creates. The three program fragments shown in Listing 3.2 map a tree structure onto a multicomputer such that if the tree is balanced, the number of processes in any two nodes will differ by no more than 1.

As shown in Figure 3.1, the tree is first mapped to a linear array such that a process with an ID of (node,pid) on a multicomputer with N nodes will have an index of k = pid*N + node. The two siblings of the process will have an index of 2k+1 and 2k+2, respectively. The list is than folded into the multicomputer using the "%" and the "/" operators.

The functions mypid, mynode, and nnodes return the pid of the process, the node of the process, and the number of nodes in the machine. The spawn function creates a process

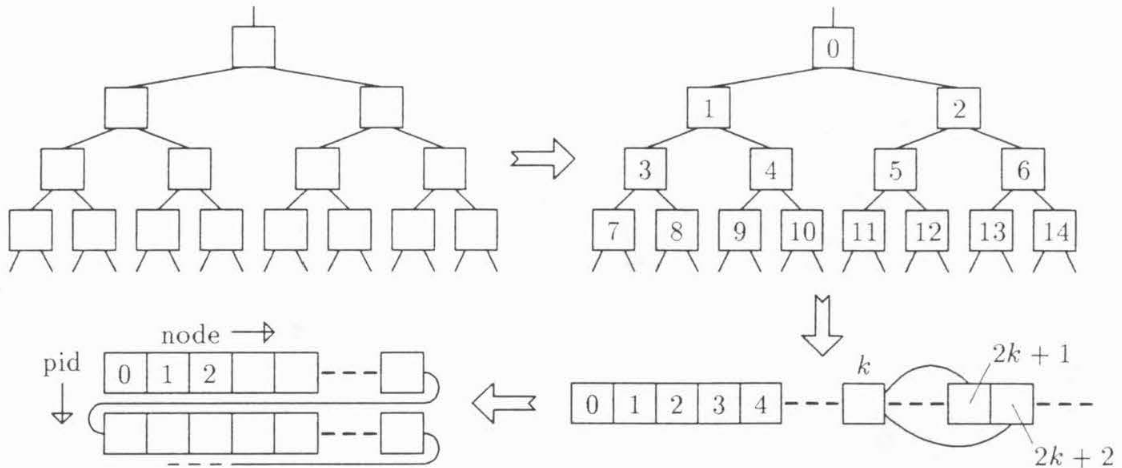

Figure 3.1 Mapping a binary tree to a multicomputer.

whose program file name is specified in the first argument, and whose ID is specified in the second and third arguments. The program file in this case is named pfac. The first process to be spawned by the caller should have an ID of (0,0).

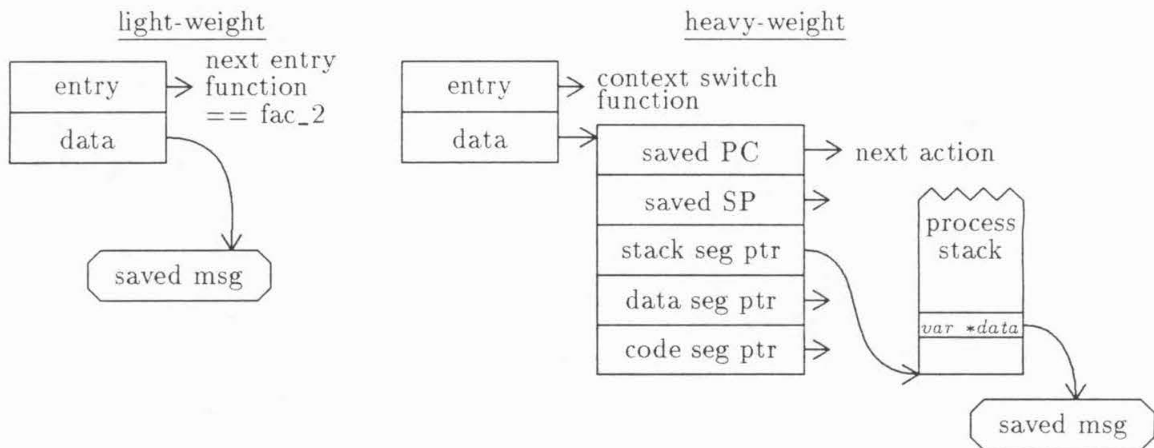

Figure 3.2 Process structure comparison.

The equivalence between the light-weight and heavy-weight processes is most obvious when the process structures of the two factorial processes are compared at the time that they are both waiting for their first reply message (Figure 3.2). The light-weight factorial process retains its message buffer in the `data` pointer of its process structure; the heavy-weight factorial process retains its message buffer in a pointer located on its program stack. The light-weight factorial process specifies its next action with the `entry` pointer of its process structure; the heavy-weight factorial process specifies its next action with the program

counter stored in its context structure.

The basic b-layer primitives can be summarized in the following list. The set is minimal, given the decision that processes are allowed to directly control process placement.

`char *b_malloc();`	Allocates a message buffer.
`b_free();`	Releases a message buffer.
`char *b_recvb();`	Receives a message.
`b_send();`	Sends and releases a message buffer.
`int mynode();`	Returns the `node` of the calling process.
`int mypid();`	Returns the `pid` of the calling process.
`int nnodes();`	Returns the number of nodes in the machine.
`spawn();`	Instantiates a new process.
`exit();`	Terminates the calling process.

3.1.2 The length-carrying layer (l-layer)

We shall introduce the general concept of layering by a very simple example. We will create a new set of functions, the l-layer functions, that are parallel to the b-layer functions with the exception that l-layer functions contain an additional function for accessing the length of a message buffer. To store the length information, we will make each message buffer a little larger than it needs to be, and store the length information in the extra space.

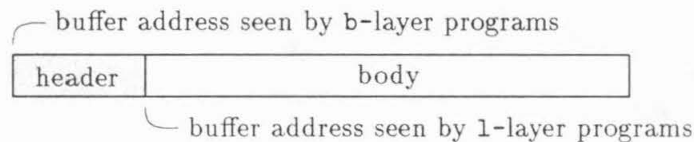

Figure 3.3 Structure of a l-layer message buffer.

That extra space is placed at the front of each message buffer and is called the *header* of the message; the rest of the message is called the *body*. We can hide the header by having l-layer functions work only with pointers to the body of the message. As a result, the l-layer functions become a super set of the b-layer functions.

```
1    typedef struct { int length; } HEADER;
2    #define BODY_OF(h) (h+sizeof(HEADER)) /* given header, find body */
3    #define HEAD_OF(b) (b-sizeof(HEADER)) /* given body, find header */
```

The HEADER structure shown above defines the content of the header for an l-layer message buffer. The only field in this header is an integer that contains the length of the message body. In order to allow all data types in the message body, headers should normally be padded to the maximum data alignment requirement of the hardware. In the interest of simplicity, however, padding is neglected for our examples.

```
 5      char *l_malloc(n)
 6          int  n;
 7      {
 8          char *p;

10          p = b_malloc(n + sizeof(HEADER));
11          ((HEADER *) p)->length = n;
12          return(BODY_OF(p));
13      }

15      char *l_recvb() { return(BODY_OF(b_recvb())); }
```

The two functions that return message buffers — receive and allocate — call the corresponding b-layer functions to get message buffers. When one is obtained, the pointer to the body of the buffer is returned by the functions. In addition, the l_malloc function stores the buffer length into the message header before it returns. Similarly, a function that takes a message buffer as input has to locate the real beginning of the message buffer before passing it to the corresponding b-layer function.

```
17      l_free(p) char *p; { b_free(HEAD_OF(p)); }

19      l_send(p,node,pid)
20          char *p;
21          int node, pid;
22      {
23          b_send(HEAD_OF(p), node, pid);
24      }

26      l_length(p)
27          char *p;
28      {
29          return(((HEADER *)HEAD_OF(p))->length);
30      }
```

This is the simplest application of layering; it does not change the message properties in any way. By adding more fields to the header structure, we can just as easily include any information that we would like to send along with a message, such as length of the message buffer, message type, and sender node and pid.

3.1.3 The non-blocking-receive layer (nb-layer)

A process running in a reactive-process programming environment should not monopolize the processor by running nonstop for long periods between receive calls, for if a process does not call a receive function, other processes in the same node will not get a chance to run.

A conventional multi-tasking operating system makes scheduling fair by interrupting a long-running process with a timer in order to wrest control away from a process. The same thing can be done in a Reactive-C implementation of a heavy-weight programming system by treating a timer-interrupt mechanism — as a process resource. A process, therefore, includes an interrupt mechanism and an interrupt service routine. When a process is interrupted by the timer, the interrupt service routine of the process calls a receive function to relinquish control.

A timer-interrupt is just one of the ways to make a process call a receive function periodically. While a timer may still be needed as a backup mechanism to stop runaway processes, the preferred method is to convert a non-reactive process into a reactive process by having the process call a receive function periodically during extended computations. Although the messages received may not be needed right away, they can always be queued by the process until they are needed.

It is better for the process to be de-scheduled at *choice points* in the program rather than at arbitrary points selected by the timer. Choice points are places in a program where much of the system resources used by the program, such as floating-point accelerators, direct-memory-access units, and processor registers, are released by the process as a normal part of the program execution. The amount of state information that needs to be saved and restored when a program is stopped and restarted at a choice point is usually small and can be reliably predicted during compile time.

Calling a receive function, either from a timer-interrupt handler or from a choice point, presents a problem, however. A process that relinquishes control by calling a receive function will not be re-started until a message is ready for it. As a result, a node can sit

idle with runnable processes suspended because there are no messages queued for them. Furthermore, if a suspended process does not receive any more messages, it will remain suspended indefinitely.

What we need is a receive function that does not block. This function can be implemented by having the process send a uniquely identifiable message to itself just before it calls a blocking receive function. We can create such messages by the same layering mechanism that we used for message length. Let us prefix the new functions with nb_, and let us invent a new receive function, nb_recv. A call to nb_recv has the same effect as a call to a normal receive function, except that in cases where a normal receive function would block, nb_recv returns a null pointer. (A nb_recv call may still return a null pointer at other times but it will always cause the process to release control first.)

Below is a set of routines that implement the nb-layer functions. We will list only those functions that are different in form from the l-layer functions. First of all, two private variables are needed. The token_got variable indicates whether a uniquely identifiable token message has been previously allocated. The token_msg pointer contains the token message if it is allocated and if the process is currently holding it; the pointer contains null otherwise.

```
1       typedef struct { int is_token; } HEADER;

3       static int   token_got = 0;
4       static char *token_msg = 0;

6       char *nb_recv()
7       {
8           char *p;

10          if(!token_got) { token_msg = l_malloc(0); token_got = 1; }

12          if( token_msg) { ((HEADER *)HEAD_OF(token_msg))->is_token = 1;
13                           b_send(HEAD_OF(token_msg), mynode(), mypid());
14                           token_msg = 0; }
15          p = b_recvb();
16          if(((HEADER *) p)->is_token) { token_msg = p; return(NULL); }
17          return(BODY_OF(p));
18      }
```

The first thing that the non-blocking nb_recv does is to check for the existence of the token message. If the token message has not been allocated, the function allocates it. Next,

the function checks to see if it is currently holding the token message. If it is, the function sends the token message to itself, so that a subsequent b_recvb call is guaranteed to return. Lastly, it calls b_recvb to get a message. If the message obtained is a token message, the token message is saved and null is returned. Otherwise, the message is returned.

```
20      char *nb_recvb()
21      {
22          char *p;

24          p = b_recvb();
25          while(((HEADER *) p)->is_token) { token_msg = p; p = b_recvb(); }
26          return(BODY_OF(p));
27      }

29      nb_send(p,node,pid)
30          char *p;
31          int node, pid;
32      {
33          ((HEADER *)HEAD_OF(p))->is_token = 0;
34          b_send(HEAD_OF(p), node, pid);
35      }
```

The blocking nb_recvb waits for a non-token message and returns that message when it is received. If a token message is received first, it is stored in token_msg and nb_recvb continues to wait for the next message. The nb_send function clears the token flag in the message header before sending the message because it can only send ordinary messages.

In order to improve efficiency, detection of token messages is ordinarily integrated into the kernel so that the kernel can defer token messages until the input message queue is otherwise empty. The primary effect is that processes with pending non-token messages are favorably scheduled. The side effect is that processes have a reliable method of determining whether the input queue of the node is empty. This special treatment of token messages constitutes the basis for indefinite-lazy computation in distributed simulation. This will be discussed in a later section.

3.1.4 Handler layering

Running a heavy-weight process inside a handler is an example of layering. We can also run a light-weight process inside a heavy-weight process, or a light-weight process inside another light-weight process. When each handler process controls just one reactive process, the ID

of the handler is sufficient to uniquely identify the process. When there may be more than one process inside a handler, a secondary pid needs to be included in the message header to distinguish them. Examples of handler layering are the Reactive Kernel for heavy-weight processes and simulators for light-weight processes.

```
 1  typedef struct { int pid2; } HEADER;          /* message header */

 3  struct PROC ptab[MAX_PID2];                   /* process table  */

 5  main_loop()
 6  {
 7      char *mesg;
 8      PROC *proc;

10      while(1)
11      {
12          mesg = b_recvb();
13          proc = ptab + ((HEADER *)p)->pid2;
14          (*proc->entry)(proc,BODY_OF(mesg));
15      }
16  }
```

Shown above is the main loop of a heavy-weight process capable of handling more than one light-weight process. The message functions resemble the l-layer functions, but with the second pid, rather than the message length, in the message header. The heavy-weight process repeatly calls `b_recvb` to get a message, finds the real destination process by the `pid2` field, and calls the entry function of the process. If this program fragment looks familiar, it is because this is the main loop of the Reactive-C kernel. The Reactive-C kernel is itself a reactive-process program.

Although the definition of a reactive-process program is fixed as stated in the beginning of Chapter 2, certain properties of the programming system are implementation-dependent. Handler layering provides a way of running a programming system with a different set of properties on top of another programming system. For example, assume that we have a programming system in which all messages to non-existing processes are thrown away. To implement systems such as the Cantor run-time system, messages to non-existing processes must be preserved. Suppose we were to support Cantor by running a Cantor handler under a reactive kernel. As far as the kernel is concerned, all messages will find their destination processes, namely, the Cantor handler processes. When the handler gets a message, the

message is beyond the jurisdiction of the kernel; the handler can do any number of things with it. In particular, the handler can queue messages for Cantor processes that have not yet been created.

Section 3.2 Message Type

It is convenient in many computations for a process to respond differently to different types of messages. In the factorial examples, there are three types of messages: the message from the parent, the first message to arrive from the siblings, and the second message to arrive from the siblings. These messages do not have to be distinguished by type because they are identified by their order of arrival. In the Reactive-C example, different responses to different messages are specified by storing different function pointers into the process structure after each message is received. In the b-layer version, the responses are specified by the locations in the program where `b_recvb` is called.

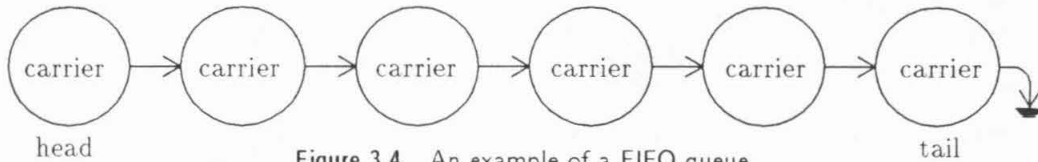

Figure 3.4 An example of a FIFO queue.

In the next example, however, it is necessary to distinguish messages by type. The FIFO (first-in-first-out queue) structure shown in Figure 3.4 can be constructed with the chain of `carrier` processes described in Listing 3.3. The `carrier` processes are connected into a singly linked list by the `next_node` and `next_pid` variables in each process. The FIFO is accessed by a reference to the head `carrier` and a reference to the tail `carrier`.

When an item is to be added to the FIFO, the item is sent as a message to the tail of the FIFO. The process at the tail of the FIFO spawns a new `carrier` for the new item and returns the reference of the new `carrier` to the caller. When an item is to be retrieved from the FIFO, a message is sent by the caller to the head of the FIFO. The process at the head of the FIFO sends its item and the reference of the next `carrier` to the caller. The process then removes itself from the FIFO. Message types are needed because the two

commands — "new item" and "retrieve item" — can arrive in any order when a FIFO is only one element long.

```
 1  typedef struct { int type, value, node, pid; } REQ_MESG;

 3  main()
 4  {
 5      REQ_MESG *req;
 6      int        value;
 7      int        caller_node, caller_pid;
 8      int          next_node,   next_pid;

10      while(1)
11      {
12          req = (REQ_MESG *) b_recvb();

14          switch(req->type)
15          {
16              case ADD_VALUE: spawn_anywhere("carrier",&next_node,&next_pid);
17                              req->type    = SET_VALUE;
18                              b_send(req, next_node, next_pid);
19                              break;

21              case SET_VALUE: value       = req->value  ;
22                              next_node   = INVALID_NODE;
23                              next_pid    = INVALID_PID ;
24                              caller_node = req->node   ;
25                              caller_pid  = req->pid    ;
26                              req->node   = mynode()    ;
27                              req->pid    = mypid ()    ;
28                              b_send(req, next_node, next_pid);
29                              break;

31              case GET_VALUE: req->value  = value     ;
32                              caller_node = req->node;
33                              caller_pid  = req->pid ;
34                              req->node   = next_node;
35                              req->pid    = next_pid ;
36                              b_send(req, next_node, next_pid);
37                              exit(0);
38          }
39      }
40  }
```

Listing 3.3 The carrier program for building FIFO.

When a carrier receives an ADD_VALUE message, it spawns another carrier, and the message is passed to the new carrier after its message type is set to SET_VALUE (16–19). The spawn_anywhere function will spawn the specified process on some available node and return the node and pid of the process in the next_node and the next_pid variables.

When a carrier receives a SET_VALUE message, the process is the new tail process. The value field of the message is copied into the value variable of the carrier. The next

reference of the carrier is initialized to a null ID. The ID of the carrier is written into the message, and the message is returned to the caller (21–29). After the message is received by the caller, the caller's tail reference is updated.

When a carrier receives a GET_VALUE message, its value and its next-carrier reference are copied into the message. The message is sent back to the caller and the process exits (31–38).

Section 3.3 Discretion on Receive

Discretion on receive means allowing a process to select certain messages to consume while deferring other messages. The Reactive-C, the b-layer, and other simple layered variants all have the same message property in that they do not supply any mechanisms for discretion; their processes have no choice but to take messages in the order they arrive. Discretion can, however, be implemented inside a process.

3.3.1 Discretion using b-layer functions

An example in which discretion is implemented in the program is a merge-sort program, in which the list to be sorted is split recursively along the branches of a *time-on-target tree* until every processing node in the machine is used. The machine should have a power-of-two number of nodes to support this *doubling* approach.

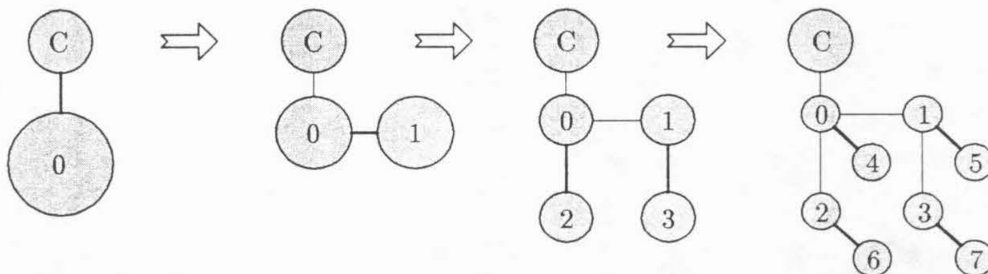

Figure 3.5 Expansion steps in the merge-sort program.

At the beginning of the sort, the zeroth-generation process is created in a machine with 2^n nodes, and a list of numbers to be sorted is sent to the process as a message. The zeroth-generation process then proceeds to fill the machine with processes in a total of n expansion steps. In the kth expansion step, every process in the machine creates a new

kth-generation process, giving half of its list to the new process and keeping the other half for itself. After n steps, there will be 2^n processes on the machine, each holding $1/2^n$th of the original list.

The processes begin to sort their share of the list locally. When sorting is complete, the expansion steps are reversed to merge the fragmented lists. In the kth merging step (k decreasing), each kth-generation process sends its list back to its parent in a reply message. After n steps, only the zeroth-generation process remains. The list that it now holds is the sorted version of the original list.

When the process structure is fully instantiated, each kth-generation process has a sibling for every generation number from $k+1$ to n. Since the computation is asynchronous, returning messages from the siblings may arrive in a different order from the order of the merging steps. Since each process needs to consume reply messages from its siblings in the order of decreasing generation number, each sibling will need a different message type for its reply message, and the process will selectively wait for a certain message in each merging step.

The sorting program in Listing 3.4 first appeared in "Multicomputers: Message-Passing Concurrent Computers" [2]. The first version of the program, which uses integer-based types, was written by C.L. Seitz; the version appearing in Listing 3.4 and in the *IEEE* paper was modified by the author to use pointer-based types.

```
 1  typedef struct MESG MESG;                 /* Message header structure.          */
 2  struct MESG { int    pnode, ppid;         /* Address of the parent process.     */
 3                int    tbase       ;        /* Base for time-on-target tree.      */
 4                int    len         ;        /* Number of elements in the vector.*/
 5                MESG   **type      ;} ;     /* Type field for filtering message.*/
 6  #define BUF(v) ((double *)(v+1))          /* Data follows MESG immediately.     */

 8  unsigned int this_node, this_pid, node_cnt;

10  main()
11  {   MESG *v;

13      this_node = mynode();                 /* Node number of this process.       */
14      this_pid  =  mypid();                 /* Pid number of this process.        */
15      node_cnt  = nnodes();                 /* number of nodes in this machine. */

17      v = (MESG *) b_recvb();               /* Receive list from parent process.*/
```

```
18      if(v->len > 1) merge_sort(v);     /* Sort the list.                  */
19      b_send(v, v->pnode, v->ppid);     /* Send result back to parent.     */
20  }

22  merge_sort(v)
23      MESG *v ;
24  {   unsigned l1, l2, i, new_node;
25      MESG *v1, *v2, *v3;
26      double *d, *s, *b1, *b2;

28      l1 = ( v->len + 1 ) / 2;          /* Break the list into two lists.  */
29      l2 = ( v->len     ) / 2;
30      v1  = (MESG *) b_malloc(sizeof(MESG)+sizeof(double)*l1);
31      v2  = (MESG *) b_malloc(sizeof(MESG)+sizeof(double)*l2);
32      for(i = v1->len = l1, d = BUF(v1), s = BUF(v); i--; ) *d++ = *s++;
33      for(i = v2->len = l2, d = BUF(v2)             ; i--; ) *d++ = *s++;

35      new_node = this_node ^ v->tbase;          /* Next node to be used for */
                                                  /* spawning a sibling.      */
37      v1->tbase = v2->tbase = v->tbase << 1;    /* New base for building    */
                                                  /* time-on-target tree.     */
39      if(v1->len > 20 && new_node < node_cnt)
40      {                                         /* If list is too long and  */
41          spawn("msort",new_node,this_pid,""); /* if next node is valid    */
42          v1->pnode = this_node       ;         /* spawn a sibling          */
43          v1->ppid  = this_pid        ;         /* and send it a list.      */
44          v1->type  = &v1             ;         /* The type field holds the */
45          b_send(v1,new_node,this_pid);         /* address of the msg ptr.  */
46          v1 = 0;                               /* Msg ptr is set to null.  */

48      } else if(v1->len > 1) merge_sort(v1);    /* Sort if cannot split.    */

50             if(v2->len > 1) merge_sort(v2);    /* Sort the other list.     */

52      while(!v1) { v3 = (MESG *) b_recvb(); *v3->type = v3; }

54      for(b1 = BUF(v1), b2 = BUF(v2), d = BUF(v); l1 || l2; )  /* merge.    */
55      {   while(l1 && (!l2 || (l2 && *b1 <= *b2))) { l1--; *d++ = *b1++; }
56          while(l2 && (!l1 || (l1 && *b2 <= *b1))) { l2--; *d++ = *b2++; }
57      }
58      b_free(v1); b_free(v2);
59  }
```

Listing 3.4 The merge-sort program.

In each level of recursion where a sibling is created (41), the type field of the message for the sibling is filled with the address of the automatic pointer variable, `v1` (44). These `v1` pointers on the program stack are set to null before the merging phase (46), which begins when the recursive **`merge_sort`** function starts to unwind. Since there is at most one sibling created in each level, the list sent to each sibling must contain an address that is different from the others — the address of the `v1` pointer in effect when the sibling is created.

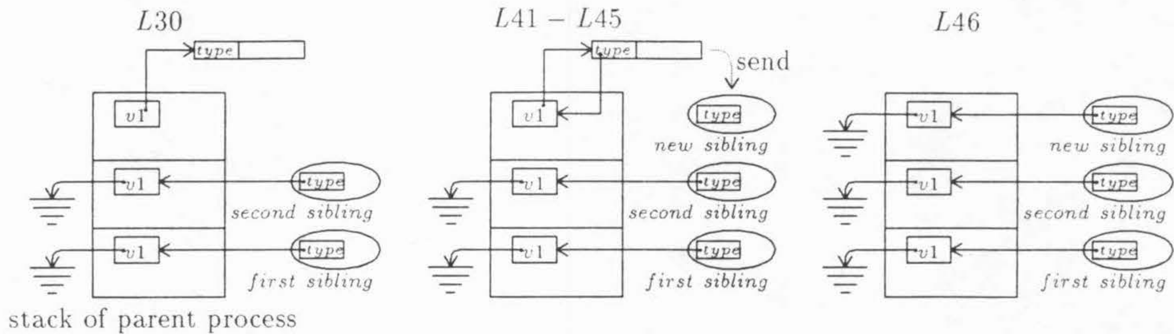

Figure 3.6 Giving away a list for the third time (stack grows up).

After the expansion phase, the program progresses to line 48 and 50, where the remaining numbers are sorted using a sequential merge-sort algorithm performed by the same `merge_sort` function. During the merging phase, each sibling returns a message of the type it was assigned (19). A process selectively waits for the message for the current recursion level by polling the `v1` pointer at that level; at the same time, the process repeatly requests a message and stores it into the pointer whose address is equal to its message type (52).

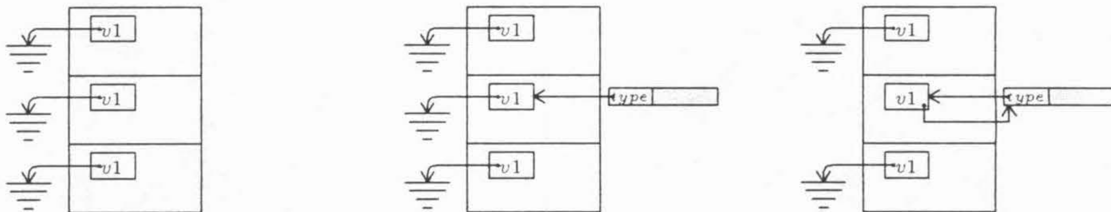

Figure 3.7 Getting an out-of-sequence reply.

When the program reaches line 52, `v1` can take on one of the three possibilities:

1. `v1` is not null, because its list has not been given away;
2. `v1` is null, because although its list has been given away, a reply has not been received; or
3. `v1` is not null, because although its list has been given away, the reply was received while the program was waiting for a different reply.

The distribution of work is accomplished by *divide and conquer;* the merge-sort example can be used as a *template* for other divide-and-conquer applications. Assigning deferred

messages into holding pointers is sufficient for this application because no more than one message for each type needs to be queued. When more than one message of each type must be deferred, the process has to store them in a more general list structure.

3.3.2 The RPC-discretion layer (r-layer)

While discretion is used in the merge-sort program, the process still takes messages in the same order they arrive. However, some programs can be made simpler by creating an illusion that messages are dispensed by the kernel in an order other than first come, first serve. Such effects can be achieved with layering as well.

The implementation of a *remote procedure call* (RPC) is one example. Suppose we want to make available a generic file operation, `read`, implemented by message exchange with a file controller, a process responsible for maintaining a file. A prototype function might look like the one in Listing 3.5.

```
 1  typedef struct { int fs_node   ;              /* Structure of one entry of */
 2                   int fs_pid    ; } FSTRUCT; /* the process's file table. */

 4  FSTRUCT file_tab[20];                       /* The process's file table. */

 6  typedef struct { int operation;             /* Format of request message */
 7                   int my_node   ;            /* to be sent to the file    */
 8                   int my_pid    ;            /* server process to request */
 9                   int read_size; } REQUEST; /* for a read operation.     */

11  #define OP_READ 3                           /* Code read request.        */

13  read(fd,buf,len)
14      int fd, len;
15      char *buf;
16  {
17      REQUEST *request;
18      char     *reply;

20      request            = (REQUEST *) b_malloc(sizeof(REQUEST));
21      request->operation = OP_READ ;
22      request->my_node   = mynode();
23      request->my_pid    = mypid ();
24      request->read_size = len      ;

26      b_send((char *) request, file_tab[fd].fs_node, file_tab[fd].fs_pid);

28      reply = b_recvb();
29      bcopy(reply,buf,len);
30      b_free(reply);
31      return(len);
32  }
```

Listing 3.5 An incorrect implementation of the C `read` function.

The file_tab array contains the node and pid of all file-controller processes accessible by this process. The read function sends a request to a file controller selected from file_tab using fd as the index. When the file controller finishes reading the requested amount of data, the data is sent back in a message. The function is shown to be waiting for the reply using the normal b_recvb function.

```
28      reply = b_recvb();
```

However, the b_recvb function is not adequate because it may pick up the wrong message if another message arrives before the reply message. A receive-discretion mechanism must be used to ensure that only the reply message for the read function is returned. The reply messages, called the RPC messages, must therefore be distinguishable from other messages that the process uses. Furthermore, messages that arrive before the reply message must be queued and released in a transparent way so that the requesting program cannot distinguish a local read from a RPC read.

The r-primitives implement the new message properties by layering and by adding two more functions: RPC send and RPC receive. The message header for this layer contains a RPC flag and a chaining pointer. Since RPC calls do not interleave in a process, a process can have no more than one outstanding reply message at any one time. Storing one distinguished type in a Boolean variable is therefore sufficient for positively identifying a reply message. The defer_h and defer_t pointers are used to implement a queue for non-RPC messages. The next pointer in the message header is used to chain deferred messages into a linked list for the queue.

```
1       typedef struct HEADER { int             is_rpc;
2                               struct HEADER *next; } HEADER;

4       #define BODY_OF(h) (h+sizeof(HEADER)) /* given header, find body  */
5       #define HEAD_OF(b) (b-sizeof(HEADER)) /* given body, find header  */

7       HEADER *defer_h, *defer_t;  /* queue for holding non-rpc messages */
```

The r_recvb function replaces the b_recvb function for receiving normal messages. Instead of calling b_recvb immediately, it checks the queue for any deferred messages. If there are

deferred messages, a message is removed from the queue and returned. Otherwise, `b_recvb` is called.

```
 9      char *r_recvb()
10      {
11          char *p;

13          if(defer_h) { p = (char *) defer_h;
14                        defer_h = defer_h->next;
15                        return(BODY_OF(p)); }

17          return(BODY_OF(b_recvb()));
18      }
```

The `r_recvrpc` function is a function that waits for a reply message. It calls `b_recvb` repeatly until a reply message is received. The RPC message is then returned. Meanwhile, all non-RPC messages that have arrived are stored in the queue.

```
20      char *r_recvrpc()
21      {
22          char *p;

24          while(p = b_recvb())
25          {
26              if(((HEADER *)p)->is_rpc ==  1) return(BODY_OF(p));
27              if(defer_h) defer_t = defer_t->next = (HEADER *) p;
28                     else defer_t = defer_h       = (HEADER *) p;
29              ((HEADER *) p)->next = 0;
30          }
31      }
```

The `r_send` function clears the RPC flag before sending the message. The `r_sendrpc` function sets the flag before sending the message.

```
33      r_send(p,node,pid)
34          char *p;
35          int node, pid;
36      {
37          ((HEADER *)HEAD_OF(p))->is_rpc = 0;
38          b_send(HEAD_OF(p), node, pid);
39      }

41      r_sendrpc(p,node,pid)
42          char *p;
43          int node, pid;
44      {
45          ((HEADER *)HEAD_OF(p))->is_rpc = 1;
46          b_send(HEAD_OF(p), node, pid);
47      }
```

If replies from the file controller are sent using `r_sendrpc`, the `read` function can be correctly defined as:

```
62  read(fd,buf,len)
63      int fd, len;
64      char *buf;
65  {
66      REQUEST *request;
67      char      *reply;

69      request            = (REQUEST *) r_malloc(sizeof(REQUEST));
70      request->operation = OP_READ ;
71      request->my_node   = mynode();
72      request->my_pid    = mypid ();
73      request->read_size = len     ;

75      r_send((char *) request, file_tab[fd].fs_node, file_tab[fd].fs_pid);

77      reply = r_recvrpc();
78      bcopy(reply,buf,len);
79      r_free(reply);
80      return(len);
81  }
```

Listing 3.6 A correct implementation of the C `read` function.

The introduction of the RPC message type makes it possible for standard utility functions to be implemented by message passing; however, the use of RPC and other discretion mechanisms in utility functions has the potential effect of diminishing the available concurrency in a program. For example, the use of `read` in a program forces all non-RPC messages to wait while `read` is being completed, regardless of whether some of these messages can be consumed without waiting for `read` to complete.

3.3.3 The CSP-discretion layer (csp-layer)

Layering can also be used to implement the CSP synchronization primitives. In Hoare's definition of CSP, send and receive are performed by $P!expression$ and $P?variable$, respectively, where P is the process reference of the communication partner. In later CSP variants, such as OCCAM, send and receive are performed by $C!expression$ and $C?variable$, respectively, where C is the channel connecting the sender and the receiver. Both send and receive functions will block until the communication partner has completed the complementary operation on the same channel. The send and the receive functions can be implemented with a mutual exchange of messages between the two processes. We will show an implementation of CSP with channels.

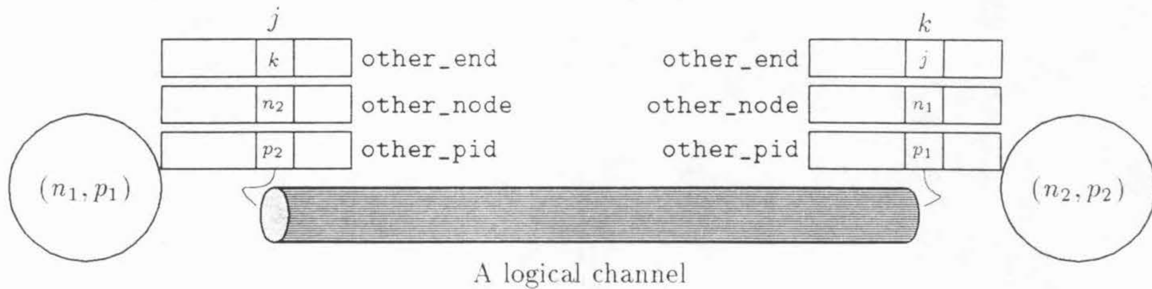

Figure 3.8 Structure of a channel in a channel-based CSP implementation.

Since messages associated with different channels may arrive in an order other than the one in which CSP communication is to take place, messages must be tagged with a type field, and those that have arrived early must be deferred. Let us construct a channel using two logical communication endpoints, one each in the sender and the receiver. If we identify the endpoints in each process by a small array index, the connectivity of the channels can be completely described by four arrays in each process:

```
 1  typedef struct { int type; int value; } CSP_MSG;

 3  int      other_end [MAX_CHAN];
 4  int      other_pid [MAX_CHAN];
 5  int      other_node[MAX_CHAN];
 6  CSP_MSG *chan_queue[MAX_CHAN];
```

In each process, the entries `other_node[j]` and `other_pid[j]` identify the process at the other end of channel `j`. The entry `other_end[j]` is channel `j`'s identity at the other side of the channel; *ie.*, the channel `j` on this side and the channel `other_end[j]` on the other side both refer to the same channel. An unambiguous typing system can be constructed by giving messages for channel `j` the type `other_end[j]`. The `chan_queue` array is an array of pointers that holds queued messages for channels. Since each channel can have no more than one pending message, only one pointer for each channel is needed for buffering early messages. The `csp_send` and the `csp_recv` functions can be written as:

```
 8  csp_send(chan,expr)
 9      int chan, expr;
10  {
11      CSP_MSG *sp = (CSP_MSG *) b_malloc(sizeof(CSP_MSG));

13      sp->value = expr            ;
14      sp->type  = other_end[chan];
15      b_send(sp, other_node[chan], other_pid[chan]);
```

```
17      while(!chan_queue[chan]) { sp = (CSP_MSG *) b_recvb();
18                                   chan_queue[sp->type]  = sp; }

20      b_free(chan_queue[chan]); chan_queue[chan] = 0;
21  }

23  csp_recv(chan,var)
24      int chan, *var;
25  {
26      CSP_MSG *sp = (CSP_MSG *) b_malloc(sizeof(CSP_MSG));

28      sp->type  = other_end[chan];
29      b_send(sp, other_node[chan], other_pid[chan]);

31      while(!chan_queue[chan]) { sp = (CSP_MSG *) b_recvb();
32                                   chan_queue[sp->type]  = sp; }
33      *var = sp->value;

35      b_free(chan_queue[chan]); chan_queue[chan] = 0;
36  }
```

In both functions, a message buffer is allocated and sent to the other side of the channel. The process then waits for a reciprocal message from the other side, if one has not already arrived. The process frees that message, clears the message-queuing pointer, and returns. The only difference between the send and the receive functions is that in `csp_send`, the value to be sent is stored in the `value` field before the send. In `csp_recv`, the value is retrieved from the message received before it is freed.

A more elaborate implementation of a superset of CSP were created by A.J. Martin [18] and Marcel van der Goot.

3.3.4 A more general type-discretion layer (t-layer)

When user-defined message types are needed in a program with type discretion, the type information can be encoded in the message body, and discretion can be handled by the program itself, as in the `merge_sort` example. Alternatively, we can hide the message type in the message header, as in the `t`-layer example below.

In the `t`-layer, the program supplies a type for the message when it is sent with the `t_send` function. The `t_send` function stores the message type into the header before the send. In the receive function, the program specifies the type of message to wait for. Messages of other types are queued if they arrive before a message of requested type is received.

```
1      typedef struct HEADER { int            type;
2                              struct HEADER *next; } HEADER;

4      t_send(p,node,pid,type)
5          char *p;
6          int node, pid, type;
7      {
8          ((HEADER *)HEAD_OF(p))->type = type;
9          b_send(HEAD_OF(p), node, pid);
10     }
```

The two pointer arrays, `defer_h` and `defer_t`, implement the queues. This queue structure imposes a limit on the range of usable types, but a more general queue structure can be used instead. The `t_recvb` function takes a message type as an argument. It waits for and puts messages into the respective queue while the queue of the desired type remains empty. When the queue is non-empty, a message is removed from the queue and returned to the program.

```
12     HEADER *defer_h[MAX_TYPE], *defer_t[MAX_TYPE];

14     char *t_recvb(type)
15         int type;
16     {
17         char *p; int t;

19         while(!defer_h[type])
20         {
21             p = b_recvb();
22             t = ((HEADER *) p)->type;
23             if(defer_h[t]) defer_t[t] = defer_t[t]->next = (HEADER *) p;
24                       else defer_t[t] = defer_h[t]       = (HEADER *) p;
25             ((HEADER *) p)->next = 0;
26         }

28         p = (char *) defer_h[type];
29         defer_h[type] = defer_h[type]->next;
30         return(BODY_OF(p));
31     }
```

Section 3.4 Other Layers

3.4.1 A flow-controlling layer (f-layer)

Layering can also be used to implement transparent flow control of messages. Suppose we have an application where it is necessary to limit the number of unconsumed messages produced by each process. We can introduce a layer in which an acknowledgment message is sent for every message consumed, and have the send function block until the number of messages sent is no more than a preset value over the number of acknowledgments received.

In the following example, when more than ten messages are outstanding, the send routine will call `b_recvb` to wait for messages. Since `b_recvb` does not distinguish normal messages from acknowledgment messages, we will use the `r`-layer mechanism to selectively wait for acknowledgment messages in the `f`-layer routines:

```
1       typedef struct { int node, pid, is_ack;
2                          struct HEADER   *next; } HEADER;

4       #define BODY_OF(h) (h+sizeof(HEADER)) /* given header, find body */
5       #define HEAD_OF(b) (b-sizeof(HEADER)) /* given body, find header */
6       #define COUNT_MAX 10

8       static int        o_count;  /* number of outstanding messages.   */
9       HEADER *defer_h, *defer_t;  /* queue for holding normal messages.*/
```

Since the receiver has to send an acknowledgment to the sender, the `f`-layer message header must contain the ID of the of the sending process in addition to the `next` field of the `r`-layer header. The header must also contain the flag `is_ack` to differentiate a normal message from an acknowledgment message.

```
11      char *f_recvb()
12      {
13          HEADER *p, *q;

15          if(defer_h) { p = defer_h; defer_h = defer_h->next; }
16                  else { while(1) { p = (HEADER *) b_recvb();
17                                    if(!p->is_ack) break;
18                                    o_count--; b_free(p); }     }

20          q = (HEADER *) b_malloc(sizeof(HEADER));
21          q->is_ack = 1; b_send(q,p->node,p->pid);

23          return(BODY_OF(((char*)p)));
24      }
```

In the receive function, if there are any queued messages, one message is removed from the queue. If the queue is empty, the function calls `b_recvb` repeatedly until a normal message is received. In both cases, an acknowledgment is sent to the sender and the message returned to the caller. While waiting for a normal message, any acknowledgment messages received cause the outstanding message counter to decrement.

```
26      char *f_send(p,node,pid)
27          char *p;
28          int node, pid;
29      {
30          HEADER *q;
```

```
32          while(o_count >= COUNT_MAX)
33          {
34              q = (HEADER *) b_recvb();
35              if(q->is_ack) { o_count--; b_free(q);                        }
36                       else { if(defer_h) defer_t = defer_t->next = q;
37                                      else defer_t = defer_h        = q;
38                              q->next = 0;                                 }
39          }

41          q = (HEADER *) HEAD_OF(p);
42          q->node   = mynode();
43          q->pid    = mypid ();
44          q->is_ack =        0;

46          o_count++;
47          b_send((char *) q, node, pid);
48      }
```

In the send function, as long as the counter value is larger than ten, `b_recvb` is called to obtain a message. If the message is a normal message, it is queued; if the message is an acknowledgment message, the counter is decremented. If the outstanding message counter is or has become less than `COUNT_MAX`, the outgoing message is sent and the outstanding message counter is incremented.

If the communication graph is fixed (*ie.*, channel-like connectivity), it is more efficient to have a separate counter for each channel, and to send an acknowledgment for every `COUNT_MAX/2` messages in each channel. Each acknowledgment message represents the consumption of `COUNT_MAX/2` messages.

3.4.2 The CK primitives

The old `CK` (*Cosmic Kernel*) primitives, the original message primitives for the *Cosmic Cube*, can also be built from the reactive primitives by layering. The primitives are defined around a data structure called a message descriptor. (This is very similar to the way in which the `C` standard I/O functions are defined around the `FILE` structure.)

```
typedef struct{
                short          node;
                short          pid;
                short          type;
                short          seg;
                char           *buf;
                unsigned short msglen;
                unsigned short buflen;
                short          lock;
              } MSGDESC;
```

We have treated messages as information carriers. Sending and receiving messages are similar to memory allocation operations in `C`, in that it is the carrier that is affected. The transfer of information is merely a side effect of moving these carriers. The `CK` primitives, on the other hand, treat messages as information encoded in binary bit patterns and stored in arrays of memory cells. When a message is being sent, the system fetches the information from a designated storage buffer; when a message is received, the system writes the information into a designated storage buffer.

Since the send and receive requests are not always completed when the send and receive functions return, processes are allowed to run asynchronously while the transactions are being completed. However, in order to avoid access conflicts in the buffers, a lock variable is used for each transaction to indicate whether the transaction has completed. The `buf` and `lock` variables in the `MSGDESC` structure are used to hold the buffer and the completion lock.

When a message descriptor is used to send a message, the `node` and `pid` fields store the `ID` of the destination process. The `type` and `msglen` fields store the message type and the length of the message. The `buf` pointer references a memory buffer where the message is contained. When send is called, the call will return immediately, but the lock remains set until the send operation is complete.

When a message descriptor is used to receive a message, the `type` field is set to the type of the message to be received. The `buf` field is set to reference the memory buffer where the message body is to be stored. The `buflen` field contains the size of the memory buffer. When a receive function is called, the call will return immediately, but the lock remains set until the receive operation is complete. When receive is complete, the `node` and `pid` fields contain the `ID` of the sending node. The `msglen` field contains the actual length of the message. Incoming messages that do not have matching receive requests waiting for them will be queued.

```
typedef struct HEADER { int      snode, spid;
                        int           msglen;
                        int             type;
                        struct HEADER *next; } HEADER;
```

Other functions in the CK primitives are described in detail in the CK programming guide [4]. In making the transition from the CK primitives to the RK *(Reactive Kernel)* primitives, which we use on our machines, a compatibility library was created for the old CK programs by layering. The message header for a CK layer would therefore contain the sender node and pid, the message length, and the message type. It would also contain a pointer for making linked lists for discretionary receives. The details and the listings for the implementation have been omitted for brevity.

3.4.3 The RK primitives (x-primitives)

The RK primitives, or x-primitives, can also be built from the b-layer functions by layering. The RK primitive set includes the following list of functions:

```
char  *xmalloc();     ---> b_malloc();
char    *xrecv();     ---> nb_recv();
char   *xrecvb();     ---> b_recvb();
char *xrecvrpc();     ---> r_recvrpc();
         xsend();     ---> b_send();
      xsendrpc();     ---> r_sendrpc();
         xfree();     ---> b_free();
int    xlength();     ---> l_length();
```

The xmalloc, xrecvb, xsend, and xfree functions are equivalent to the b_malloc, b_recvb, b_send, and b_free functions, respectively. The xrecv function is equivalent to the nb_recv function, the non-blocking receive. The xlength function is equivalent to the l_length function, the function that returns message length. The RPC functions are similarly equivalent to those of the r-layer functions.

The RK primitives can therefore be implemented using a combination of l-layer, nb-layer, and r-layer. However, in the actual implementation of the Reactive Kernel, all three of the layers are incorporated into the basic kernel for greater efficiency.

The x-primitives and associated functions will be discussed in the next section in conjunction with the description of the *Cosmic Environment*, the generic multicomputer op-

erating environment in which the x-primitives are supported as the primary programming system.

Section 3.5 Layering on Light-Weight Processes

Any layering that applies to heavy-weight processes and that makes sense in the context of the light-weight processes can be applied to light-weight processes as well. If we represent the kernel, handler, layer routines, and user program as four separate components, the chain of control flow is shown in Figure 3.9.

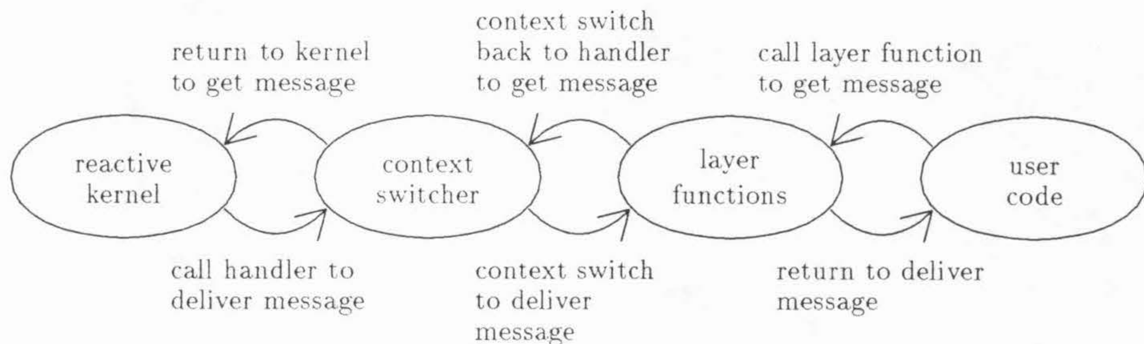

Figure 3.9 Control flow for heavy-weight processes.

The control flow for light-weight processes, shown in Figure 3.10, is identical except for the absence of the handler component.

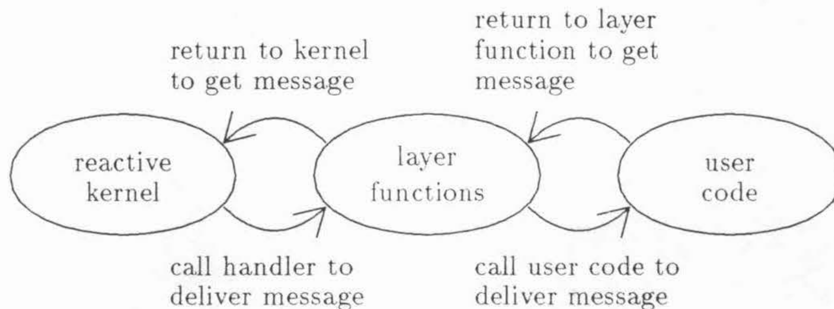

Figure 3.10 Control flow for light-weight processes.

Although these two programming models are essentially interchangeable, light-weight processes are more efficient in most machines because they avoid the context-switch cost.

However, programs composed of light-weight processes are more difficult to develop because processes are not protected against each other in case of a programming error. The processes must, in practice, coexist in the same address space.

Chapter 4 Cosmic Environment

The *Cosmic Environment*, or CE, is a multicomputer programming specification that also exists as an implementation on a number of multicomputers. Details for using CE can be found in "The C Programmer's Abbreviated Guide to Multicomputer Programming"[3]. We will concentrate here on the reasoning behind the design of our implementation, but first we will give a short definition of the *Cosmic Environment Specification*. The specification covers the process model, the message system, and the library functions.

Section 4.1 The Cosmic Environment Specification

The agents of a computation in CE are:

Processes: Each process is identified by a unique process ID, which is a (node,pid) pair. Node identifies the multicomputer node containing the process, and pid distinguishes one process from another on the same multicomputer node.

Messages: Each message is tagged by the ID of its destination process.

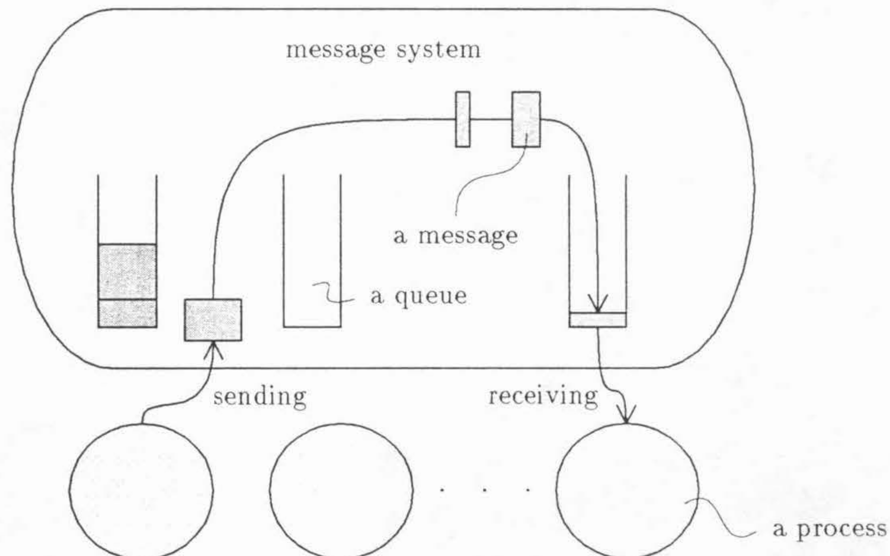

Figure 4.1 Elements of a computation.

Message system: The message system accepts messages from the processes, routes them according to their destination process ID, and delivers them to their destination processes. Messages are queued enroute to

their destinations; message order between any pair of processes is preserved.

In CE, a process can allocate and release message buffers, send and receive messages, create other processes, and terminate itself. The functions available to a C program are:

```
char *xmalloc(n)                    : Allocates and returns a message buffer
    unsigned n;                       sufficient for n bytes of data.

xfree(p)                            : Releases a message buffer.
    char *p;

char *xrecvb()                      : Waits for and returns a message from the
                                      message system.

char *xrecv()                       : Returns a message from the message system
                                      if one is available; returns a null pointer
                                      otherwise.

xsend(p,node,pid)                   : Frees the message buffer, p, from the calling
    char *p; int node, pid;           process, and sends the message buffer to the
                                      process whose ID is (node,pid).

spawn(name,node,pid,option)         : Runs the program called name and assigns it
    char *name, *option; int node, pid;  the ID (node,pid).

int mynode()                        : Returns the node number of the calling
                                      process.

int mypid()                         : Returns the pid number of the calling
                                      process.

exit()                              : Terminates the calling process.
```

This specification is short and simple. When our emphasis is on the study of multicomputer programming, we do not need unnecessary features to distract us; what we do need is a system that does not inhibit creativity. CE preserves the value of our work by making it easy to provide efficient implementations for its specification on many multicomputers that are otherwise software-incompatible.

Our CE specification was designed with the following two rules in mind:

1. Programming systems should be portable.
2. Programming manuals are evil.

The first design rule regards the *portability* of CE. A programming environment is portable if many types of machines can be made to support the programming environment. Portability is easy to achieve with CE because its functions are easy to provide in most multicomputers and multiprocessors. CE can be supported at the user-program level with a compatibility library, or at the system level with a reactive kernel. The reactive kernel makes kernel implementation or substitution simple because it does not require much support from the hardware.

The second design rule regards programming manuals. Manuals are a necessary evil. Therefore, whenever possible, CE has been made easy to explain in order to shorten the manuals. Besides this obvious advantage for people who do not enjoy reading manuals, CE has become simple and intuitive because making it easy to explain has also made it easy to use.

Having a short programming manual is self-rewarding. In an evolving system where old features are constantly being revised or dropped and new features are constantly being added, keeping a large manual up-to-date is a non-trivial task for a small research group. By keeping the manual simple, we not only make manual revision less laborious, but also make system improvement easier, since we are not obliged to support any mis-features that have not been previously documented. Our view is that the less a user has to know in order to efficiently complete the work, the better.

Section 4.2 Our Cosmic Environment Implementation

An implementation of the CE specification is a programming environment that embodies the specification. Currently we have implementations that contain drivers for the *Cosmic Cube*, the *iPSC/1*, the *iPSC/2*, the *Symult 2010*, and for the *ghost cube* — a set of network-connected workstations treated as a single multicomputer. (For historical reasons, we retain

the use of the word "cube" to mean a multicomputer even though not all multicomputers are binary n-cubes.) Other implementations that use shared memory for message passing exist for the Sequent and for the Cray X-MP.

4.2.1 Structure of our CE implementation

We start with the process model. A *process group* contains a set of processes connected to the message system (Figure 4.2). Processes communicate with each other by sending and receiving messages, and they refer to each other by means of their process IDs.

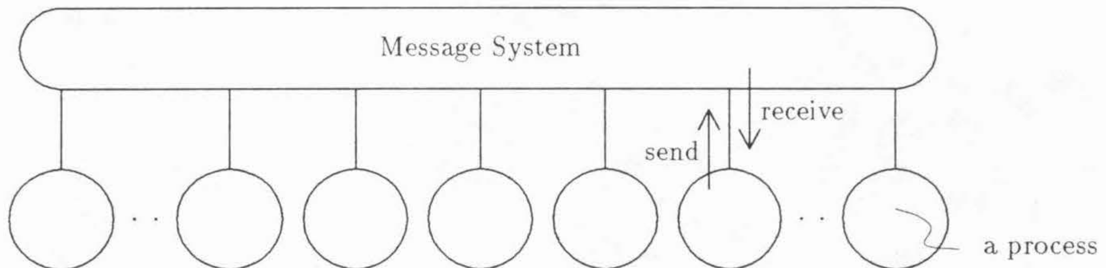

Figure 4.2 A process group.

In order for the set of processes to communicate with the outside world, the logically uniform message system is physically partitioned into two parts: One resides in the multicomputer and is called the node message system; the other resides outside of the multicomputer and is called the host message system. The two parts are connected by a message gateway, and the separation is made transparent to the processes (Figure 4.3). Processes are then allowed to run either on the hosts or on the nodes.

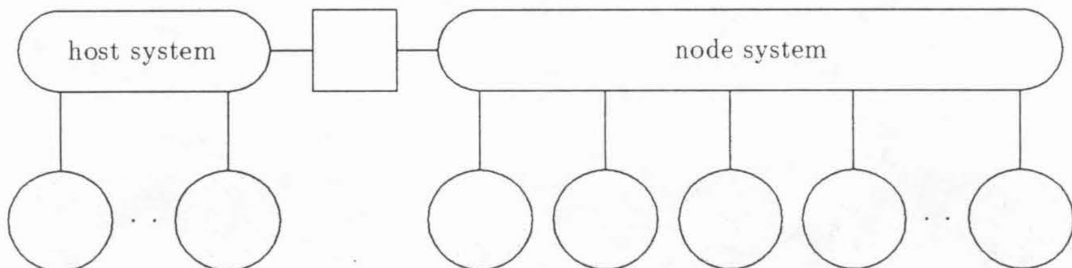

Figure 4.3 Partitioning into two parts.

Since our multicomputers are used in classes for student experiments, there are many more users who need to use the multicomputers than there are available multicomputers. But since most experiments require fewer nodes than are available in a multicomputer, we

want to support several users simultaneously on the same multicomputer. *Space sharing* is the sharing of a multicomputer by more than one user such that each user is given a separate subset of nodes in a multicomputer. The programming environment within each subset is indistinguishable from one in which the user owns an entirely separate multicomputer having the same number of nodes in the subset. Our message gateway must therefore interface with more than one host message system and pass messages to and from each user's nodes (Figure 4.4).

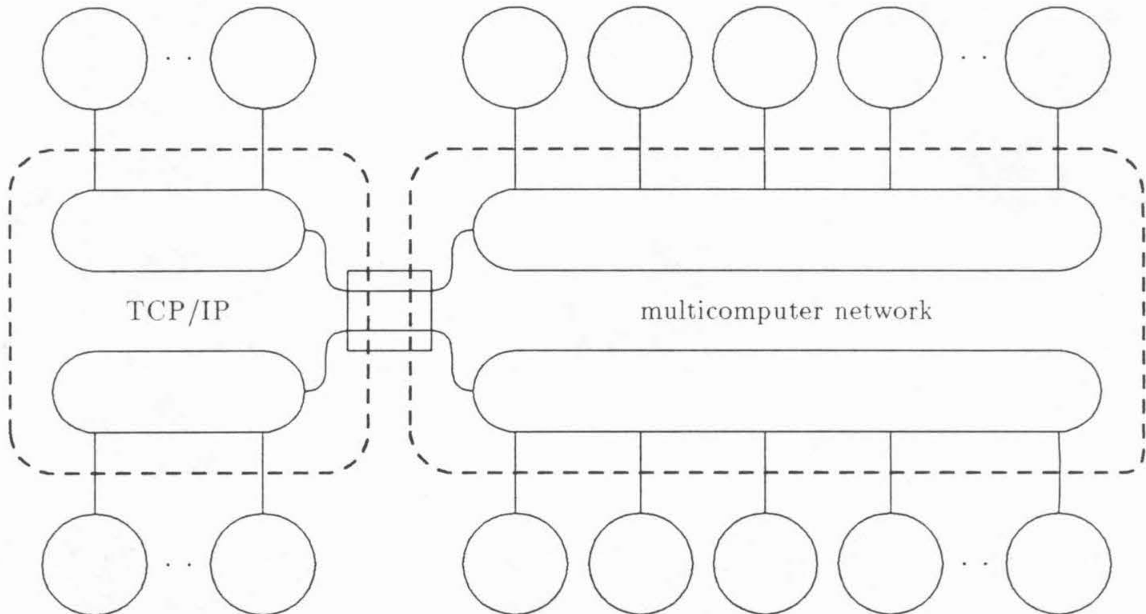

Figure 4.4 A multicomputer shared by two users.

In our implementation, the host system is built on top of the *TCP/IP* network, and the host processes run on any network-connected host that uses the Berkeley UNIX *socket* mechanism. The node system is built on top of the multicomputer network, and may involve either a replacement kernel in each node or a set of emulation routines for the CE functions.

In this particular implementation, the gateway is a single `ifc` process, and each host message system is a single *message-switcher* process. The message switcher is the spoke of the host message system. It is connected to each host process and to the `ifc` process via TCP/IP stream sockets. Message-sending functions in a host process convert CE messages into TCP/IP messages before sending them to the message switcher. Depending on the

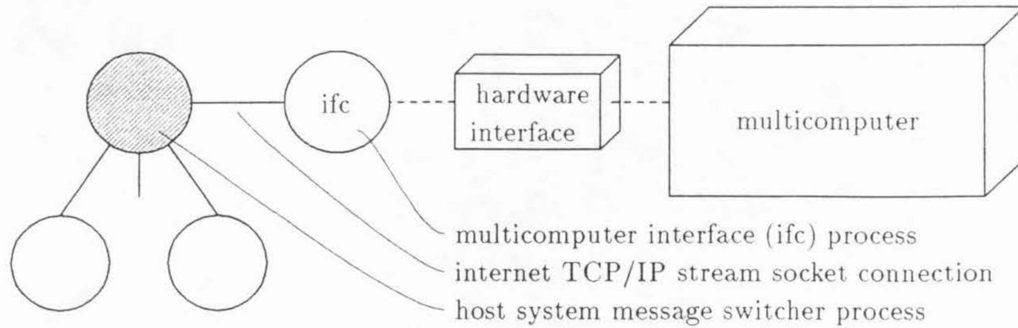

Figure 4.5 Host message-system implementation.

ID of the destination process, the message switcher will send a message either to another host process or to the ifc process. The ifc process waits for messages from both the multicomputer and the switchers. When it gets a message from a switcher, it converts the message into a multicomputer message and sends it to a multicomputer node owned by the user who owns the switcher.

When the ifc process gets a message from the multicomputer, the node ID of the sender is used to determine the destination switcher process. The ifc process then converts the message into a TCP/IP message and sends it to the switcher. When the switcher gets a message from the ifc process, it sends the message to the destination host process. The receive function in the host process then converts the message into a CE message to be returned to the user program.

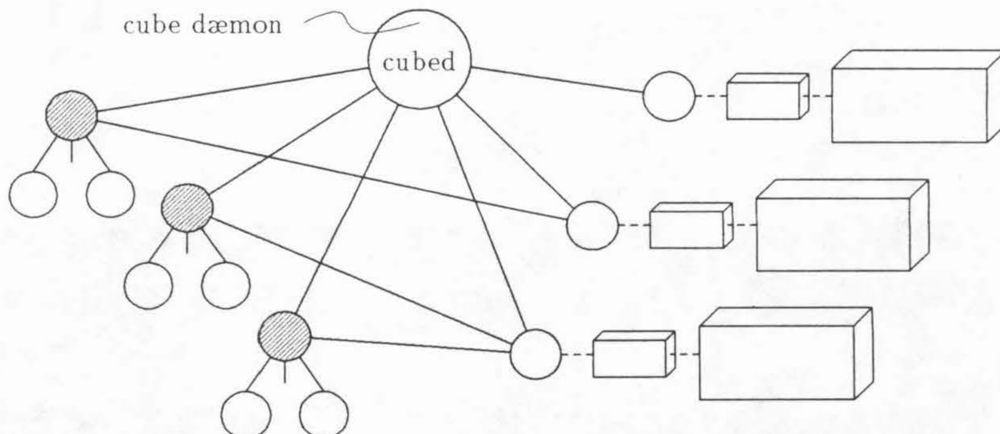

Figure 4.6 Cosmic Environment with unified resource management.

Since we have several multicomputers, and since some of them are of the same type,

we centralize the allocation of all multicomputers in a process called the *cube dæmon*. When a multicomputer is requested by type, the cube dæmon tries to assign an available multicomputer of the required type by searching the list of all multicomputers registered to it. Thus, the user is not concerned with locating an available machine because it makes no difference which one is assigned.

We connect all `ifc` processes and switcher processes with the cube dæmon via TCP/IP stream sockets. These sockets do not carry much traffic; they are merely tokens of participation in CE for the switchers and the `ifc` processes.

4.2.2 Cosmic Environment exterior

Having been spoiled by the convenience of the *Network File System* (NFS) on workstations, the first thing that we decided that we did *not* want to know is where to go to access the multicomputers. Like files in a NFS environment, CE is equally accessible from everywhere in the same network. The cube dæmon resides on a known host in a network, and a configuration file in each participating machine is initialized to contain the network address of the cube dæmon.

Every utility that accesses CE connects to the cube dæmon using the network address found in the configuration file, making CE available and equally accessible from anywhere within the same network. The most frequently used utility is the program called `peek`, which prints the status of CE:

```
CUBE DAEMON version 7.2, up 9 days 20 hours on host ganymede

   {            }  3d cosmic cube, b:0000 [  venus fly trap]  2.3h
   {            }  6d cosmic cube, b:0000 [  ceres TEST    ]  2.3h
   {  sim mikep }  4d ipsc2 cube , b:0000 [ saturn iPSC2   ]  2.1h
   {group david }  7d ipsc cube  , b:0000 [  titan :iPSC d7]  3.4h
   {            } 28n s2010      , b:0000 [ psyche :ginzu  ]  4.9d
   {group apl   }  4n s2010      , b:000b [salieri :ginzu  ]  6.9h
   {            } 48n s2010      , b:0000 [perseus :S2010  ]  4.9d
   {group sharon}  8n s2010      , b:0007 [perseus :S2010  ] 29.2m
   {group tony  }  8n s2010      , b:000c [ mozart :S2010  ]  4.7h
```

The `peek` utility lists all available, occupied, and fragmented multicomputers. In the display above, user `tony` and user `sharon` each occupy 8 nodes in a 64-node S2010 without

interfering with each other. User `apl` is using 4 nodes of a 32-node S2010. User `david` is using a 128-node iPSC/1, and user `mikep` is using a 16-node iPSC/2.

To use a multicomputer, we must first allocate a multicomputer. We specify the multicomputer type, and the cube dæmon picks the best allocation according to an algorithm specific to that type. To allocate a 3-node `s2010`, we can enter "`getcube 3n s2010`." A `peek` will now show the following list:

```
CUBE DAEMON version 7.2, up 9 days 20 hours on host ganymede

    {              }  3d cosmic cube, b:0000 [  venus fly trap]  2.9h
    {              }  6d cosmic cube, b:0000 [  ceres TEST     ]  2.9h
    {  sim mikep   }  4d ipsc2 cube , b:0000 [ saturn iPSC2    ]  2.1h
    {group david   }  7d ipsc cube  , b:0000 [  titan :iPSC d7]  3.4h
    {              } 28n s2010      , b:0000 [ psyche :ginzu   ]  4.9d
    {group apl     }  4n s2010      , b:000b [salieri :ginzu   ]  7.4h
    {              } 45n s2010      , b:0000 [perseus :S2010   ]  4.9d
    {group wen-king}  3n s2010      , b:0007 [neptune :S2010   ] 12.0s
    {group sharon  }  8n s2010      , b:0007 [perseus :S2010   ] 29.2m
    {group tony    }  8n s2010      , b:000c [ mozart :S2010   ]  5.3h

GROUP {group wen-king}  TYPE reactive  IDLE 5.0s

    ( -1  -1)     SERVER   0s   0r   0q    [neptune 18339]  3.0s
    ( -1  -2)   FILE MGR   0s   0r   0q    [neptune 18340]  3.0s
    (--- ---)    CUBEIFC   1s   1r   0q    [perseus 4238 ]  7.0s
```

In this example, the allocation algorithm carves out a 3-node subset from the multicomputer shared by `sharon` and `tony`, instead of from the one used by `apl`. After the allocation, any multicomputer programs that we run on the hosts or on the nodes become part of our process group. The host processes will be connected to our switcher and the node processes will be spawned on our nodes. Host processes are shown in the extended peek display below the main list. In this example, a set of server programs was automatically started and added to the process group when `getcube` returned.

4.2.3 Cosmic Environment processes

While CE is not in use, the only active processes in the hosts are the cube dæmon process and the `ifc` processes. Each `ifc` process resides in a host containing an interface to a multicomputer, and maintains a TCP/IP connection to the cube dæmon process. The cube dæmon keeps track of its set of `ifc` connections; that a connection remains open is an indication that the multicomputer attached to the `ifc` process is ready for use. An `ifc`

process passes the multicomputer status to the cube dæmon via its TCP/IP connection. The cube dæmon process passes allocation and deallocation commands to the `ifc` process via the same connection.

When a user requests a multicomputer by running the getcube program, the getcube process connects to the cube dæmon and sends it a set of allocation requirements. If the requirements can be fulfilled, the requested multicomputer or a partition of the multicomputer is marked as allocated in cube dæmon's table. An allocation command is then sent to the corresponding `ifc` process. The `ifc` process initializes nodes allocated to the user and then connects to the user's getcube process. The getcube process then fades to background to become the switcher process, giving the user the appearance that the getcube command has terminated as an indication that the allocation has completed.

A set of service processes is started by the getcube process as it fades to background. These processes are responsible for such mundane tasks as the details of process spawning, file access, and printing of error messages. Additional host processes and utilities are run by the user to perform computation.

Porting CE to another multicomputer involves the creation of a new plug-in node system for the new multicomputer. We have a choice of implementing the CE node system on top of the native node kernel or writing a new kernel that implements the CE node system. The Cosmic Cube and the S2010 both have the CE node system as their native system. We replaced the iPSC/2 kernel with a custom kernel. On the iPSC/1 and on earlier versions of the iPSC/2, the CE node system is layered on top of their native systems — the `NX` kernels.

When we layer a CE node system on top of the native node kernel, the `ifc` process is linked with the native host library for the multicomputer, and it interacts with the multicomputer via the native message functions. To the native system running underneath, the `ifc` process appears to be just an ordinary host process of the native system. The CE node system can operate within the confines of user-accessible functions of the native

system because it has simple requirements; it does not need special capabilities from the native system and it does not interfere with the functioning of the native system.

4.2.4 Program compilation

Different commercial multicomputers will invariably provide dissimilar methods of compiling programs for their multicomputers. The compiler options are different; those with the same name may have different meanings to different compilers and some that are available to one compiler may be missing for another compiler. The sequence of operations that the user has to go through may be different, and the set of end products may also be different. However, we recognize that only a small set of the options is useful, and we can easily hide any difference among the compilers by the use of a program that runs programs. By declaring that only a limited set of commonly used compiler flags are supported, the compilation tools for all machines can be described in one table:

	host	ghost	cosmic	iPSC/1	iPSC/2	S2010
compiler	cch	ccgh	cccos	ccipsc	ccipsc2	ccs2010
linkable-file suffix	.o	.gh.o	.086	.o286	.o386	.s2010.o
runnable-file suffix		.gh	.cos	.ipsc	.ipsc2	.s2010
archiver	arh	argh	arcos	aripsc	aripsc2	ars2010
archive-file suffix	.a	.gh.a	.A86	.a286	.a386	.s2010.a

The following sequence will compile the program myprogram.c for all of these machines, and the runnable object code generated will be named myprogram, myprogram.gh, myprogram.cos, myprogram.ipsc, myprogram.ipsc2, and myprogram.s2010, respectively.

```
% cch     -o myprogram myprogram.c -lcube
% ccgh    -o myprogram myprogram.c -lcube
% cccos   -o myprogram myprogram.c -lcube
% ccipsc  -o myprogram myprogram.c -lcube
% ccipsc2 -o myprogram myprogram.c -lcube
% ccs2010 -o myprogram myprogram.c -lcube
```

To illustrate the amount of complexity hiding that can be performed, actual compilation for the iPSC/1 can be done only on the controller box of the iPSC/1 — the Intel 286/310. The program ccipsc copies the source files to the 286/310 for compilation, and copies back

compiled object files when compilation is completed. It creates an illusion that compilation takes place where the `ccipsc` command is issued.

4.2.5 Spawning programs

Like compilers, different multicomputers supply their own method of running a node program. We can hide the differences by using programs that run other programs; but, unlike the compilers, we no longer have to differentiate one multicomputer from another by giving them different names. While a compiler can be invoked by the user at any time, a program loader can be invoked only when the user has an active process group.

We can therefore eliminate another level of complexity by having the generic loader, `spawn`, check the type of the multicomputer being used and have it run the loader command specific to that multicomputer. Thus, to load the program generated in the previous example into any of the multicomputers, we can run "`spawn myprogram,`" regardless of the multicomputer we are using.

Utilities such as the node-program compilers are called machine-specific utilities; utilities such as `spawn` are called machine-dependent utilities; and utilities such as `peek` are called machine-independent utilities. The node system for each type of multicomputer, therefore, contains the `ifc` process, the machine-specific utilities, the machine-dependent utilities, and the compiler libraries.

4.2.6 Data representation and conversion

We have tried to simplify CE and, at the same time, to hide the differences between different multicomputers; but, it is not always possible to do both. The difference in data representation among processors of different multicomputers and hosts is one that we cannot hide in vanilla C. When two communicating processes are run on two machines having different data representations, data in messages sent from one process to another need to have their representations converted before they can be used. We can always move the conversion problem into the compiler, but we still have to decide how the problem is to be solved.

```
68020: 01000000 00001001 00100001 11111011 01010100 01000100 00101101 00011000
  vax: 01001001 01000001 11011010 00001111 00100001 10100010 11000010 01101000
80286: 00011000 00101101 01000100 01010100 11111011 00100001 00001001 01000000
```

Listing 4.1 Three representations of π in double-precision floating-point-number format.

Data-representation problems have been a subject of study ever since computers were first connected by networks. The most common solution is to define an interchange data representation. The sender converts data items in its outgoing messages from the sender's representation to the interchange representation; the receiver converts data items in its incoming messages from the interchange representation to the receiver's representation. A set of conversion routines with the same name but having different functions on different machines is provided to make programs portable. A program needs only to be capable of converting its data to and from the interchange representation, rather than to and from all possible representations.

In the case of a multicomputer, however, message traffic is usually much higher and message latency is usually much lower between the nodes than between the hosts. Having to convert the data in each internode message to and from an interchange representation can significantly reduce the performance of message-intensive applications unless the interchange representation happens to be identical to the representation of the multicomputer.

Our solution is therefore to make the interchange representation adjustable; we define the interchange representation for a process group to be the representation used by the multicomputer of the process group. Node processes are not required to convert the data in their messages, and, if they do, the functions that they call to perform the conversion will have no effect. A host process is required to convert message data to the interchange representation before it sends a message, and from the interchange representation after it receives a message. Host processes already have a large per-message overhead, and they can absorb the extra work of converting the data.

The node programs never need any conversion routines, but host programs must carry routines that convert data representations to and from those of all multicomputers that CE supports. The conversion routines check the multicomputer type before deciding how data

is to be converted. Adding a new multicomputer to CE may require that host programs be recompiled if the data format for the multicomputer is not already supported.

In order to preserve the CE specifications, conversions are done in place, because message buffers are treated like memory buffers from `malloc`. Having to convert a message and put the converted data in another buffer weakens the specification. In order to have such conversion make sense, however, the location and the size of each data item in the messages must be the same for all processes. However, different machines do have different sizes and alignment rules for the same data type.

```
struct test { char  AA[3];
              short BB   ;
              long  CC   ;
              int   DD   ; } ;

68020: AAA*BBCCCCDDDD
  vax: AAA*BB**CCCCDDDD
80286: AAA*BBCCCCDD
```

Listing 4.2 Three layouts of a structure, in order of increasing byte address.

For data sizes, we made the decision that in all the machines that we support data items will have the following sizes, and a message should include only the following data types:

double-precision floating-point number	64 bits
single-precision floating-point number	32 bits
long integer	32 bits
short integer	16 bits
character	8 bits

For alignment, we add any necessary padding to force each data item to align on its strictest alignment boundary: A k-byte data type should be aligned on a k-byte boundary. The bottom of a data structure should also be rounded out by padding it to the alignment boundary of the largest data item in the structure. Whenever possible, a structure should be rearranged to minimize the amount of padding necessary.

When data items are aligned using these rules, the location of each data item in a message is the same for all machines. A set of conversion routines can be used to perform in place conversion on the items:

```
htocs(p,n)   ctohs(p,n)   Convert short integers.
htocl(p,n)   ctohl(p,n)   Convert long integers.
htocf(p,n)   ctohf(p,n)   Convert single-precision floating-point numbers.
htocd(p,n)   ctohd(p,n)   Convert double-precision floating-point numbers.
```

The `htoc` set of functions converts data from the format used by the calling process to the interchange format. The `ctoh` set of functions performs the reverse conversion. Parameter `p` is a pointer to an item of the appropriate type and parameter `n` is the number of consecutive data items to be converted by the functions. There is no conversion routine for the character type because the basic units of the messages are bytes and their correct ordering is enforced by the ifc process.

The data representation problem may require rethinking after machines with a 64-bit data bus become available. Data-type conversion is only an inconvenience, and it can always be taken care of by writing a new compiler that inserts code to do the conversion for the user. However, such is beyond the scope of this research.

Chapter 5 Model of Simulation

Section 5.1 Mathematical Framework and Analysis

5.1.1 Systems and elements

A *system* consists of a *system body*, a set of *system inputs*, and a set of *system outputs*. It is a "*black box*" whose only external connections are the inputs and outputs. In a representation of a simulator, each individual output conveys an *atomic property* of the simulated system. A property is atomic if at any point during the simulation the simulator contains all information about that property up to some simulated time, but none beyond that simulated time.

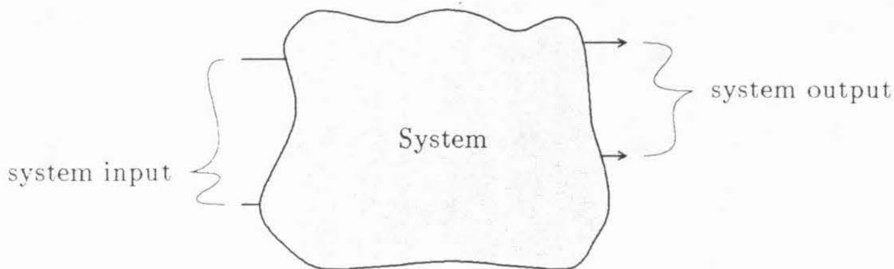

Figure 5.1 Representation of a system.

A system can be defined recursively as a collection of systems linked together by *arcs;* each arc connects an output of its source system to an input of its destination system, and each arc represents the source system's direct influence on the destination system. The recursion terminates with systems that are called *elements;* the behavior of each element is defined algorithmically to correspond to a model of some physical device or process.

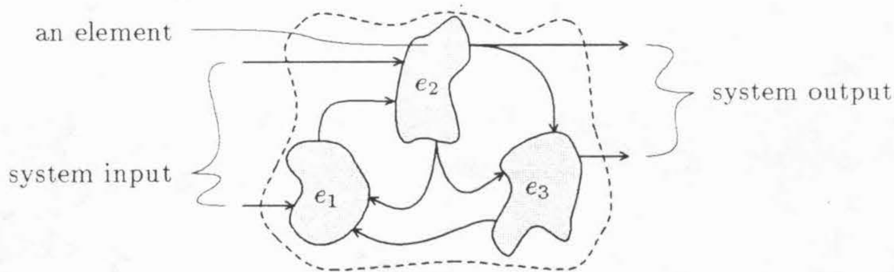

Figure 5.2 Representation of a system composed of elements.

If the hierarchy that is induced by this recursive definition is *flattened* by expanding each system recursively into its constituent systems and elements, we obtain a system that

is composed entirely of elements. In order to simplify the following exposition, we shall, without loss of generality, discuss a system that is composed entirely of elements.

In a *composite system*, each element input can be connected to no more than one arc, whereas each element output can be connected to any number of arcs. The set of system inputs is the set of unconnected element inputs; whereas the set of system outputs can be any subset of the element outputs. Systems without any inputs are called closed systems. In order to simplify the mathematical framework, we shall close each system with an environment element, e_ϵ, that provides inputs to all unconnected system inputs and accepts outputs from all unconnected system outputs.

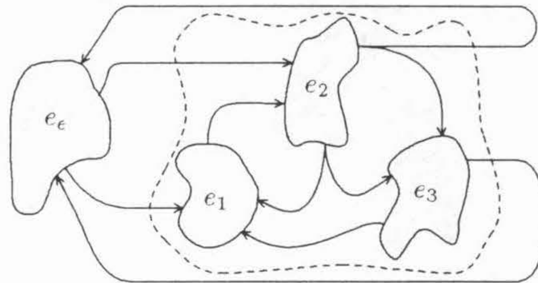

Figure 5.3 Closing a system into a closed graph.

The representation is now a graph that can be described as below:

E : The set of elements in a system.

A : The set of arcs in a system.

U : $::= E \cup \{e_\epsilon\}$

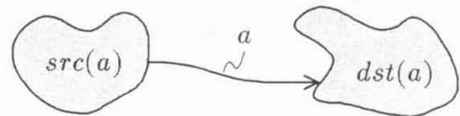

Figure 5.4 Arc source and destination.

$inp(e)$: The set of all arcs terminating at e.

$out(e)$: The set of all arcs originating from e.

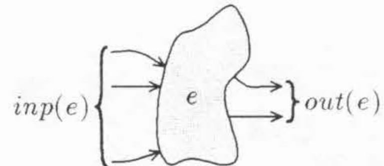

Figure 5.5 Element inputs and outputs.

$src(a)$: The source element of a.

$dst(a)$: The destination element of a.

path: A path of length n is a sequence of arcs, $(a_0, a_1, a_2, ..., a_{n-1})$, such that $dst(a_i) = src(a_{i+1})$ for $0 \leq i < n-1$.

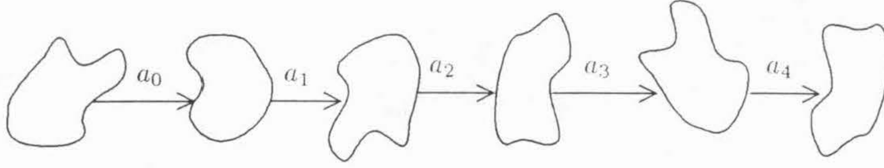

Figure 5.6 Arcs a_{0-4} form a path of length 5.

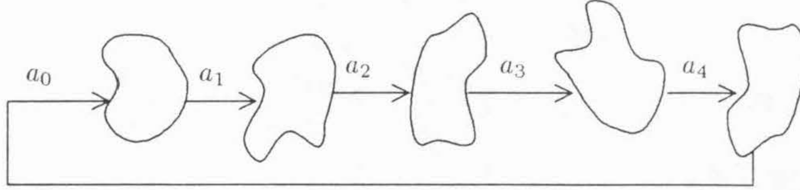

Figure 5.7 Arcs a_{0-4} form a circuit of length 5.

circuit: A circuit of length n is a path of length n in which $src(a_0) = dst(a_{n-1})$.

5.1.2 States and time

The state of a system includes both its internal state and the state of its outputs. Let $S_U(t_0, t_1)$ be the state description of the closed system between the time t_0 and t_1, $t_0 \leq t_1$, and let $S_L(t_0, t_1)$ be the state description restricted to the subset or member, L. The state of the closed system can be written as a Cartesian product of the environment state and the system state:

$$S_U(t_0, t_1) = S_{e_e}(t_0, t_1) \times S_E(t_0, t_1)$$

Similarly, the system state can be written as the Cartesian product of the element states:

$$S_E(t_0, t_1) = S_{e_1}(t_0, t_1) \times S_{e_2}(t_0, t_1) \; S_{e_3}(t_0, t_1) \times \ldots \times S_{e_n}(t_0, t_1)$$

A simulator is said to be *progressive* if it can compute the following function for any valid input description, $S_{inp(E)}(t_0, t_1)$, which is a description of input state over a time interval, and any valid initial state of the system, $S_E(t_0, t_0)$.

$$S_{inp(E)}(t_0, t_1), S_E(t_0, t_0) \longmapsto S_E(t_0, t_1)$$

A simulator may be able to compute more state information for some of its outputs than is specified above. For example, if the system can compute the following function for some $\delta \geq 0$, the output o is said to have a delay of no less than δ at time t_1.

$$S_{inp(E)}(t_0, t_1), S_E(t_0, t_0) \longmapsto S_o(t_0, t_1 + \delta)$$

If δ is the largest value for the above to remain true, then δ is the delay of the output o at simulated time t_1. The delay of a system at simulated time t_1 is defined to be the smallest of all output delays of the system at t_1. The definition of a progressive simulator precludes the possibility of negative delays.

5.1.3 Knots and progress

In this section, we shall define a set of rules that allows us to recursively construct progressive system simulators by connecting progressive element simulators in the same manner in which the elements of the system are connected. We shall call such a simulator a *composite simulator*. In order to discuss progress, we make a minimal assumption that information computed at any element simulator, e, will be available to all $dst(out(e))$. We shall assume for the moment that elements are deterministic; that is, $S_{inp(e)}(t_0, t_1)$ and $S_e(t_0, t_0)$ completely determine $S_e(t_0, t_1)$. Thus, in order to determine whether a simulator is progressive, we need to consider only the arc state, $S_A(t_0, t_1)$.

A simulator lacks progress if and only if there exists a combination of $S_{inp(E)}(t_0, t_1)$ and $S_E(t_0, t_0)$ such that the simulator fails to compute $S_a(t_0, t_1)$ for some $a \in A$. Let t_K be the time value, $t_0 \leq t_K < t_1$, such that the simulator can compute $S_A(t_0, t_K)$ but not $S_A(t_0, t_K^+)$. Let $K \subseteq A$ be the set of arcs such that the simulator can compute $S_a(t_0, t_K)$ but not $S_a(t_0, t_K^+)$. The set K is called a *knot* in the simulation. The presence of a knot is synonymous with a lack of progress.

Knot: Simulator can compute $S_a(t_0, t_K^+)$ for all $a \notin K$.

Simulator can compute only $S_a(t_0, t_K)$ for all $a \in K$.

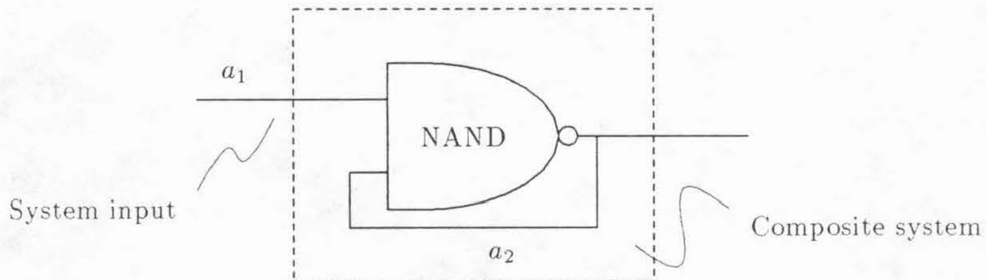

Figure 5.8 Example of a knot-containing system.

An example of a knot-containing system is a zero-delay NAND-gate with one of its inputs connected to its output, as shown in Figure 5.8. Although the element simulator for the NAND-gate may be progressive, the composite simulator for this system cannot be. For example, if the input to the system is the following:

$$S_{inp(E)}(0,2) = \begin{cases} 0 & \text{for } 0 \le t < 1; \\ 1 & \text{for } 1 \le t \le 2. \end{cases}$$

then the composite simulator can compute only the following for the arc a_2:

$$S_{a_2}(0,2) = \begin{cases} 1 & \text{for } 0 \le t < 1; \\ ? & \text{for } 1 \le t \le 2. \end{cases}$$

The simulator cannot compute S_{a_2} for $1 \le t \le 2$ because a self-consistent state assignment for a_2 cannot be found. The set of arcs $\{a_2\}$ is a knot.

Theorem 5.1: If a is an arc of knot K, then the following conditions hold:

a. $inp(src(a))$ is not empty; *ie*, $src(a)$ is not a source node in the directed graph of elements.

b. The delay of $src(a)$ at t_K is 0.

c. Some member of $inp(src(a))$ is also a member of K.

Proof:

a. If the set of arcs, $inp(src(a))$, is empty, then $src(a)$ is a closed system. A closed system does not need any information from its environment in order to compute its state — it is able to compute its outputs up to any arbitrary time. Therefore, $inp(src(a))$ cannot be empty.

b. By the definition of a knot, the simulator can compute up to t_K for all arcs in $inp(src(a))$. If the delay for $src(a)$ is greater than zero, the simulator would be able to compute up to t_K^+ for a. Since it cannot, by definition, the delay of $src(a)$ must be zero.

c. If no member of $inp(src(a))$ is in K, then, by the definition of a knot, the simulator should be able to compute up to t_K^+ for all members of $inp(src(a))$. Furthermore, since delay cannot be negative, the simulator

should be able to compute up to t_K^+ for a. Therefore, if a is in K, some member of $inp(src(a))$ must also be a member of K.

5.1.4 Rules of thumb — sufficient conditions for progress

Corollary 5.2: Every knot contains a circuit.

Proof: There is a finite number of arcs in a system. If for every arc, $a_i \in K$ there is at least one arc, $a_j \in K$, such that $a_j \in inp(src(a_i))$, then there must be a circuit in K.

Corollary 5.3: If the system contains no circuits, then the composite simulator is progressive.

Proof: Since every knot must contain a circuit, a system that does not contain any circuits cannot have knots.

Corollary 5.4: If every element has a delay greater than 0, then the composite simulator is progressive.

Proof: Follows directly from Theorem 5.1, part b.

Corollary 5.5: If in every circuit there is some element with non-zero delay, then the simulator is progressive.

Proof: From Corollary 5.2, if K exists, it must contain a circuit. From Theorem 5.1, if such a circuit exists, all the elements in it must have zero delay. Therefore, if all circuits have at least one element with non-zero delay, then K cannot exist.

Although the progress conditions stated in Corollaries 5.3, 5.4, and 5.5 identify a set of systems with progressive simulators, they do not identify, either by themselves or all together, the set of all systems with progressive simulators. These are not minimal conditions, because there are systems with progressive simulators that do not satisfy any of the

three corollaries. The corollaries are useful as simple rules of thumb because there exists an effective procedure for testing each of them.

5.1.5 Non-existence of necessary and sufficient progress conditions

5.1.5.1 Simulation and Boolean satisfiability

An algorithm that tests for a necessary and sufficient condition, if any such condition does exist, must be NP-hard. Figure 5.9 shows a system that tests for the satisfiability condition in a set of Boolean clauses. The system contains a zero-delay NAND gate, a counter, a clock source, and a network of zero-delay gates forming the clauses. A simulator for the system is not progressive if and only if there exists a counter output such that all of the clauses are true. If there is an algorithm that can determine whether a simulator for any system of this form has progress, we can use it to determine whether any collection of clauses can all be true at the same time. Since the latter operation (Boolean satisfiability [17]) is known to be NP-complete, the algorithm must be NP-hard. Therefore, any generic algorithm that tests for a necessary and sufficient condition must be NP-hard.

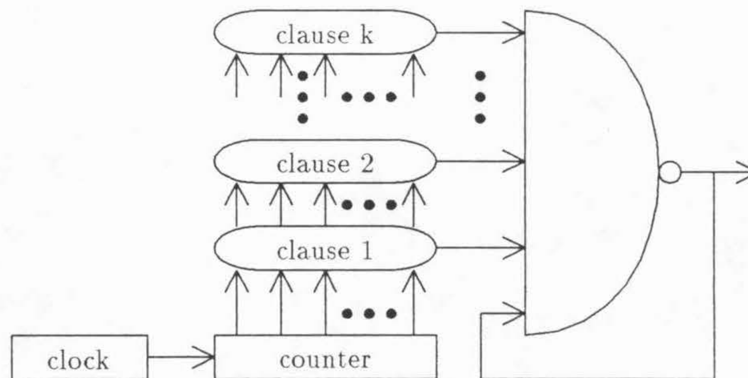

Figure 5.9 A circuit to evaluate satisfiability of a set of clauses.

5.1.5.2 Simulation and simultaneous equations

Another way to demonstrate the futility of searching for a necessary and sufficient condition is to examine the relationship between simulation and simultaneous equations. We define a progressive simulator to be one that can compute the following function for any valid input description, $S_{inp(E)}(t_0, t_1)$, and any valid initial state: $S_E(t_0, t_0)$.

$$S_{inp(E)}(t_0,t_1), S_E(t_0,t_0) \longmapsto S_E(t_0,t_1)$$

Let H_e be the mapping associated with a progressive simulator for the element e; we can express a composite simulator as the following set of equations:

$$\forall_{e} \in E \quad S_e(t_0,t_1) = H_e(S_e(t_0,t_0),\ S_{inp(e)}(t_0,t_1))$$

Since $S_e(t_0,t_1)$ describes $S_{out(e)}(t_0,t_1)$, and since $S_A(t_0,t_1)$ and $S_E(t_0,t_0)$ determine $S_E(t_0,t_1)$, a composite simulator can also be expressed as the following set of equations:

$$\forall_{a} \in A \quad S_a(t_0,t_1) = G_a(S_{src(a)}(t_0,t_0),\ S_{inp(src(a))}(t_0,t_1))$$

G_a is $H_{src(a)}$ restricted to the arc a. These are simultaneous equations in the form:

$$\forall_i : \quad X_i = F_i(X_1, X_2, ..., X_n)$$

Furthermore, any set of simultaneous equations can be transformed into a physical system for which a composite simulator can be constructed. The set of all simulators and the set of all simultaneous equations must be equivalent.

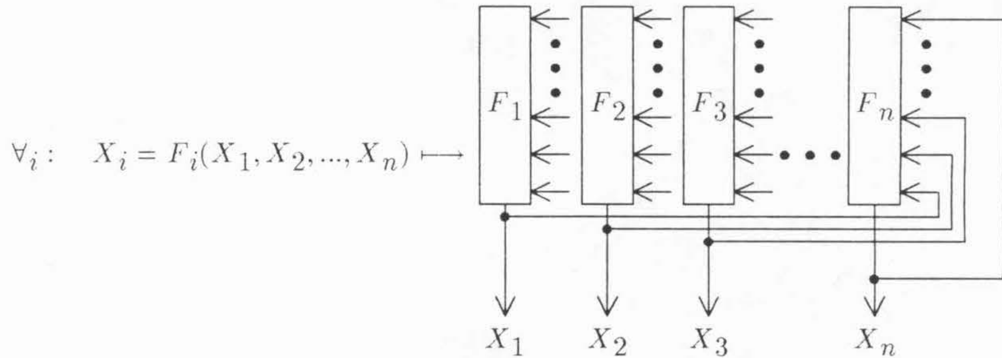

Figure 5.10 Mapping equations into physical system.

In any set of simultaneous equations, only one of the three possibilities listed below can exist.

1. The simultaneous equations have no solution.
2. The simultaneous equations have exactly one solution.
3. The simultaneous equations have more than one solution.

Since a simulation is progressive if and only if its set of simultaneous equations has a solution, any test for determining progress of a simulator can be used as a test for determining the existence of solutions for simultaneous equations, and vice versa. Since the test

for the latter has not been found, the test for the former also has not been found. The search for a necessary and sufficient condition is, therefore, both difficult and, so far, futile.

Section 5.2 Operational Framework

Although an effective simultaneous-equation solver for the general case does not exist, the simultaneous-equation representation brings us one step closer to an operational model, because effective procedures — such as Gaussian elimination for ordinary linear equations — exist for specific classes of equations.

The equations for a simulation are generally difficult to analyze because its variables and constants describe states over the entire simulation interval, and the equations themselves can be arbitrarily complex. We may be able to obtain a set of simpler equations, however, if we restrict the analysis to those simulations that span only a short interval. If the interval of a simulation can be broken down into a finite number of smaller intervals such that each interval can be computed by an effective procedure, we will have found an effective procedure for the simulation.

5.2.1 Breaking a simulation into smaller slices

Any equation whose associated output has a delay δ, such that $\delta \geq \sigma$, can be reduced to a constant equation by restricting the simulation to an interval equal to σ. Let L be the set of output arcs with a non-zero delay at time t. Suppose L is non-empty, let σ be the smallest non-zero delay. The state of all arcs between t and $t+\sigma$ are related by the following set of simultaneous equations (justifications to follow shortly):

$$\forall_{a \in A}\; S_a(t, t+\sigma) = \begin{cases} G_a(S_{src(a)}(t,t) \qquad\qquad\qquad\qquad\qquad) & \text{if } a \in L; \\ G_a(S_{src(a)}(t,t),\; S_{inp(src(a))}(t, t+\sigma)) & \text{if } a \notin L. \end{cases}$$

If equations like these can be solved, simulation for a system can be performed by dividing the simulation interval into σ-wide slices, and repeatly solving for $S_A(t, t+\sigma)$, computing $S_E(t, t+\sigma)$, and advancing to time $t+\sigma$. Since the set of equations above covers a slice of time, let us simply refer to it as a *slice*. The operation of a composite simulator that advances one slice at a time can be described by the actions of its element simulators.

Figure 5.11 depicts the sequence of actions taken by the simulator for element, e, whose output arc, a, has a non-zero delay of δ. At the beginning of the slice that starts at t (Figure 5.11(a)), the simulator has progressed to t and has computed $S_e(t,t)$. Since the delay for a is δ, $S_e(t,t)$ contains the output state description: $S_a(t, t+\delta)$.

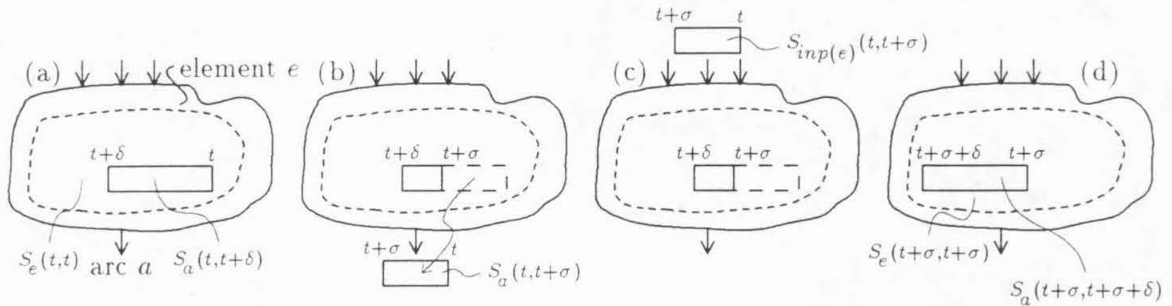

Figure 5.11 Element-simulator operation for an element with a non-zero delay.

Since σ is no larger than δ, the equation for S_a does not depend on the state of other arcs, and the simulator can output the state description, $S_a(t, t+\sigma)$, (Figure 5.11(b)) without any additional inputs. If the state description over the interval $(t, t+\sigma)$ can be computed for every arc in the system, $S_{inp(e)}(t, t+\sigma)$ will become available to e (Figure 5.11(c)). Since element simulators are assumed to be progressive, the simulator for e will compute $S_e(t, t+\sigma)$ from $S_e(t,t)$ and $S_{inp(e)}(t, t+\sigma)$, and will be ready for the next slice (Figure 5.11(d)).

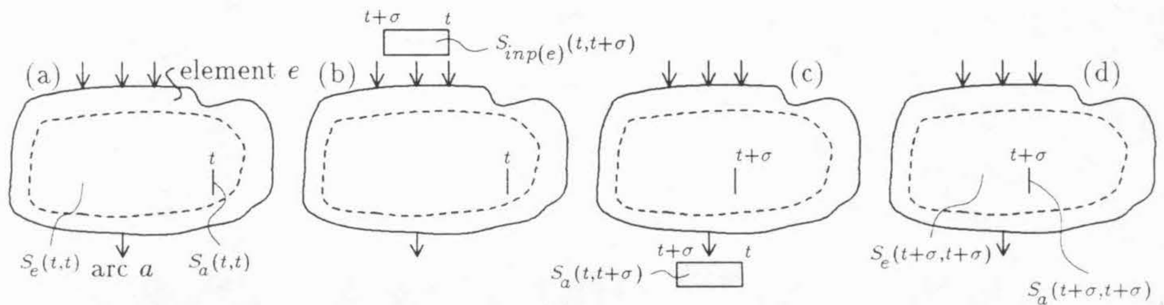

Figure 5.12 Element-simulator operation for an element with a zero delay.

If the delay is zero (Figure 5.12), the simulator for e does not contain any output state description beyond the starting time of the slice (Figure 5.12(a)). The equation for S_a depends on the state of other arcs, and the simulator is unable to produce $S_a(t, t+\sigma)$ until it has received $S_{inp(e)}(t, t+\sigma)$ (Figure 5.12(b)). If e is not a member of a zero-delay circuit

(Corollary 5.5), $S_{inp(e)}(t, t+\sigma)$ will eventually be available. When $S_a(t, t+\sigma)$ is computed, the simulator will be ready for the next slice.

A slice that does not contain zero-delay circuits can be solved by simple variable substitution; a slice that contains zero-delay circuits (called an *obligatory slice*) requires simultaneous equation solving. A system has a progressive simulator if and only if a solution exists for every slice of a system. If a slice has no solutions, then the slice contains a knot.

5.2.2 Slices and knots

For a system that contains only deterministic elements, a non-obligatory slice always has exactly one solution. An obligatory slice, however, can have three possible outcomes: no solution, one solution, and multiple solutions. All three of the outcomes can be found in the cross-coupled zero-delay XOR-NOR circuit in Figure 5.13.

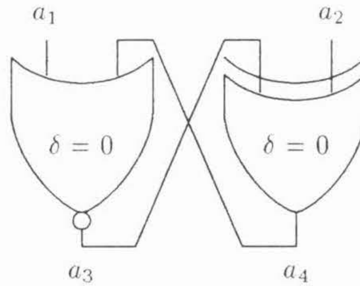

Figure 5.13 A system that contains all three types of slices.

$$
\begin{aligned}
S_{a_1}(t, t+\sigma) &= \text{A function of the environment.} \\
S_{a_2}(t, t+\sigma) &= \text{A function of the environment.} \\
S_{a_3}(t, t+\sigma) &= \neg(S_{a_4}(t, t+\sigma) \vee S_{a_1}(t, t+\sigma)) \\
S_{a_4}(t, t+\sigma) &= \quad S_{a_3}(t, t+\sigma) \oplus S_{a_2}(t, t+\sigma)
\end{aligned}
$$

When the inputs a_1 and a_2 are both 0 over the $(t, t+\sigma)$ interval, the set of simultaneous equations for the circuit can be reduced to the set of two equations below, which has no solution:

$$
\begin{aligned}
S_{a_3}(t, t+\sigma) &= \neg(S_{a_4}(t, t+\sigma)) \\
S_{a_4}(t, t+\sigma) &= \quad S_{a_3}(t, t+\sigma)
\end{aligned}
$$

The slice is a *no-solution slice.* A no-solution slice contains a knot, and no simulator is able to complete the simulation when a no-solution slice is encountered. When a_1 is 0 and a_2 is 1, the set of simultaneous equations for the circuit can be reduced to the set of two equations below, which has arbitrarily many solutions.

$$S_{a_3}(t, t+\sigma) = \neg(S_{a_4}(t, t+\sigma))$$
$$S_{a_4}(t, t+\sigma) = \neg(S_{a_3}(t, t+\sigma))$$

The value of the pair (a_3, a_4) can be either $(1, 0)$ or $(0, 1)$, but their value can switch between the two, spontaneously and for arbitrarily many times. The slice is a *multiple-solution slice.* A simulator can make progress if it is able to continue using one of the solutions. When both a_1 and a_2 are 1, the only solution for the simultaneous equations is $a_3 = 0$ and $a_4 = 1$. The slice is a *single-solution slice.*

5.2.3 Implementation considerations

Thus far, we have analyzed the composite simulator using only abstract models, because any real simulator is bounded by these frameworks. We can never find progressive simulators for more systems than those indicated by these frameworks. We can derive a number of simulators directly from the framework, but in order for any implementation to cover the range indicated by the framework, it must satisfy the following two conditions:

Eventual Delivery: The simulator must make available any information that is present in the simulation to any element that requires it for further computation.

Slice Resolution: The simulator must have mechanisms to resolve any obligatory slice that has a solution.

Simulators that satisfy both of these properties are called *complete simulators.* Not all simulators are, or need to be, complete simulators. For example, if every element in a system has a non-zero delay, slice resolution is not necessary. Complete simulators that operate on all possible systems are beyond our goal; we often restrict ourselves to specific subjects such as discrete-event simulation. We will temporarily restrict ourselves to systems that do

not require slice resolution. in order to allow for the development of a working simulator model.

Section 5.3 The Generic Simulator Model and Its Derivatives

Since it is sufficient to synchronize the elements through their inputs and outputs. strict synchronization of all elements on slice boundaries is unnecessary; elements should be allowed to progress at their own pace as their input data becomes available. Furthermore, if δ for an element is larger than σ, the element does not have to stop producing output at $t + \sigma$, because it already has computed $S_{out(e)}(t, t + \delta)$.

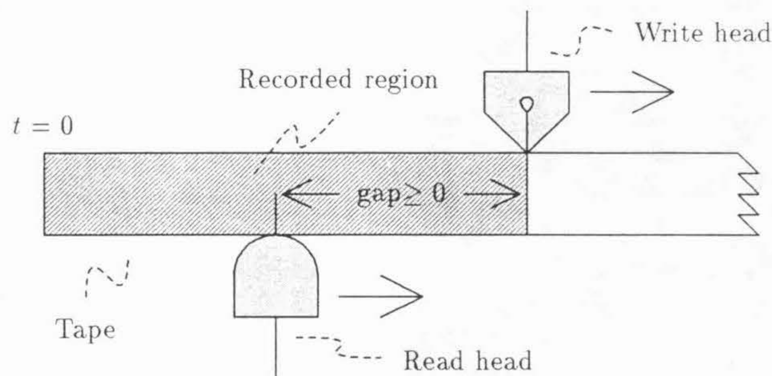

Figure 5.14 Representation of an arc.

If we ignore the existence of obligatory slices, we can construct a generic simulator model using a set of multi-tape automata. We replace each arc in the system with a read head, a write head, and a tape, such that:

1. As information is produced by the originator of the arc, the information and the simulation time are recorded along the length of tape as the write head advances. The recorded time strictly increases.
2. The read head recovers the recorded information and the time from the tape as it advances.
3. Both tape heads move in one direction only, but the read head will never move past the write head.

Since information over periods of time is written onto the tape by its source element before being read from the tape by the destination element, element simulators are decoupled

in simulated time. The gap between a write head and a read head on the same tape is called the *slack*. Since the element simulators are moved forward by consuming and producing slack, this simulator model is called the *slack-driven* simulator model.

A slack-driven simulator is not a complete simulator because the model does not include a mechanism to solve simultaneous equations; when a system encounters an obligatory slice and equation-solving is required, the element simulators involved will stop. They are blocked while waiting for each other to produce more tape; this condition is called *deadlock*. We will describe, in brief, a few derivatives of the slack-driven simulator, some of which are more permissive and some more restrictive; thus, some are more complete and some are less complete than the slack-driven simulator.

5.3.1 Message-driven simulation

A slack-driven simulator can be expressed as a set of concurrent message-passing processes in which the processes are the element simulators and the message streams are the tapes. Whenever a stretch of tape is written by the slack-driven simulator, the information on the tape is sent in a message; whenever a stretch of tape is read, the information in a received message is read. Since the slack is represented by messages queued in transit, a message-passing implementation of a slack-driven simulator is called a *message-driven simulator*.

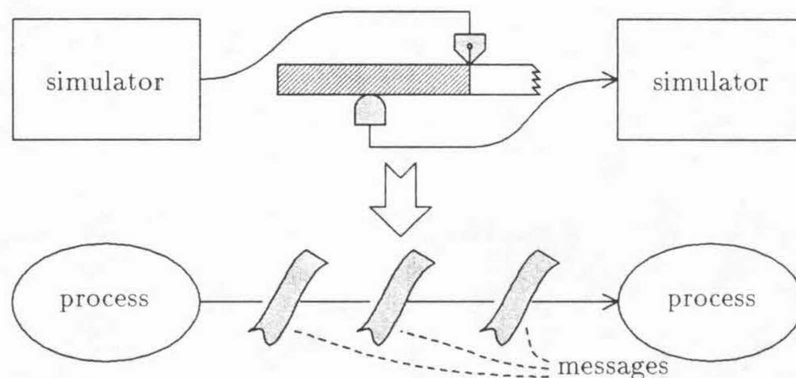

Figure 5.15 Replacing tape by messages.

Since a message-driven simulator is an exact implementation of a slack-driven simulator, the simulation will not make any further progress when equation-solving is required.

5.3.2 Concurrent event-driven simulation

The slack-driven simulator satisfies eventual delivery because each stretch of tape written is immediately available to the destination process. The message-driven simulator duplicates that property by immediately packing and sending the output information as a message, oblivious to the value of the information content of the message. An event-driven simulation is a modified message-driven simulation in which message traffic is reduced by classifying messages and by treating different types of messages differently.

Messages are classified by whether they are needed at the receiving end. Messages that are considered to be non-essential are held back with the objective of combining as many non-essential messages as possible with the next essential message, and packaging them all in a single entity. The total volume of messages in the simulation is reduced without impeding the progress of the simulation. Whether a message is needed, however, depends on the state of the simulation, and is often impossible to determine on the basis of local information alone.

In event-driven systems, however, messages containing state transitions are more likely to be needed than those that do not; most event-driven simulators make the *classification* on that basis alone. Since the transitions are often called events, and since there is generally one in each message for such a simulator, these simulators are called event-driven simulators. Messages containing no events are called *null messages*. Event-driven simulators were first explored by Chandy, Misra, and Bryant [13, 12], though their derivation paths are different from ours. This exposition illustrates that null messages are a consequence of applying a more general model to a specific class of subjects, rather than a necessity when going from a sequential simulator to a distributed simulator.

Culling null messages, as is true with many other methods for reducing message volume, violates the rule of eventual delivery because the rules that decide whether a message is needed at the receiving end can fail. Without additional mechanisms to assure eventual delivery of necessary null messages, deadlock may still occur. A ring of elements with

stable values for their cyclic outputs will fail to produce progress because each element is waiting for its preceding element to produce a message, yet none will arrive if they send only messages containing transitions.

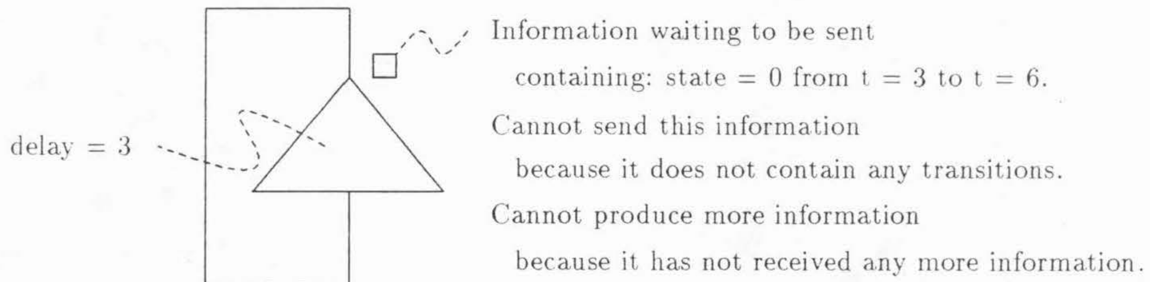

Figure 5.16 Example of deadlock in an event-driven simulation.

5.3.3 Sequential simulator

A sequential simulator is a simple example of a backtracking simulator for event-driven systems. If we describe it in the context of our model, a sequential simulator keeps all of its read heads aligned during the simulation. (All read heads are initially aligned at $t = 0$ at the start of the simulation.) Each write head records not only the output state derived from the element input, but also the expected output state, assuming that the element will encounter no further input change.

If there are currently no state transitions recorded under the read heads, the sequential simulator is free to move the read heads forward without delivering any of the state descriptions to any elements. The state description on the portion of the tapes covered by the motion were produced on the assumption that no transition has occurred over that period, and the assumption was shown to be valid. When a transition is encountered, the assumption by its destination element is shown to be false and the transition is delivered to its destination element so that a new output can be computed. Since the delay of an element must not be negative, the tape already covered by the read heads will never have to be revised.

In an implementation of the sequential simulator, the set of tapes is replaced by a merged list of pending events. Each pending event represents an expected change in an

output of an element given that the input state of the element remains unchanged. Items in the list are sorted in an ascending order with respect to their time values.

The position of the read heads is kept in a single variable called the *global clock*. Moving the read heads forward is accomplished by storing increasingly larger values into the global clock as events are pulled from the list of pending events. The simulator repeatedly sets the global clock to the time of the earliest event in the list, pulls that event from the list, and delivers it to the destination element. All events in the list except the top-most event are subject to revision because the assumptions of the elements that posted them — that their inputs will remain unchanged — may now be shown to be false. The event pulled from the top of the list will never need to be revised because the assumption of the element that posted it is now shown to be correct. The sequence of events pulled from the list represents the result of the simulation.

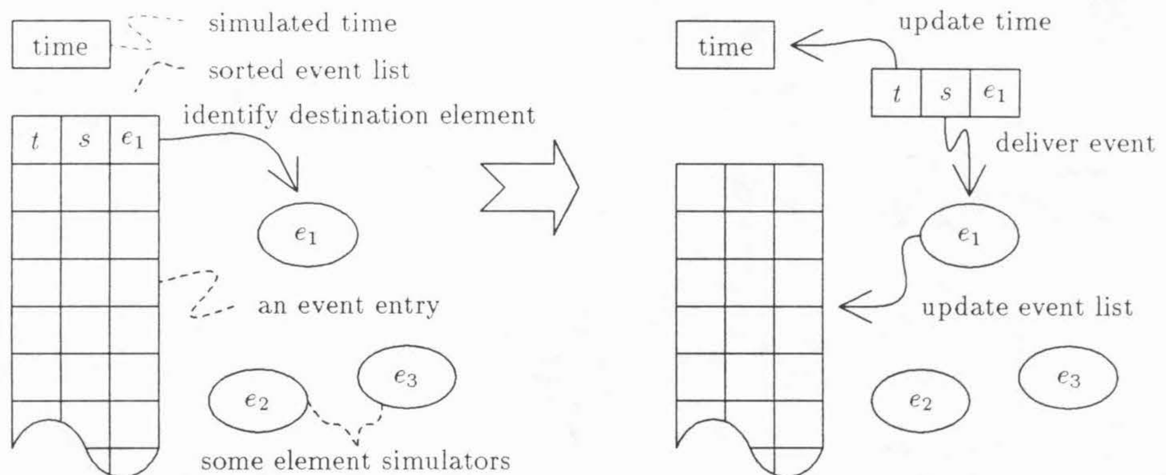

Figure 5.17 Model of a sequential simulator.

Suppose an obligatory slice is encountered during the simulation. If the state under the read heads forms a self-consistent state assignment for the slice, then there will be no events scheduled to change that assignment. The simulator will pass over the slice without detecting it. If the state assignment is not self-consistent, there will be events that change the state assignment. As the result of delivering such events, more events may be scheduled for the current simulation time because some destination elements may have a zero-delay.

If the intermediate state assignments eventually lead to a consistent state assignment, the pool of events under the read head will become empty and the global clock will be allowed to advance: if not, the simulator will be stuck processing an endless stream of events having the same event time.

Since there is one event delivery for every transition, a sequential simulator is also labeled event-driven; however, unlike the concurrent event-driven simulator described previously, the sequential simulator will never deadlock. The simulator is a complete simulator.

5.3.4 Concurrent backtracking simulators

Message-driven simulators do not backtrack, because every piece of information that each element simulator produces is correct. Backtracking simulators produce speculative information that can be revised when assumptions fail. In a sequential event-driven simulator, the amount of backtracking is limited by the alignment of the read heads. Since alignment is costly and reduces concurrency, concurrent backtracking simulators do not align read heads. The element simulators are allowed to produce outputs and to consume inputs according to their own heuristics and assumptions. When those assumptions are shown to be wrong, they have to restart the simulation from the point where the computation went wrong by backing up the write heads to discard erroneous information.

When a write head needs to be moved back behind a read head, the destination element of the read head has already consumed and may have produced its state and output based on false inputs; it too must be *rolled back*. In order to roll back to the time at which the input becomes invalid, the element simulator has to store a sequence of past states in addition to its current state.

Not all of the past state needs to be stored, however. In the *Time Warp* simulator of David Jefferson [14], a behind-the-scenes mechanism called the *global virtual time* is used to compute concurrently the lower bound of time for which rollback may still occur. The global virtual time attempts to keep track of the minimum time of all events and elements

in the simulation. Any saved state with a time value less than the global virtual time can be discarded, because no element will ever roll back to an earlier time.

The advantage of a backtracking simulator is that when a processor of the machine is otherwise idle, spare cycles can be used for speculative computing. Since this simulator must keep a record of past states for the elements, the concurrent backtracking simulator trades off space for speed by using larger processing nodes than would otherwise be necessary.

Concurrent backtracking simulators are complete simulators, and they handle obligatory slices the same way as do sequential simulators. When one is encountered, and if the state assignment of the elements involved is already self-consistent, the simulator moves ahead without detecting it. If the state assignment is not self-consistent, some of the elements involved will be rolled back to the starting time of the slice. and perhaps some more after that. The flurry of rollbacks ends when a self-consistent state is achieved.

5.3.5 Branch-and-bound simulators

If a backtracking simulator is likened to a depth-first search, then its breadth-first equivalent resembles a branch-and-bound simulator. This is one that trades off space for speed by using more processing nodes (rather than larger nodes) than would otherwise be necessary.

Suppose an element simulator computes to a point where its output can take on one of several states, depending on some inputs that have not yet arrived. Instead of producing a speculative output as would a backtracking simulator, the element simulator will, in effect, fork the simulation into a set of concurrent *branches* to follow each of the possibilities. In each branch, when the decisive input has finally arrived, should the input not match the assumption for a branch, then the branch will be terminated (*bound*).

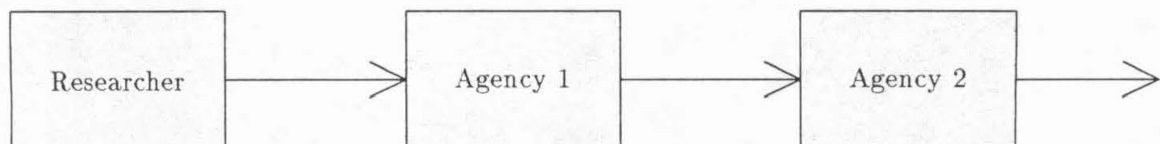

Figure 5.18 A researcher submitting a grant.

For comparison, suppose that a research grant request has to be approved in tandem by two government agencies. The first agency spends a *long* time classifying the grant into one of three classes, `A`, `B`, or `C`. The second agency spends a *long* time deciding whether the grant will be accepted based on the classification and the available funding for each class. A researcher submitting a grant can be represented by the system in Figure 5.18.

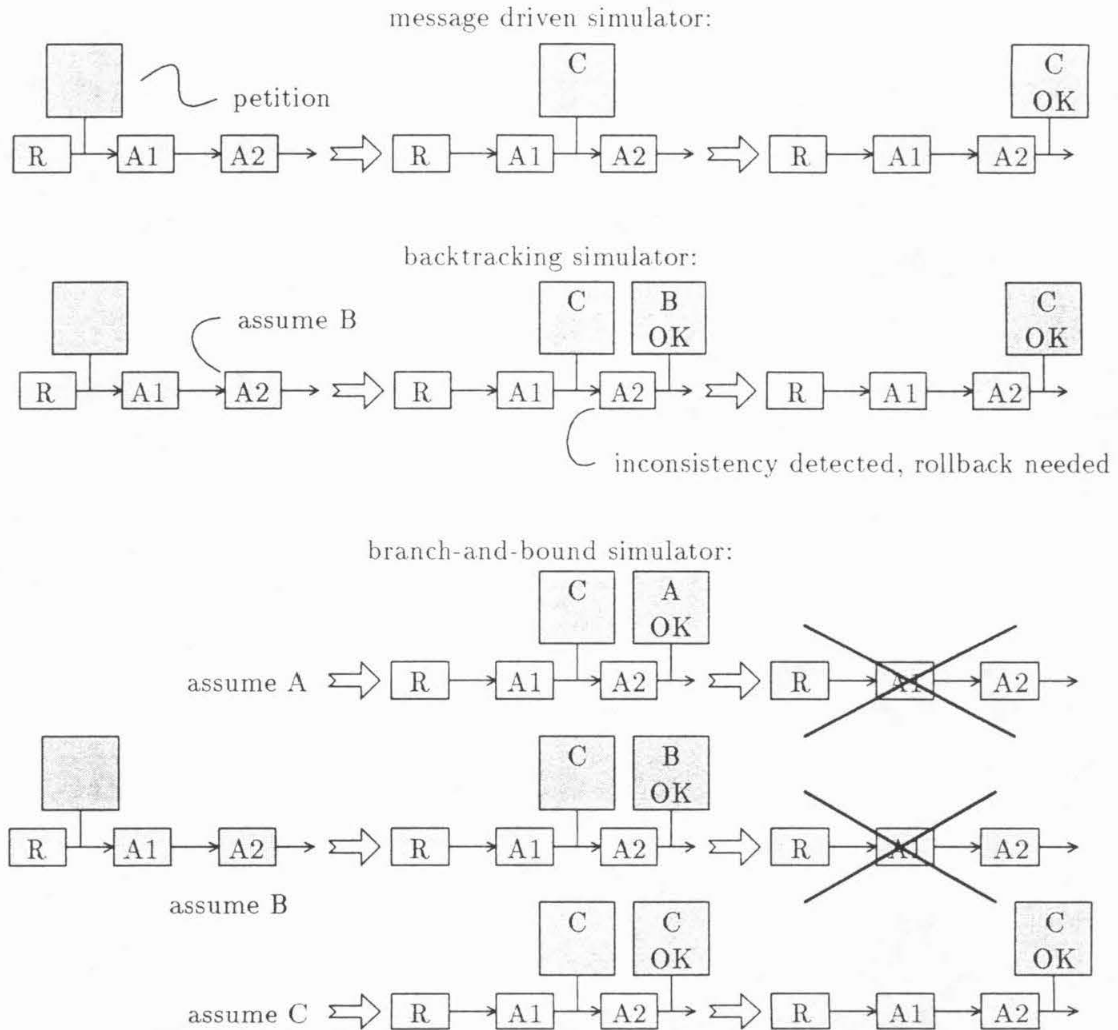

Figure 5.19 Comparison between three simulators.

In a message-driven simulator, only one agency simulator can be active at any one time. The time it takes to simulate the approval of the grant is equal to the sum of the time taken in each agency, because the operation is sequential. In a backtracking simulator, while the simulator for the first agency is working, the second agency can choose and pursue one

but only one of the three possible outcomes produced by the first agency. In a branch-and-bound simulator, three copies of the simulation are produced, each pursuing one of the three possibilities.

A branch-and-bound simulator is also a complete simulator. If there are any no-solution slices, all branches will be terminated and none will remain at the end. If there are any multi-solution slices, but no no-solution slices, more than one set of simulations will remain at the end, and each will correspond to one possible outcome. If there are only single-solution slices, then exactly one set will remain. The simulator will fail, however, if the number of solutions is unbounded, because the computing resource is bounded.

The branch-and-bound simulator is the only interesting type of distributed simulator that, so far as we know, is still to be explored. Efficient algorithms to fork and terminate the simulator may provide hope for the simulation of systems with very little intrinsic parallelism, and whose grain size is too small or whose behavior too unpredictable for rollback to be profitable.

5.3.6 Time-driven simulators

Thus far, we have discussed simulators that resolve slices by trial-and-error (backtracking) and by exploring all possibilities (branch-and-bound). In both methods, each element simulator needs only local information for progress. Neither method is appropriate, however, when the number of possibilities that must be explored is infinite. Exact simulation of such a system may require solving simultaneous equations analytically. When the equations can be solved, they yield functions of time, reducing the simulation to a simple task of function evaluation. When an analytical solution is inappropriate or difficult to find, empirical approximations must be used.

An example of such a system is an electrical circuit. In the system in Figure 5.20, the voltage across a capacitor is the integral of the current through the capacitor; the current, in turn, is a function of the voltage across the capacitor.

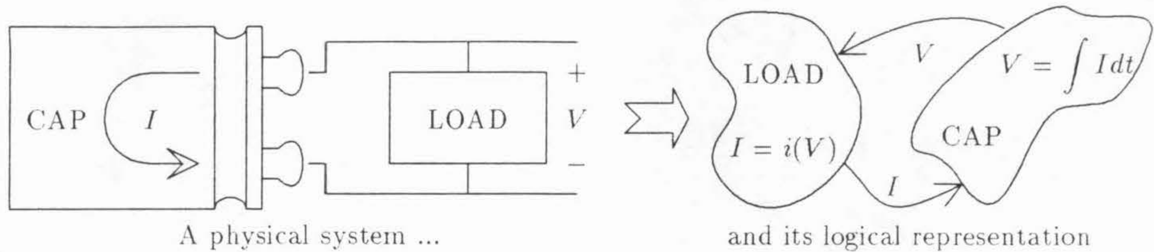

Figure 5.20 An example of a continuous system.

The equations: $V = \int I dt$

$$I = i(V)$$

In order to simulate this kind of system, we need to find a replacement system that is discrete but that will either approximate the behavior of the target system or converge to the final state of the target system. The usual method of building a simulator for such a system is to divide the simulation interval into a sequence of small slices. We then assume that information exchange takes place only at the boundaries of these slices, and information about the others can be accurately extrapolated between the boundaries.

For example, when integration of a continuous function is involved, discrete methods, such as taking the Riemann sum, can be used to approximate the integral of the function. Although discrete integration is seldom exact, we can get increasingly better approximations by reducing the size of the slices; when the size is reduced, the Riemann sum approaches the integral. However, due to accumulated numerical errors, the simulation may eventually diverge and produce an output that is valid only for a limited span of simulated time.

Simulators of this type are called time-driven simulators because they are moved forward at one time slice per step. Simulators of this type are also complete.

5.3.7 Summary

The slack-driven simulator is a generic simulator model that covers a large array of existing and hypothetical simulators. Simulators that perform computation on speculation, such as the concurrent-rollback simulator, are called *optimistic simulators*. Simulators that produce

no output other than that implied by the input are called *conservative simulators*. We will concentrate on the message-driven simulator, which is a conservative simulator.

We are particularly interested in the characteristics of the simulator itself, not those of a simulator plus any system it simulates. Thus, we have chosen the most revealing simulation subject, devised a series of conservative simulators, and reported in the following chapters the results obtained.

Chapter 6 Logic-Circuit Simulator Experiments

A Boolean network is a network of Boolean logic gates connected such that each input is driven from the output of another gate or from an input to the network. A logic circuit is a Boolean network that includes a notion of time: Each logic element in the network is assigned a positive value called the *delay* of the element. The input and output states of the gates are time-variant. If F is the Boolean function of a logic gate whose delay is δ, then the input state, I, and output state, O, are related by the equation:

$$O(t + \delta) = F(I(t))$$

Thus, unlike a Boolean network, which has a static value that is computed by solving a set of simultaneous equations, a logic circuit can have time-dependent behaviors, such as memory and oscillation. Simulation is a way of computing the behavior of a logic circuit.

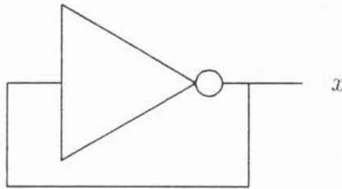

Figure 6.1 A logic circuit whose behavior is different from its Boolean network.

The Boolean network in Figure 6.1 can be described by the equation $x = NOT(x)$, which does not have a solution. As a logic circuit, however, the network is an oscillator. Although the input-output relationship of a logic circuit when it does reach a stable state is consistent with the corresponding Boolean network, our definition of a logic circuit simulator is one that reproduces the behavior of a logic circuit rather than one that solves for a stable state. The other definition is used by simulators such as *MOSSIM* [19], which simulates and verifies digital integrated circuits.

Most existing circuits found in computers and other digital systems belong to a class of circuits called *clocked* logic circuits. Clocked logic circuits are very well suited for the stable-state-solving form of simulation, because they are designed to reach a stable state during each clock cycle, and because only the final state of a clock cycle is needed to determine the future state. The exact sequence and timing of transitions that lead to a stable state

are usually not important; only the final stable state of the circuit is important. Such simulators, however, will not work very well for the unclocked, or self-timed, logic circuits.

Section 6.1 Why Logic Circuits?

We study logic-circuit simulation because it stresses a distributed simulator, and is itself of practical interest. It is easy to construct examples of logic circuits with diverse behaviors, structural difficulties such as large fan-in and fan-out, and highly non-uniform activity levels. Simple 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 involve practically no computation other than the distributed simulation itself. They are, therefore, uncluttered studies of how well the simulator itself performs.

A number of related simulators, each supporting an array of different simulation modes, have been written during the course of this study. These simulators run on multicomputers, such as the Cosmic Cube, Intel iPSC, and Symult 2010. Since they are written to run under the Cosmic Environment, they can be compiled for all of these machines without modification. The historical relationship between these simulators is shown in Figure 6.2. The arrows indicate predecessor-successor relationships.

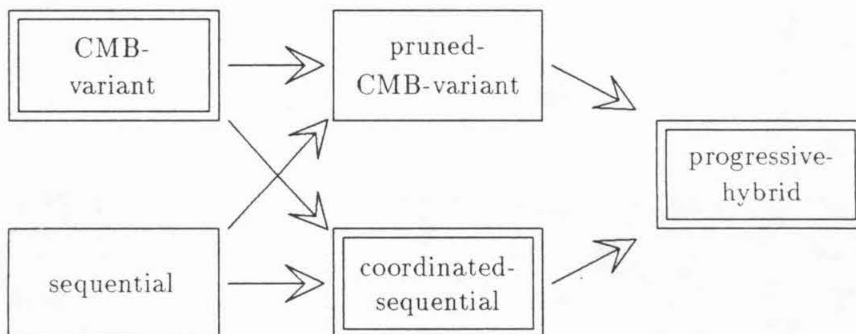

Figure 6.2 A number of circuit simulators and their relationship.

Of the five simulators shown, results obtained on three of them — the *CMB-variant*, the *coordinated-sequential*, and the *progressive-hybrid* simulators — are of interest. The *se-*

quential simulator and the *pruned-CMB-variant* are used for comparison only. The pruned-CMB-variant simulator will not be discussed.

The CMB-variant simulator is a straightforward implementation of the generic simulator in which the basic unit of information transfer is a block of state description over a time interval. The CMB-variant simulator shows excellent speedup as the number of nodes is increased, but, since it is totally oblivious to the content and effect of its information carriers, much of the work it has to do can be eliminated when an *event-driven* system is simulated on one node using a sequential simulator. However, sequential simulators cannot be readily distributed, and they cannot, in their original form, benefit from the use of multicomputers.

The three succeeding simulators attempt to combine the advantages of sequential and distributed simulators. The pruned-CMB-variant simulator is a CMB-variant simulator with sequential simulation mechanisms added. The coordinated-sequential simulator is a sequential simulator with CMB-variant mechanisms added. The progressive-hybrid simulator is the final merger of the two. In the following sections, we will describe each of these simulators in their chronological order.

Section 6.2 CMB-Variant Simulator

The CMB-variant simulator for logic circuits is a *proof of concept* for the generic simulator model described in Chapter 5. Since this is a demonstration of a generic model, in order to cover the greatest range of possible simulation subjects, special but useful properties of logic circuits have been ignored in building this simulator. In particular, the simulator ignores the fact that logic circuits are event-driven systems. We will discuss such systems in greater detail when we compare the result of this simulator to ones that do make use of the event-driven properties.

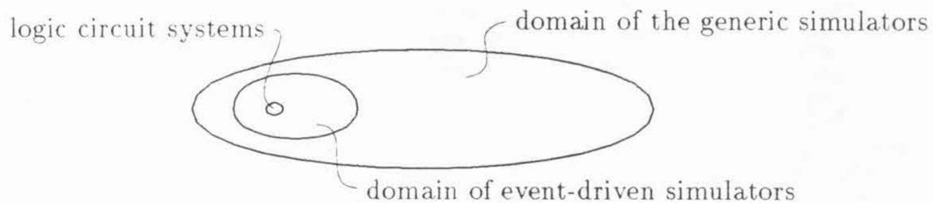

Figure 6.3 Domain of the generic simulator model.

The tape-writing and -reading processes in the generic simulator model are replaced by message-sending and -receiving processes in the CMB-variant simulator. These are light-weight, reactive processes, and the simulator is a reactive kernel for the reactive processes. As in a usual reactive-process program, the distribution of the simulation task on a multi-computer is accomplished by partitioning the set of reactive processes across a set of reactive kernels that run on a multicomputer.

We will present a simplified description of the CMB-variant simulator; the actual implementation contains extensive measurement setups and programming short-cuts that are inappropriate to report here. The simulator presented, however, is functionally correct, expresses the same principle as does the actual implementation, and is easier to understand.

6.2.1 The element simulators

First of all, a reactive process is represented by two pointers: the entry-function pointer and the data pointer. The entry-function pointer always contains the reference to the function that handles the next message for the process, but the data pointer can hold any private data structures needed by the process. For an element simulator, the private data may

include one data structure for each of the element's outputs. An output data structure contains the references to all inputs to which it connects. Each input reference contains the ID of the element that owns it and the index that identifies the input within the element. One output structure can contain more than one reference, because an output can connect to more than one input.

The private data may also include one input data structure for each of the element's inputs. Each input data structure contains the ID of the process and the identity of the output to which it connects. Each input can and must connect to one output.

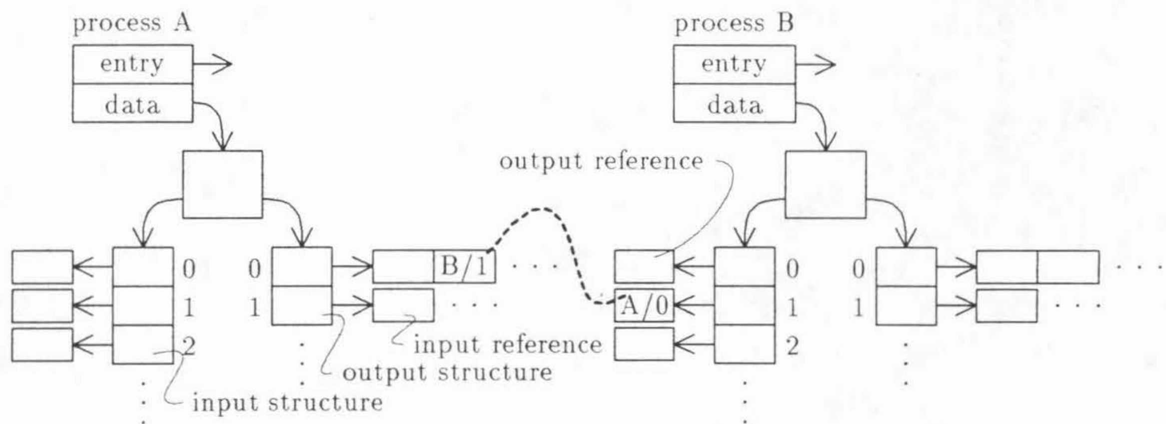

Figure 6.4 Process structure and a simple example of connectivity.

We may need a variable-sized message format to describe a piece of tape recording, because the information on the tape can be arbitrarily complex. In the interest of simplicity, however, we choose to represent each tape recording with more than one simple, fixed-sized message. We will call the structure a `STATE_FRAGMENT`. We use the name *fragment* to contrast it with the name *event* used in the study of traditional event-driven simulation systems, and to convey the fact that every entity is a fragment of a continuum that can be merged with adjacent entities and sliced into arbitrarily many entities.

The essential fields of a fragment are shown in Listing 6.1. When a fragment is received by a process, the `input_id` field identifies the element input to receive the fragment. The `state` and `span` fields describe the duration of a state at that input.

```
1  struct STATE_FRAGMENT
2  {
3      int           input_id; /* Index of the input at the dest element.    */
4      int              state; /* State contained in this fragment.          */
5      int               span; /* Duration of this fragment.                 */
6      STATE_FRAGMENT *next; /* Pointer to make a linked list of fragments.*/
7  } ;
```

Listing 6.1 Structure of a FRAGMENT.

When a piece of tape is to be written by an element in the generic simulator model, the corresponding process in the CMB-variant simulator produces one fragment or a stream of several fragments to carry the information recorded on the tape. When a fragment has arrived at its destination, the entry function of the destination process is called to accept the fragment. It is worth noting that reactive-process programming systems are themselves event-driven systems whose inputs are fragments. Thus, the simulator is always an event-driven system, even though the system it simulates may not be.

```
1  inverter_entry(pp,sb)
2      PROCESS        *pp;
3      STATE_FRAGMENT *sb;
4  {
5      OUTPUT(pp,0,!sb->state,sb->span);
6      free_fragment(sb);
7  }
```

Listing 6.2 An inverter in a CMB-variant simulator.

Listing 6.2 contains a sample entry function for an `inverter` element. As in an ordinary reactive process, the two parameters to its entry function are the process structure and the input message. When called, the entry function simply outputs another fragment of the same length, but with a complementary state value. The delay of the inverter is equal to the difference between the amount of fragments produced and the amount of fragments consumed. Such differences are set up during initialization by producing one fragment for each output of every gate, such that each fragment has a span that equals the delay of its output.

The `OUTPUT` function takes on four parameters. The first two parameters are the process structure and an index that identifies an output of the element. The function needs these

two parameters in order to access the list of destination input references for the output fragments. The next two parameters describe the state and the span of the fragment. In this example, there is only one output for the inverter, and its output index is 0. The state of the new fragment is the complement of the state contained in `sb->state` and the length of the fragment is the same as `sb->span`.

Since an inverter has only one input, it does not have to check the `input_id` of the fragments it receives, and it can immediately process any fragments it receives without waiting for other fragments to arrive. For a gate with more than one input, however, it usually has to differentiate the fragments it receives. Listing 6.3 contains a sample entry function for a two-input `XOR`-gate:

```
 1  xor_entry(pp,sb)
 2       PROCESS        *pp;
 3       STATE_FRAGMENT *sb;
 4  {
 5       int out_span, out_state;

 7       QUEUE_FRAGMENT(pp,sb);

 9       while(!Q_EMPTY(pp,0) && !Q_EMPTY(pp,1))
         {
11           out_state =     ( Q_HEAD(pp,0)->state ^ Q_HEAD(pp,1)->state );
12           out_span  = MIN( Q_HEAD(pp,0)->span  , Q_HEAD(pp,1)->span  );

14           OUTPUT(pp,0,out_state,out_span);

16           TRIM_QUEUE(pp,0,out_span);
17           TRIM_QUEUE(pp,1,out_span);
18       }
19  }
```

Listing 6.3 An `XOR`-gate in a CMB-variant simulator.

In a two-input `XOR`-gate, both of the inputs must have at least one fragment present before the gate can produce output fragments. The gate must therefore maintain a fragment queue for each of its input structures. When a fragment is received, the entry function can check the queues before deciding whether the fragment needs to be queued; but, in the interest of simplicity, the function always queues the fragment (7). The `QUEUE_FRAGMENT` function puts the fragment `sb` into an input queue of `pp` according to `sb->input_id`.

The Q_EMPTY function returns TRUE if the specified input queue for the process pp is empty. While both queues are non-empty (9), a length of fragment is removed from each queue to produce an output fragment. The state of the output fragment is equal to the exclusive-or of the states of the fragments to be removed (11). The length of the output fragment (and of each fragment to be removed) equals the length of the shorter fragment at the head of the queues (12). The Q_HEAD function returns a pointer to the first fragment in the specified queue.

The output of the exclusive-or gate remains the same as long as both inputs remain unchanged. The length of the shorter fragment is the length of time both inputs are known to remain unchanged. When fragments are consumed, output is produced (14), and a length equal to the length of the output fragment is trimmed from both queues (16,17).

The loop repeats until one of the queues becomes empty and the gate can no longer produce any additional output fragments from its queues. The inverter and the XOR-gate are simple because they are both *strict; ie.*, they do not have any partial input-state assignment such that the state of the outputs is not influenced by the state assignment of the remaining inputs.

An OR-gate, on the other hand, is *non-strict*: If any of the inputs is 1, its output will be 1, regardless of the state of its other inputs. An OR-gate can therefore continue to produce fragments in some situations where not all of its inputs are available. Listing 6.4 contains a sample entry function for an OR-gate:

```
 1  or_entry(pp,sb)
 2       PROCESS        *pp;
 3       STATE_FRAGMENT *sb;
 4  {
 5       int out_span, out_state;

 7       QUEUE_FRAGMENT(pp,sb);

 9       while(1)
10       {
11           if(!Q_EMPTY(pp,0) && (Q_HEAD(pp,0)->state == TRUE))
12           {
13               out_state =                TRUE;
14               out_span  = Q_HEAD(pp,0)->span;
```

```
16          } else

18          if(!Q_EMPTY(pp,1) && (Q_HEAD(pp,1)->state == TRUE))
19          {
20              out_state =                 TRUE;
21              out_span  = Q_HEAD(pp,1)->span;

23          } else

25          if(!Q_EMPTY(pp,0) && !Q_EMPTY(pp,1))
26          {
27              out_state =    ( Q_HEAD(pp,0)->state | Q_HEAD(pp,1)->state );
28              out_span  = MIN( Q_HEAD(pp,0)->span  , Q_HEAD(pp,1)->span  );

30          } else break;

32          TRIM_QUEUE(pp,0,out_span);
33          TRIM_QUEUE(pp,1,out_span);
34          OUTPUT(pp,0,out_state,out_span);
35      }
36  }
```

Listing 6.4 An OR-gate in a CMB-variant simulator.

When the process receives a fragment, it is added to the queue, as in the case of the XOR-gate. But, then, instead of checking both of the queues for fragments, the function checks first for possible non-strict input conditions. Lines 11–16 check the input whose index is 0; lines 18–23 check the input whose index is 1. If a fragment for an input is available and its state is TRUE, then a non-strict input condition exists. The new output fragment is specified to have a state value of TRUE and a span equal to the span of the fragment in the queue. The function then continues to line 32 where fragments are trimmed from the queues and an output fragment is produced. If no non-strict conditions have been detected, the process will compute and produce fragments in the same manner as the XOR process (26–28).

When a non-strict condition is detected on one input, the queues in both of the inputs are trimmed (32–33) because the state of the other input does not matter. However, it is possible that the queue for the other input is empty or does not contain enough fragments to cover the amount to be trimmed. In this case, the trimming extends to fragments that have not yet arrived. The process must therefore record the deficit incurred and deduct it from fragments that arrive later.

```
1  typedef struct { int        delay;             /* Delay of the element.*/
2                   I_DATA     *inpq;             /* One per gate input.  */
3                   O_DATA     *outq; } ELEMENT;  /* One per gate output. */

5  typedef struct { STATE_FRAGMENT *qh;            /* Points to top.       */
6                   STATE_FRAGMENT *qt;            /* Points to bottom.    */
7                   int        deficit; } I_DATA;  /* Deficit of the queue*/
```

The details for the process are complete; we are ready to show the essential mechanisms that support the processes. The process structure contains an entry function; an array of input data structures, one for each element input; and an array of output data structures, one for each element output. These data structures are set up during initialization. The input structure contains the deficit count and a pair of queue pointers, one for the head of the queue and one for the tail.

```
 1  QUEUE_FRAGMENT(pp,sb)
 2      PROCESS        *pp;
 3      STATE_FRAGMENT *sb;
    {
 5      I_DATA      *Q;

 7      Q = ((ELEMENT *)(pp->data))->inpq + sb->input_id;

 9      if(Q->deficit)
        {
11          if(sb->span <= Q->deficit) { Q->deficit -= sb->span   ;
12                                        free_fragment(sb); return; }
13                                 else { sb->span    -= Q->deficit;
14                                        Q->deficit  =          0; }
        }

17      if(Q->qh)
        {
19          if(sb->state == Q->qt->state) { Q->qt->span += sb->span   ;
20                                          free_fragment(sb); return; }
21                                   else { Q->qt = Q->qt->next = qt;
22                                          qt->next = 0;              }
        } else
        {
25          Q->qh = Q->qt = sb;
26          sb->next = 0;
        }
    }
```

Listing 6.5 CMB-variant QUEUE_FRAGMENT function.

The QUEUE_FRAGMENT function adds the fragment, sb, to the (sb->input_id)th input queue of the process pp. It checks first for the deficit (9). If a deficit exists, the span of the fragment is used to satisfy the deficit; if the fragment is totally consumed (11–12), the

function returns. Otherwise, the balance is advanced to the next step, where fragments are added to the queue (17). If there are already other fragments in the queue (17), and if the last fragment has the same state as the new fragment (19), the two are simply merged (19-20). Otherwise, the fragment is linked into the queue (21–22, 25–26).

```
 1 TRIM_FRAGMENT(pp,id,debit)
 2     PROCESS    *pp;
 3     int         id;
 4     int      debit;
 5 {
 6     I_DATA          *Q;
 7     STATE_FRAGMENT *sb;

 9     Q = ((ELEMENT *)(pp->data))->inpq + id;

11     while(debit && Q->qh)
       {
13         if(Q->qh->span > debit) { Q->qh->span -= debit;
14                                   debit        =      0; }
15                            else { debit -= Q->qh->span;
16                                   sb     = Q->qh       ;
17                                   Q->qh  = sb->next    ;
18                                   free_fragment(sb)    ; }
       }

21     Q->deficit += debit;
22 }
```

Listing 6.6 CMB-variant TRIM_FRAGMENT function.

The TRIM_FRAGMENT function removes debit amount of fragments from the id-th input queue of the process pp. As long as there are more fragments in the queue, the spans of as many fragments as necessary, taken from the head of the queue, are used to satisfy the debit. Any remaining debit is added to the deficit of the queue.

6.2.2 The simulator message system

The list of references and indices for each output structure described above represents a one-level tree. The root of the tree is the sending process and the leaves of the tree are the receiving processes. The job of the OUTPUT function is simple enough — it allocates a fragment for each leaf process and sends it along the branch that leads to the process. In such a simulator, however, gates with a large fan-out, such as a clock driver, may have to send the same information to the same destination computing node many times.

Because messages between computing nodes are usually more expensive than messages within the same computing node, we reduce the internode messages by organizing the tree as a two-level tree. The intermediate tree nodes are a set of *input port* processes, one for each computing node that contains a destination process. An output sends its fragment to its input ports, and an input port duplicates and forwards the fragment to the destination processes in its own computing nodes.

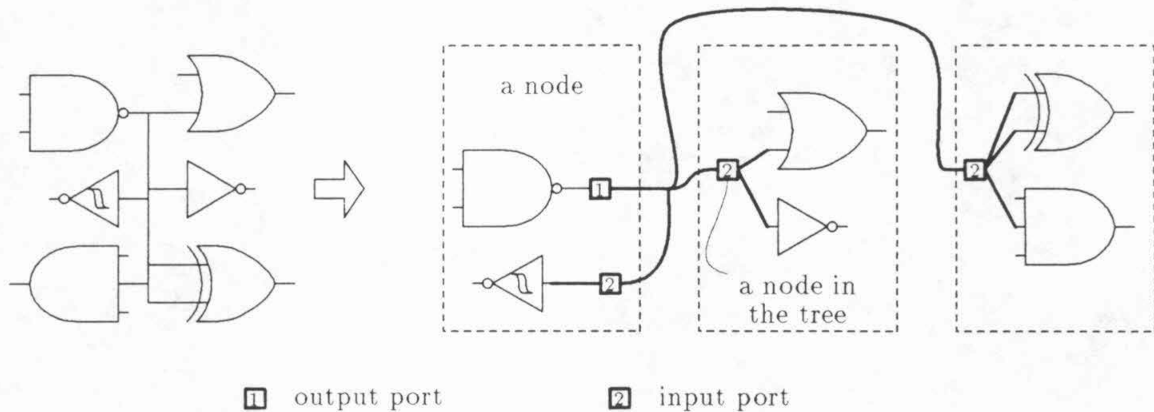

Figure 6.5 A sample circuit and a possible mapping to a multicomputer.

Many mechanisms can be added to the output structure for a more more efficient simulator, and such mechanisms account for the majority of the differences between the actual implementation and this description. Here we will present a simple OUTPUT function that converts fragments into messages that are immediately sent.

```
1  typedef struct { int       count;               /* Number of siblings.  */
2                   int       *node;               /* Dest process's node. */
3                   int       *pid2;               /* Dest process's pid2. */
4                   int  *input_id; } O_DATA;      /* Dest process's input */
```

The output data structure contains the number of ports connected and a list of references to those ports. A reference for a process in the simulator contains the node and the pid of the destination simulator process. It also contains a pid2, because the element processes are embedded in the simulator by reactive-handler layering. Only the node and the pid2 need to be stored in the output structure, because in our implementation there is only one simulator process for every node, and all of them have the same fixed pid. Listing 6.7 contains a sample OUTPUT function:

```
 1  OUTPUT(pp,id,state,span)
 2      PROCESS     *pp;
 3      int          id;
 4      int       state;
 5      int        span;
 6  {
 7      int               j;
 8      O_DATA          *op;
 9      STATE_FRAGMENT *sb;

11      op = ((ELEMENT *)(pp->data))->outq + id;

13      for(j = 0; j < op->count; j++)
14      {
15          sb              = new_fragment()       ;
16          sb->input_id = op->input_id[j]       ;
17          sb->state      = state                ;
18          sb->span       = span                 ;
19          s_send(msg,op->node[j],op->pid2[j]);
20      }
21  }
```

Listing 6.7 CMB-variant OUTPUT function.

The OUTPUT function allocates a fragment for each branch of the tree (15), initializes it with the input index of the destination input (16), sets the state and span (17–18), and sends the fragment (19). The s_send function is a layered message function that sends the message to another process in the simulator. If a two-level tree structure is used, each fragment goes to an input port process that is identical to the inverter process except that the state is not inverted (a buffer process). The main function for the simulator is identical to that of a reactive kernel:

```
 1  struct { int   (*entry)();
 2           char   *data    ; } PROCESS;

 4  struct { int        pid2  ;
 5           char  msg_body[]; } MESSAGE;

 7  simulator_main_loop()
 8  {
 9      PROCESS *proc;
10      MESSAGE *mesg;

12      while(1)
13      {
14          mesg = (MESSAGE *) xrecvb();
15          proc = process_table + mesg->pid2;
16          (*proc->entry)(proc, mesg->msg_body);
17      }
18  }
```

Listing 6.8 CMB-variant main loop.

This is the end of our description of a simple, distributed simulator derived directly from the generic simulator model. The description is complete except for the storage allocation/de-allocation mechanisms, the initialization/termination mechanisms, and the result-recording mechanisms.

6.2.3 The variants

Although this simulator exhibits excellent performance for some cases, much can be done to improve its performance for difficult cases. The number of actual messages, for example, can be reduced in a logic circuit simulation by using a more elaborate `OUTPUT` function. In particular, if message sending is deferred by putting fragments into output-holding queues, the opportunity to merge multiple fragments into a single message increases. When two successive fragments with the same state are put into the same holding queue, the two can be merged into a fragment with a larger span, saving both space and handling time. Even if they cannot merge, multiple fragments can be concatenated onto a single, longer message to share the per-message overhead.

If sending is deferred forever, however, the simulator will fail to make any progress. Good efficiency can be achieved with a proper balance of message deferral and message sending. Before we devised and evaluated a number of flow control methods, there were two methods that represented the two extremes of possibilities: the two original CMB-methods. (Hence, our methods are called variants.) In the *deadlock-avoidance* method, no fragments are deferred and deadlock does not occur. In the *deadlock-detection* method, no message is sent until the simulation runs into a deadlock, or unless the output-holding queue contains an event. A deadlock-detection mechanism running concurrently in the simulator message system detects the deadlock and forces deferred messages to be sent.

We generally call those methods that are more likely to send messages *eager* methods, and those that are less likely to send messages *lazy* methods. Thus, the *deadlock-avoidance* method is at the eager end of the spectrum, and *deadlock-detection* method is at the lazy end. To explore the middle ground, we needed to hold back messages by some criteria we

could select, but in order to prevent deadlock detection from dominating the timing, we needed a cheaper way of ensuring progress than by using standard deadlock detection.

When simulator processes defer sending output messages, they may cyclically deny themselves input messages, leading to deadlock. However, deadlock implies that some node has an empty input-message queue. Since the emptiness of the queue is a local condition, we make use of that condition to modify the behavior of the simulator to prevent deadlock. Our strategy is called *indefinite-lazy message sending*, and is implemented by replacing the `xrecvb` function in the simulator's main loop with a non-blocking `xrecv`.

```
 1  simulator_main_loop()
 2  {
 3      PROCESS *proc;
 4      MESSAGE *mesg;

 6      while(1)
 7      {
 8          if(mesg = (MESSAGE *) xrecv())
 9          {
10              proc = process_table + mesg->pid2;
11              (*proc->entry)(proc, mesg->msg_body);
12          } else
13          {
14              take_action_to_promote_progress();
15          }
16      }
17  }
```

Listing 6.9 CMB-variant indefinitely-lazy main loop.

The function `xrecv` returns a message for an element simulator if the node's input-message queue is not empty. The simulator goes on to deliver the message as before if a message is returned. While an element simulator is consuming a message, it may either send or withhold any output that the element simulator produces according to the heuristics in effect at the time.

If the node's input-message queue is empty, a null pointer is returned and deadlock is a possibility. The simulator will take special actions to break potential deadlocks. Actions can generally be classified into two types: For the source-driven type, the simulator selects a deferred output to send as a message; for the demand-driven type, the simulator selects

a blocked element, and sends a *demand* message to its predecessor to request that queued outputs be sent. The end result is that deadlock is prevented.

6.2.4 Variant algorithms

We have experimented with many CMB variants. Since many of them are closely related, and all of them show similar performance results, we will describe the operation and report the performance of just six variants (A–E) that are representative of the range of possibilities that we have studied:

A *Eager message sending:* This is the deadlock-avoidance CMB simulator.

B *Eager event:* Since successive fragments with the same state value can be merged into one fragment, the *eager-event* variant detains all output fragments until a fragment that cannot be merged with its predecessor is produced. When `xrecv` returns a null pointer, the detained fragment that extends to the earliest time is sent. This is called an eager-event variant because state changes are called *events* in event-driven systems, and because this simulator will eagerly send event-conveying fragments.

C *Indefinite-lazy, single-dispensation:* All output fragments produced by element simulators are queued. Messages are sent only when `xrecv` returns a null pointer. The output queue that extends to the earliest time is selected, and one fragment from that queue is sent.

D *Indefinite-lazy, multiple-event:* This scheme is a variation on *C*, motivated by characteristics of multicomputer message systems that make it economical to pack multiple events into fewer messages. All output fragments produced by element simulators are queued. When `xrecv` returns a null pointer, the output queue that extends to the earliest time is selected to generate a message using all of the fragments in that queue, instead of just one.

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 fragments produced by element simulators are queued. When **xrecv** returns a null pointer, the input port that lags furthest behind is picked to select the destination for a demand message. Upon receipt of a demand message, if the output queue is not empty, the simulator sends all fragments in the output queue; if the output queue is empty, the simulator propagates the demand message. For the demand-driven variant, the message header must also carry a type field to distinguish a normal message from a demand message.

```
 1  struct { int       pid2  ;
 2           int       type  ;
 3           char  msg_body[]; } MESSAGE;

 5  simulator_main_loop()
 6  {
 7      PROCESS *proc;
 8      MESSAGE *mesg;

10      while(1)
11      {
12          if(mesg = (MESSAGE *) xrecv())
13          {
14              if(mesg->type == DEMAND_TYPE)
15              {
16                  handle_demand_message(mesg->msg_body);
17              } else
18              {
19                  proc = process_table + mesg->pid2;
20                  (*proc->entry)(proc, mesg->msg_body);
21              }
22          } else
23          {
24              take_action_to_promote_progress();
25          }
26      }
27  }
```

Listing 6.10 CMB-variant demand-driven main loop.

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 for processes that get them. In the examples that we show, the threshold is halved. The

simulator is accordingly able to adapt itself to the characteristics of the system being simulated.

6.2.5 Instrumentation

Although execution time is one of the most natural bases of comparison between any two programs that perform the same function, and although it 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, we run our simulator programs under the control of a multicomputer simulator (*sweep mode*). A close examination of the main routine of the simulator reveals that it can be transformed with minimal modification into a light-weight reactive-process program under yet another layer of the reactive kernel:

```
 1  SIM_DATA *simulator_data;

 3  simulator_main_loop(simp,mesg)
 4      PROCESS *simp;
 5      MESSAGE *mesg;
 6  {
 7      PROCESS *proc;

 9      simulator_data = (SIM_DATA *)(simp->data);

11      if(mesg)
12      {
13          if(mesg->type == DEMAND_TYPE)
14          {
15              handle_demand_message(mesg->msg_body);
16          } else
17          {
18              proc = simulator_data->process_table + mesg->pid2;
19              (*proc->entry)(proc, mesg->msg_body);
20          }
21      } else
22      {
23          take_action_to_promote_progress();
24      }
25  }
```

Listing 6.11 CMB-variant main loop as a light-weight process.

The process structure in this reactive kernel is described by the `SIM_DATA` structure in the above listing. The structure contains a list of element simulator processes and any other

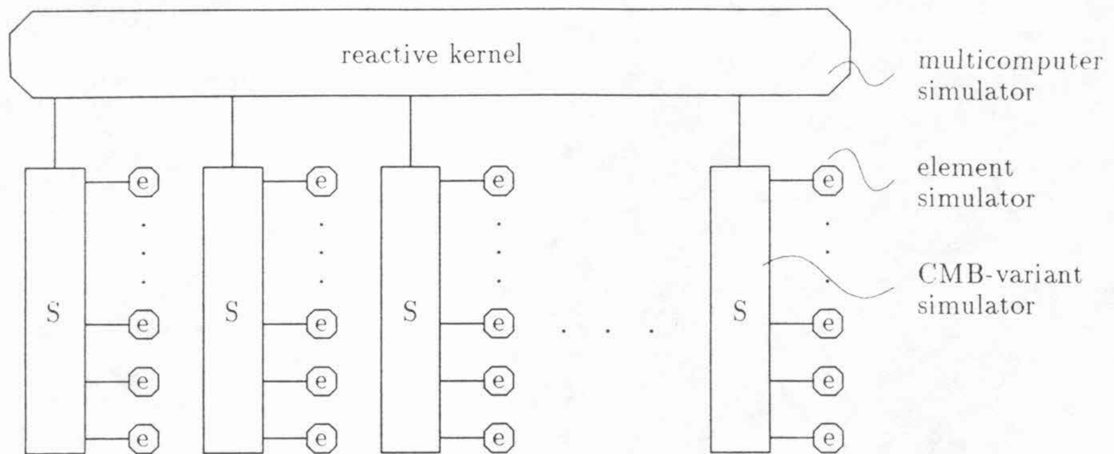

Figure 6.6 Structure of a sweep-mode simulation.

data structures private to this instance of the simulator.

Sweep-mode simulation for an N-node multicomputer is accomplished with a reactive kernel that runs N copies of the simulators as reactive processes. Execution time is then measured in a unit called a *sweep* [2, 15], 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.

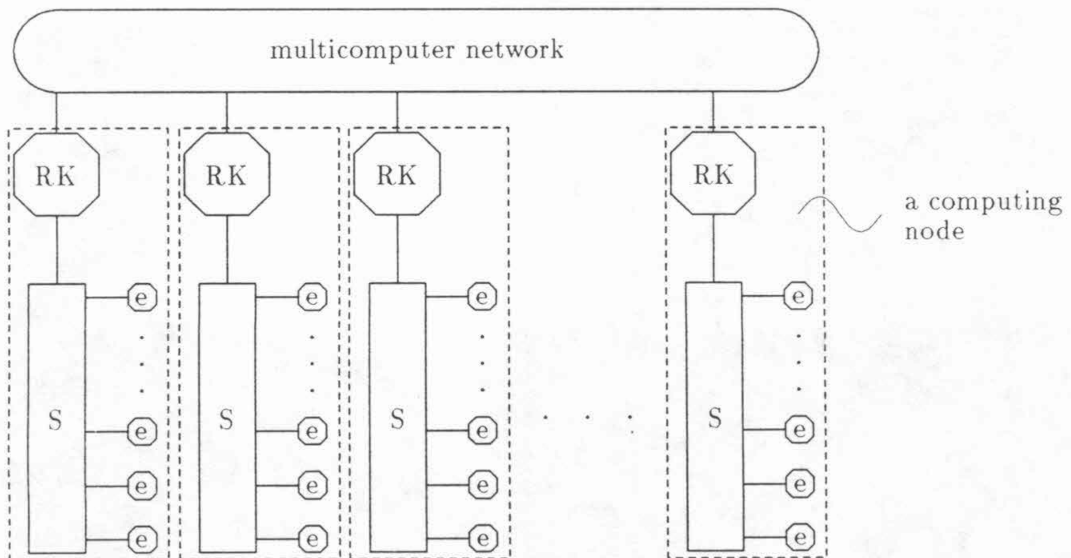

Figure 6.7 Structure of a real-mode simulation.

In the *real-mode* simulation, the simulator is linked with a reactive heavy-weight handler

and run 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 the host computer's real-time clock.

6.2.6 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. The majority of the logic gates have delays of between 1 and $80ns$, with $20ns$ being a typical value. Each simulation was run for a predetermined, simulated interval, and a set of measurements, including the real elapse time, was recorded. A larger variation in performance was observed among networks with different characteristics than between algorithm variants.

The parallel multiplier is a good example of an ordinary logic network. The 14×14 array 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 advantages or troubles that depend 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 expected to do neither 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. The resulting activity level is quite low: The entire circuit is idle 25% of the time. For the other 75% of time, there is a wavefront of activity

moving diagonally down the array. After the wavefront hits the bottom-left corner, the multiplier input changes and broadcasts the change to 144 gates. A trace of the product outputs shows that the simulator and the circuit are running correctly.

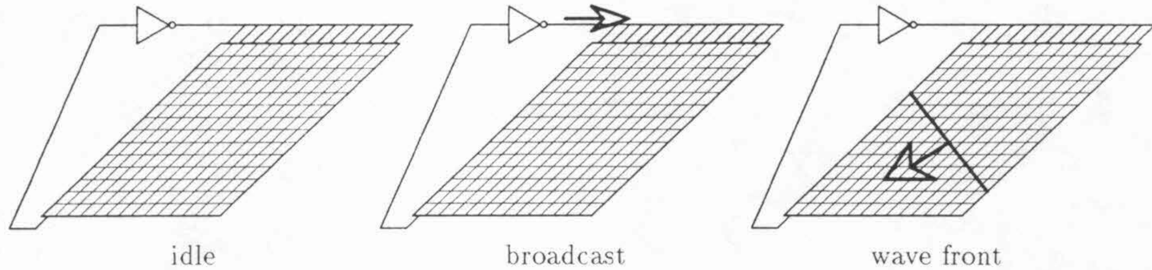

Figure 6.8 Three phases of the oscillating multiplier.

The plot in Figure 6.9 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.

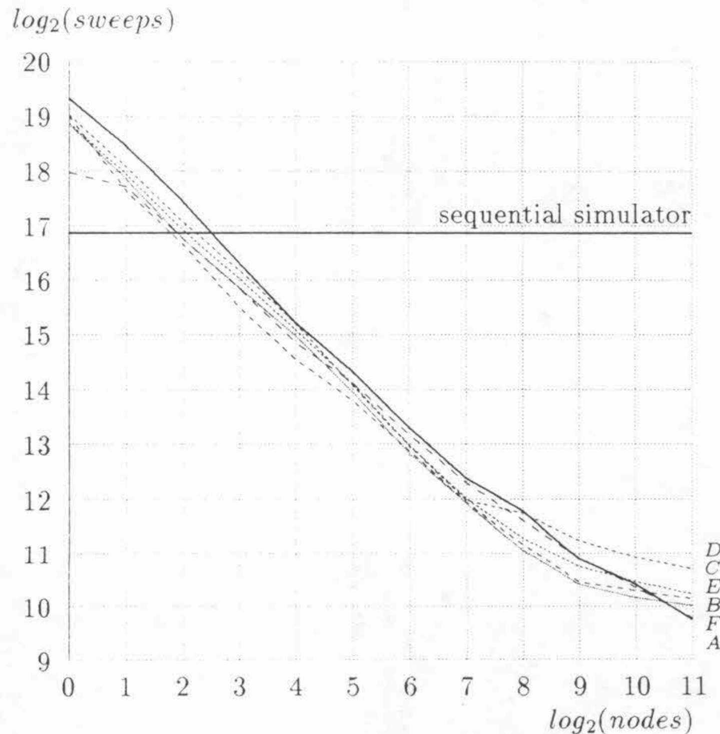

Figure 6.9 A 1376-gate multiplier, sweep-mode.

It is not useful to continue the plot beyond 2^{11} nodes, since at this point there are as many nodes as simulated gates. Each horizontal division represents a factor of two in the

number of nodes used; each vertical division represents a factor of two in sweep count or time. The placement of elements in nodes for these trials is a systematic pattern that tends to put related elements into the same node.

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 the sweep 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 performance is 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 minimum 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. The gross characteristics of these curves are similar to those of other concurrent programs [2], and are quite understandable and predictable.

Like many other concurrent algorithms, a more efficient sequential algorithm exists for the CMB-variant simulator when applied to circuit simulation. The heavy horizontal line represents the number of sweeps a sequential event-driven simulator requires for this same simulation. 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. We shall refer to this

extra work that the CMB-variant simulator does as the *overhead* of distributing the simulation. We will discuss the sequential event-driven simulator and additional performance measurements in the next and subsequent sections.

Section 6.3 Sequential Simulator

At $N = 1$, the sequential simulator does better than do the CMB-variant simulators for two reasons: The first is that logic circuits are event-driven systems in which the time it takes for a sequential simulator to handle and process a fragment is zero if the fragment does not convey an event. (A fragment conveys an event if its state differs from the fragment that precedes it. A message that carries an event-conveying fragment is an *event message*; a message that does not is a *null message*.) The second is that logic gates are simple and the time it takes for an element simulator to process an event-conveying fragment is almost zero.

Since the message-handling times for null messages and event messages are identical in the CMB-variant simulator, the ratio at $N = 1$ (N is number of nodes used) between the time taken by the sequential and the CMB-variant circuit simulators reflects the proportion of event messages in a CMB-variant circuit simulator. The cost of handling null messages is the *overhead* of the CMB-variant simulator at $N = 1$.

6.3.1 Sequential simulator mechanism

Like the CMB-variant simulator, our sequential simulator is also a reactive-process program with embedded, light-weight, reactive processes. Each message in this simulator, called an *event*, describes a state transition and includes the following fields:

```
1  struct EVENT
2  {
3      int   input_id; /* Index of the input at the dest element. */
4      int       time; /* Time of the transition.                 */
5  } ;
```

Listing 6.12 Sequential-simulator event structure.

The `time` field of an event represents the time when a state change will occur at the input (identified by the value of the `input_id` field) of the process that receives the event. The function contained in Listing 6.13 can be used as an entry function for an inverter gate.

```
1  inverter_entry(pp,ep)
2      PROCESS *pp;
3      EVENT   *ep;
4  {
```

```
5       SEND_EVENT(pp, 0, ep->time);
6       free_event(ep);
7   }
```

Listing 6.13 An inverter in sequential simulator.

When the simulator delivers an event to the inverter, the inverter will generate an output event with an event time that is pp->delay units larger. The SEND_EVENT function takes three parameters: Like the OUTPUT function of the CMB-variant simulator, the first two parameters are the process structure and the index that identifies an output of the element; the third parameter is a time value whose sum with the element delay becomes the time of the output event. Listing 6.14 contains a simple output routine for the sequential simulator:

```
 1  SEND_EVENT(pp,id,time)
 2      PROCESS    *pp;
 3      int          id;
 4      int        time;
 5  {
 6      EVENT      *ep;
 7      O_DATA     *op;
 8      int          ot;

10      op = ((ELEMENT *) (pp->data))->outq  +   id;
11      ot = ((ELEMENT *) (pp->data))->delay + time;

13      for(j = 0; j < op->count; j++)
14      {
15          ep           = new_event()    ;
16          ep->input_id = op->input_id[j];
17          ep->time     = ot             ;

19          ADD_EVENT(ep,op->pid2[j]);
20      }
21  }
```

Listing 6.14 The SEND_EVENT function in sequential simulator.

The routine allocates an event structure (15) for every input connected, fills in the receiver input index (16), fills in the time of the event (17), and inserts the event into the event list (19). This routine is structurally similar to the OUTPUT routine of the CMB-variant simulator, except that node numbers are not used to identify processes because all processes reside in the same node. In order to reduce the number of events that must be sorted when

more than one input is connected, output-event duplication in the actual implementation is performed at the time of event delivery.

It is interesting that the entry function for an XOR-gate is identical to that of an inverter. Listing 6.15 contains the more complex, OR-gate entry function.

```
 1  or_00(pp,ep) PROCESS *pp; EVENT *ep;
 2  {
 3      if(!ep->input_id) { pp->entry = or_01; SEND_EVENT(pp,0,ep->time); }
 4                  else { pp->entry = or_10; SEND_EVENT(pp,0,ep->time); }
 5      free_event(ep);
 6  }

 8  or_01(pp,ep) PROCESS *pp; EVENT *ep;
 9  {
10      if(!ep->input_id) { pp->entry = or_00; SEND_EVENT(pp,0,ep->time); }
11                  else { pp->entry = or_11;                             }
12      free_event(ep);
13  }

15  or_10(pp,ep) PROCESS *pp; EVENT *ep;
16  {
17      if(!ep->input_id) { pp->entry = or_11;                             }
18                  else { pp->entry = or_00; SEND_EVENT(pp,0,ep->time); }
19      free_event(ep);
20  }

22  or_11(pp,ep) PROCESS *pp; EVENT *ep;
23  {
24      if(!ep->input_id) { pp->entry = or_10;                             }
25                  else { pp->entry = or_01;                             }
26      free_event(ep);
27  }
```

Listing 6.15 An OR-gate in sequential simulator.

When both gate inputs are 0, the entry function is or_00. When an event is received, the event is distinguished by the input it affects. If the event is for the input whose index is 0, the entry-function pointer is set to or_01, and an output event is produced (2). If the event is for the other input, the entry function is set to or_10 and an output is also produced (3). The actions for the other three entry functions are similar.

An element can compute its output state based only on a transition from one of its inputs, because the transition carries the assurance that the other inputs of the element have not changed. Such assurance can be provided in several ways. The most common method is to keep the set of yet-to-be-delivered events (the *pending events*) sorted by time

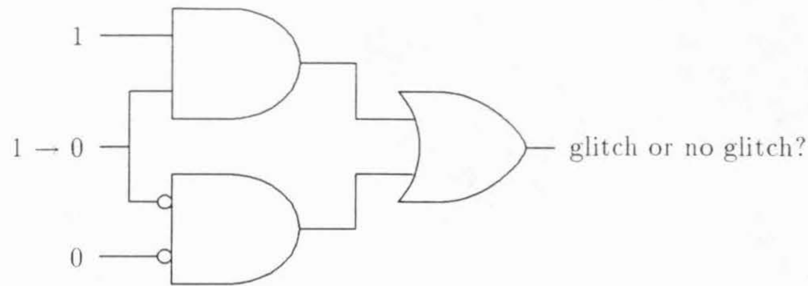

Figure 6.10 A circuit containing a dynamic hazard condition.

in an *event list*, and to deliver the event with the smallest time value first. Since element delays cannot be negative, an event cannot trigger events with smaller time values. When an event is delivered to an element, it is assured that the other inputs of the element, and indeed of all other elements, will remain unchanged up to the time of the event.

```
 1  struct { int       pid2  ;
 2           char  msg_body[]; } MESSAGE;

 4  simulator_main_loop(simp,mesg)
 5      PROCESS *simp;
 6      MESSAGE *mesg;
 7  {
 8      PROCESS *proc;

10      proc = (SIM_DATA *)(simp->data)->process_table + mesg->pid2;
11      (*proc->entry)(proc, mesg->msg_body);
12  }
```

Listing 6.16 Sequential-simulator main loop as a light-weight process.

The simulator main loop is similar to that of the CMB-variant simulator; the message system, however, has a different property. The message system for the CMB-variant simulator dispenses messages on a first-come, first-served basis; for the sequential simulator, the message with the smallest time value is dispensed first.

6.3.2 Hazards in sequential simulators

Although a sequential simulator will always produce a valid simulation result, it may not always produce the same result as the CMB-variant simulator. Some input conditions in a logic circuit may trigger more than one possible outcome, and a sequential simulator has no consistent way of choosing one. For example, the `OR`-gate in Figure 6.10 can produce either no transitions, or two transitions in response to two simultaneous input events. This condition corresponds to a *static hazard* in the terminology of Boolean minimization.

Both of these responses are correct because the temporal relation between the two input events is beyond the capability of the model to resolve; the one that is produced depends on the order in which the two input events are consumed. Since both input events have the same time value, they can be taken from the list in either order. If the low-going transition is taken first, two output transitions will be produced; if the high-going transition is taken first, no output transitions will be produced. The CMB-variant simulator, however, consistently picks the response in which no output transitions are produced.

Although both responses are considered to be correct, the sequential simulator can compare unfavorably with the CMB-variant simulator when there are too many extra events. For the comparison to be meaningful, we must devise a sequential simulator that will consistently make the same choices as does the CMB-variant simulator. In a system in which every element has a non-zero delay, this can be accomplished by withdrawing the first of the two output events when the second output event is to be produced, and canceling both events. Each output data structure must maintain a reference to the last unconsumed event that it has produced. When another output event is to be produced, if the previous event has not been consumed and if the two events have the same time value, then no events are produced and the previous event is withdrawn. The following SEND_EVENT function implements this mechanism.

```
 1  SEND_EVENT(pp,id,time)
 2      PROCESS   *pp;
 3      int         id;
 4      int       time;
 5  {
 6      EVENT      *ep;
 7      O_DATA     *op;
 8      int         ot;

10      op = ((ELEMENT *) (pp->data))->outq  + id  ;
11      ot = ((ELEMENT *) (pp->data))->delay + time;

13      for(j = 0; j < op->count; j++)
14      {
15          if(op->last_e[j] && (op->last_e[j]->time == ot))
16          {
17              DEL_EVENT(op->last_e[j]);
18              op->last_e[j] = 0;

20          } else
```

```
21          {
22              ep              = new_event()    ;
23              ep->input_id    = op->input_id[j];
24              ep->time        = ot             ;
25              op->last_e[j]   = ep             ;

27              ADD_EVENT(ep,op->pid2[j]);
28          }
29      }
30  }
```

Listing 6.17 A SEND_EVENT function that reduces glitches.

Missing from Listing 6.17 is the part that places a back-reference pointer into each event structure. The back-reference is used by the simulator to dissociate an event from its output (by setting the corresponding `last_e[j]` to 0) when the event is delivered.

6.3.3 Instrumentation

The sequential simulator also exists in two modes, sweep mode and real mode. Like the CMB-variants, the sweep-mode simulator consumes one sweep for every element input delivery. In the real mode, the CMB-variant simulator must poll the system's input message queue once for every null message or event message delivered; the sequential simulator is also made to poll the same queue once for every event message delivered, even though this is never necessary. Polling for messages consumes a significant amount of time in many multicomputers but there is nothing inherently costly about the operation. It should be possible in a future machine to poll the queue by checking only a single pre-defined memory location that has been mapped into each process's memory space.

The resulting real-mode simulator runs at a speed of about $500\mu s$ per event for our examples on the iPSC/2 and the Symult 2010, and at about $3000\mu s$ per event on our iPSC/1. The polling time is about $170\mu s$ for the Symult 2010 and $760\mu s$ for the iPSC/1. The iPSC multicomputers were running Cosmic Environment in compatibility mode instead of in the potentially more efficient native mode. The exact speed depends on the size of the event list. The event list is implemented with a tree structure called the *leftist tree* [16]. This data structure shows $Olog(n)$ timing characteristics for insertion and deletion operations in even the most highly unbalanced cases, but it does not provide an easy way to

traverse the tree in a sorted order. The leftist tree is an excellent choice for the simulators because tree-traversal is not needed in a simulator.

6.3.4 Big multiplier results

The sweep-mode simulation results, shown in section 5.2, indicate a 2–4× overhead when $N = 1$; the real-mode results generally show a 4–8× overhead. This is not unexpected because the time required in the sweep mode to deliver a message to an element is assumed to be the same in all simulators; in reality, the CMB-variant simulator has to do more work per message than does the sequential simulator.

We cannot, at this moment, reproduce the same sweep-mode performance comparisons using real multicomputers, because we do not have access to any multicomputers with 2K nodes. We do, however, have access to an assortment of multicomputers of various sizes and vintages that we can use to explore various regions of the result graph. Figure 6.11 contains the timing result for a simulation of the 1376-gate array multiplier from section 5.2. The simulation is run for a duration of $40\mu s$ in simulated time under a 16-node iPSC/2.

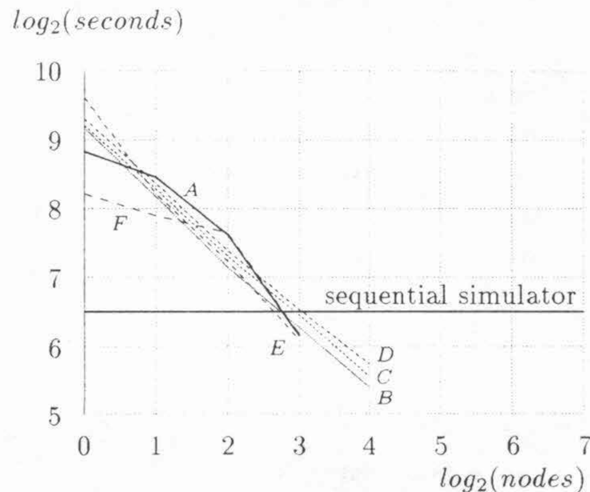

Figure 6.11 A 1376-gate multiplier for $40\mu s$ on an iPSC/2.

Aside from a larger overhead, the real-mode curves generally reflect the upper third of the sweep-mode curves. One consistent characteristic for this and other simulations is a relatively low overhead for the variant F results at $N = 1$. Variant A and F share the

property that messages can be sent eagerly, while message sending in the other variants must wait until a null pointer is returned by a call to `xrecv` — even if the messages are to be sent from a simulator process to itself. Variant F has a lower overhead than variant A because it makes eager only those elements on critical paths, thus allowing messages on non-critical paths to merge. As the simulation becomes more distributed, however, more elements become part of a critical path, and the advantage of variant F disappears.

When $N > 8$, variant A, E, and F fail as more of the eagerly-sent demand and null messages become internode messages and overload the buffering capacity of the message system. The other variants are able to continue because many messages are eliminated by being detained and merged with other messages.

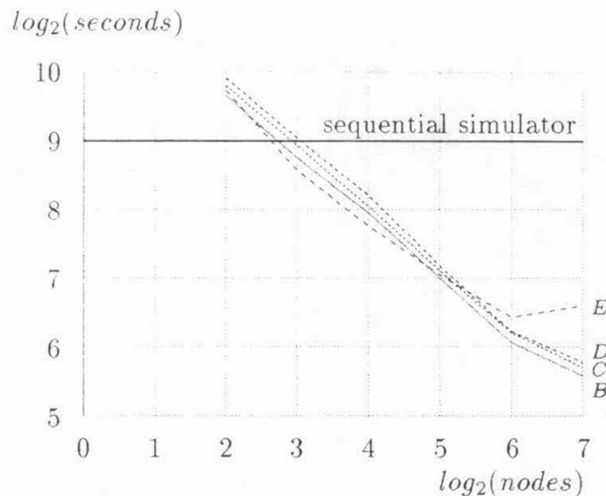

Figure 6.12 A 1376-gate multiplier for $40\mu s$ on an iPSC/1.

Figure 6.12 contains the result of the same simulation on a 128-node iPSC/1. Due to an excess of null messages, variant A and F fail for all N; due to a lack of memory, none of the variants will run when $N < 4$, nor will the sequential simulator run at $N = 1$. (Our iPSC/1 has only one-half megabyte of memory per node, whereas the iPSC/2 has 4 megabytes per node.) The sequential simulator result is an estimate derived from a simulation of a smaller circuit (to be described later).

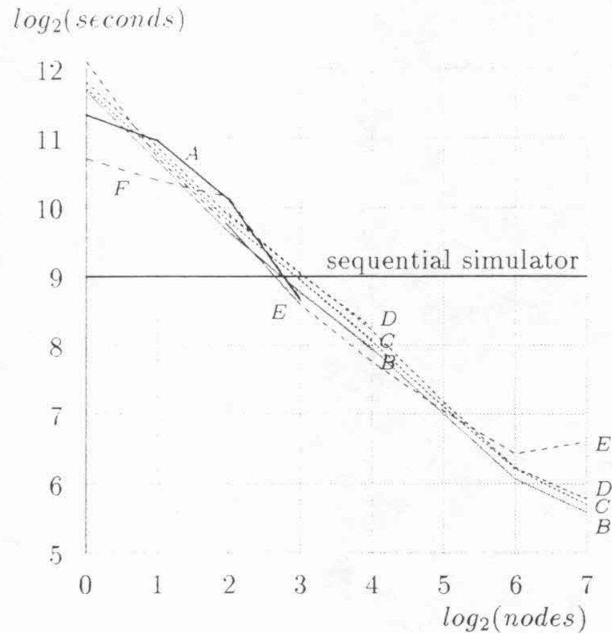

Figure 6.13 Combining the iPSC/2 and iPSC/1 graphs with sequential timing aligned.

The results that we are able to obtain from the iPSC/1 simulation indicate a continuation of the near-linear speedup until $N > 64$, when there are fewer than 22 elements in each node. The total speedup obtained is 64 when the two sets of results are combined in Figure 6.13.

A 64-node Symult 2010 multicomputer allows us to explore a large overlapping portion of these two combined graphs. Since the S2010 nodes are much faster than the iPSC/1 nodes, the simulation interval has been scaled from $40\mu s$ to $100\mu s$; in order for the timing to remain meaningful when $N = 64$. Figure 6.14 matches Figure 6.13 closely, but every variant is able to complete its simulation for every N on the S2010. Variant F resembles variant A because as queuing limits vanish throughout the simulator, the simulator effectively becomes a variant-A simulator. Variant F is a little worse than variant A because it still must produce demand messages in addition to any eagerly sent message. Variant E, however, resembles other variants.

6.3.5 Small multiplier results

Since we do not have a 2048-node multicomputer, it is necessary to experiment with smaller

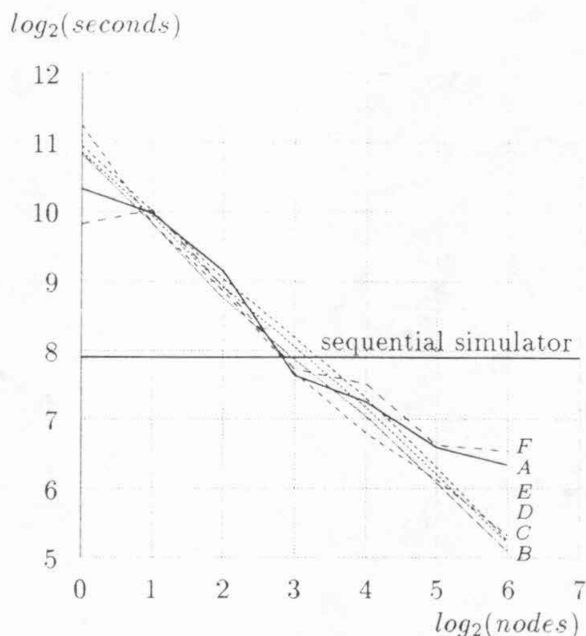

Figure 6.14 A 1376-gate multiplier for $100\mu s$ on a Symult 2010.

circuits to observe the asymptotic effects predicted by the sweep-mode simulation for large N. Figure 6.15 contains the results for the simulation of a 4×4 array-multiplier consisting of 116 logic gates. The iPSC/1 and iPSC/2 simulations were performed over a simulated interval of $100\mu s$. The S2010 simulation was performed over an interval of $400\mu s$ to preserve accuracy when many nodes are used.

Not only is the reduction in slope more visible, differences between various modes are also more apparent. There are 1, 2, and 8 elements per node when all of the nodes in the iPSC/1, S2010, and iPSC/2, respectively, are in use.

Compared to the iPSC/1 curves, the S2010 curves show a steeper slope, a larger overall speedup, and a closer match with the sweep-mode curves. The flattening of the curves for the iPSC/1 is due to the effect of message latency. The average message latency for the iPSC/1 when $N = 64$ is $\approx 3000\mu s$; this is comparable to the $3000\mu s$-per-event processing time of the sequential simulator. The user-mode message latency for the S2010 is $\approx 200\mu s$; this is smaller than the $600\mu s$-per-event processing time.

We can observe the effect of latency by varying latency in the sweep-mode simulation.

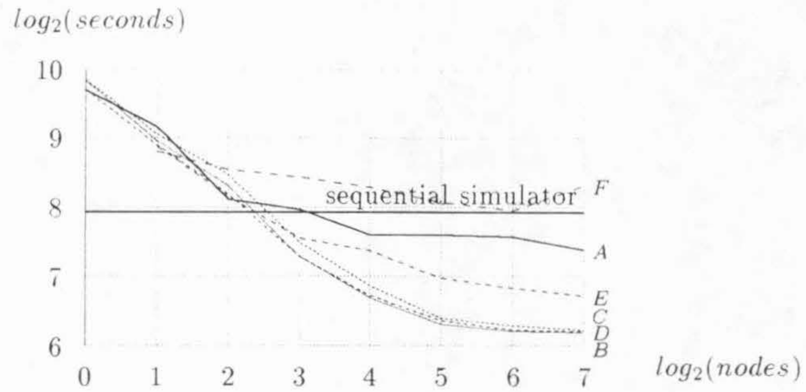

Figure 6.15 A 116-gate multiplier for $100\mu s$ on an iPSC/1.

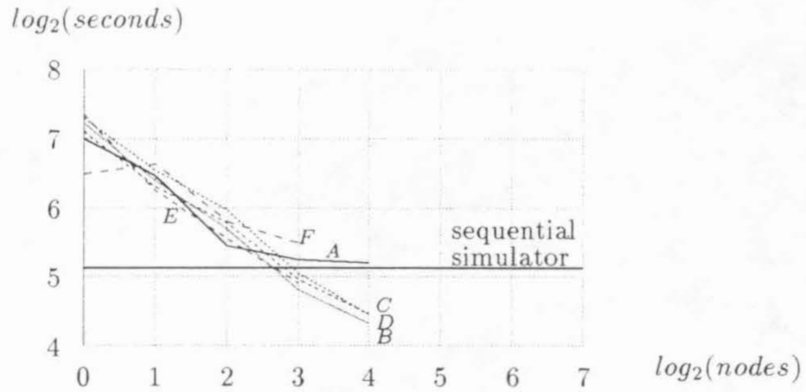

Figure 6.16 A 116-gate multiplier for $100\mu s$ on an iPSC/2.

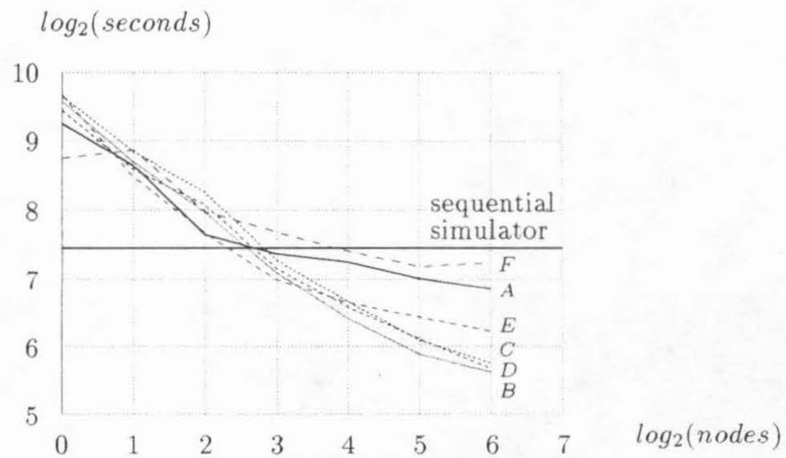

Figure 6.17 A 116-gate multiplier for $400\mu s$ on a Symult 2010.

Figure 6.18 contains two plots, one for $N = 256$ and the other for $N = 2048$. A message sent during a sweep is available to its destination in the following sweep when latency is 0. When latency is non-zero, the message is delayed by an amount equal to the latency. When simulation becomes dominated by latency, time increases linearly with latency.

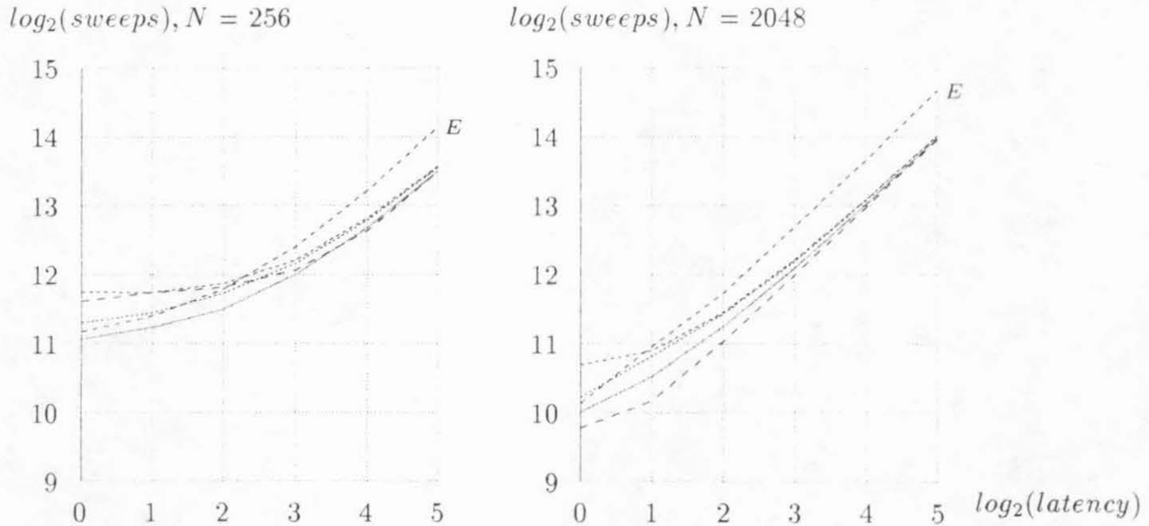

Figure 6.18 Effect of increased latency on simulation performance.

In all of the results that we have shown, the source-driven variants, B, C, and D, are the most robust variants, and they show a larger speedup than the other variants when N is large. The demand-driven variant E is hindered by a large message latency. An idling process may be delayed for two message cycles — send a demand message, receive a normal message — before it can continue. When internode message latency is large, variant E performs poorly. Variant F does better because it becomes variant A when processes are idle more frequently.

6.3.6 Circuit topology *vs.* activity level

A CMB-variant circuit simulator must supply every element input with enough fragments to cover the entire simulation interval. Since its simulation time is only weakly dependent on the content of those fragments, it is more strongly influenced by the static characteristics of the circuit connectivity, such as degree of fan-out, than by the dynamic characteristics

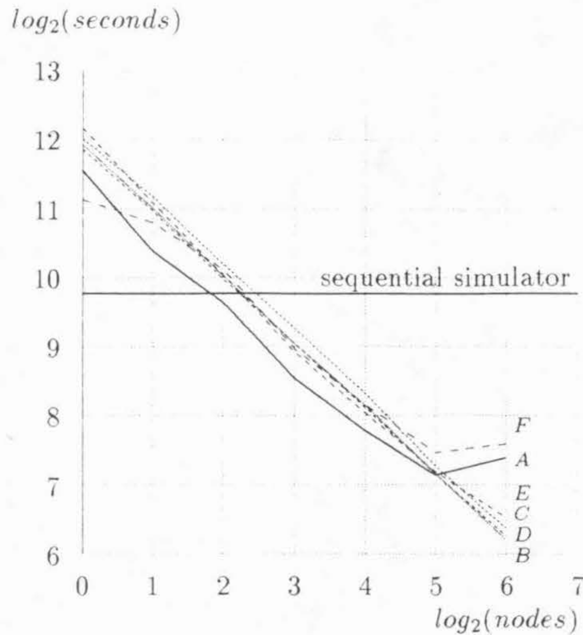

Figure 6.19 A 1376-gate multiplier for $100\mu s$ on a Symult 2010 — fast oscillation.

of the circuit operation, such as number of events produced. A sequential simulator, on the other hand, depends only on the number of events produced.

For example, if a circuit contains a cross-coupled latch, the delay of the gates in the latch determines the number and the span of the fragments produced, and the number of fragments produced determines the simulation time for the CMB-variant simulator. The number of times the latch is used determines the number of events generated in the latch, and the number of events generated determines the simulation time for a sequential simulator.

We can expect the sequential simulator performance to change by a greater degree compared to the CMB-variant simulator if we run the simulation using the same multiplier circuit, but with a different activity level. Figure 6.19 is obtained by driving the array multiplier at an elevated oscillation frequency. Four times as many events are produced, and the time taken by the sequential simulator has increased by a factor of 4. The time taken by the CMB-variant simulators, however, has increased by only a factor of 2.

Since fragments are more likely to carry transitions, the possibility of consecutive fragments merging into a single fragment is reduced. It becomes less profitable for the simulator

to withhold messages. The time taken by variant A has increased by a factor of only 1.5, and variant A performs better than the other variants when N is not too large.

6.3.7 Hybrid possibilities

The CMB-variant simulator implements an algorithm that distributes well, but, like many other algorithms, there are sequential implementations that are more efficient than the concurrent implementation. However, the CMB-variant simulator is unusual in that it is an exact implementation of an algorithm that can be defined recursively — each element simulator can also be a composite simulator. We can view the simulator process on each node as being a composite simulator that simulates the set of elements assigned to that node. We refer to the set of elements, collectively, as a macro element. The circuit simulator becomes one whose elements are not the logic gates but the macro elements; of these one exists in each node.

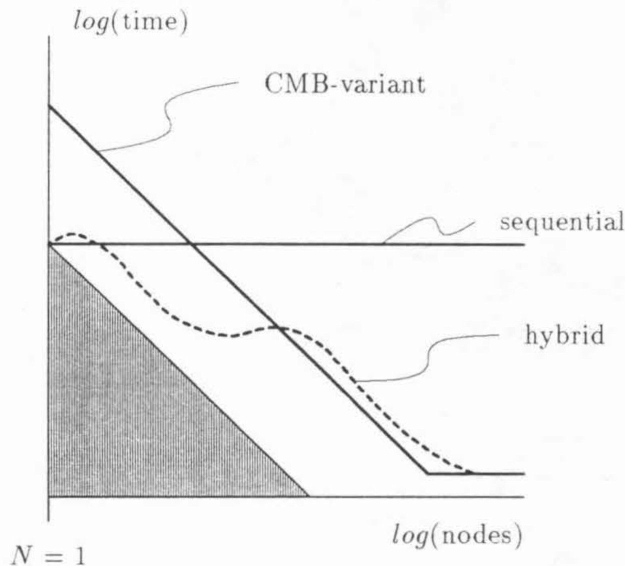

Figure 6.20 Modified Laffer Curve.

Since the elements in a macro element must reside in the same address space, and since their operations must be interleaved, it is a tempting thought that there may be a way to introduce sequential simulator efficiency into the simulation of elements in a macro element. Suppose such a *hybrid simulator* were to exist. When $N = 1$, all logic gates would reside in the same node; the simulator would have the same performance as a sequential simulator. If

N were large, there would be one logic gate per node and the performance would converge to the performance of CMB-variant simulator.

Figure 6.20 depicts a hypothetical performance plot of a hybrid simulator, a sequential simulator, and a CMB-variant simulator. We will call this hybrid-simulator curve the *modified Laffer curve* (in recognition of economist Arthur B. Laffer, who showed that tax revenue is fixed on two ends on the plot of revenue *vs.* tax rate). The quest for the algorithm and for the control over the shape of the curve between these two end points guides the rest of the experimental work, which will be discussed in the next chapter.

Chapter 7 Hybrid Simulators

Section 7.1 Coordinated Sequential Simulator (Hybrid-1)

One way to build a hybrid simulator is to use a modified sequential simulator for each macro element, and to connect the sequential simulators using a CMB-variant simulator. Since a CMB-variant simulator provides coordination for a set of sequential simulators, this hybrid simulator is called the *coordinated sequential simulator* (designated *hybrid-1*). When $N = 1$, hybrid-1 is identical to the sequential simulator, as the modification does not introduce extra work for the simulator when the macro element is a closed system.

A macro element is an open system if any of its element inputs connect to an element output in another node. Macro-element connectivities are handled by the CMB-variant simulator, and macro-element simulators must satisfy the requirements of the CMB-variant simulator: Output state descriptions produced by each macro-element simulator are packed into fragments and sent to the encircling CMB-variant simulator. The CMB-variant simulator distributes the fragments according to the connectivity of the macro elements. When a macro-element simulator receives a fragment, events extracted from the fragment are entered into the event list.

7.1.1 The algorithm

Since asynchronous events can be injected by other macro-element simulators, event order for a macro-element simulator cannot be guaranteed by the the repeated delivery of the earliest event from the event list. The simulator may not be able to consume the event at the top of the list because an event with a smaller time value may yet arrive from another macro element. To avoid a simulation error, we can employ a temporal marker in each macro element to indicate the smallest time value for any future external events. As long as the time of the first event in the event list is less than the marker time, the event can be safely consumed. If the event time will be greater than the marker time, the simulator must wait.

The encircling CMB-variant simulator assures that the time of the next event on any macro-element input is greater than or equal to the time of the macro-element input. The time of a macro-element input is equal to the total span of fragments that have passed through it, and is updated whenever a fragment is received for that input. The minimum macro-element input time is a convenient temporal marker.

Output fragments are produced by a macro-element simulator whenever additional output descriptions are computed. Since elements are strictly synchronized in a sequential simulator, the output of all elements in a macro element are known up to the same simulated time. Thus, the entire state of the macro element can be treated as an atomic property (Chapter 5), and all arcs with the same source and destination nodes can be merged into one arc.

In order to compute the temporal marker, we store the input time of each macro-element input in a special `stopper` event. The stopper is added to the event list along with the other events. When a macro-element input receives a fragment, in addition to injecting new events, it adds the span of the fragment to its stopper time, and it repositions the stopper in the event list. As long as the event at the top of the event list is not a stopper, the macro-element simulator is free to consume the event; when a stopper appears at the top of the event list, the simulator is made to wait for more inputs.

7.1.2 Sorting with a different key

A macro-element simulator derived from a conventional sequential simulator has an effective delay of zero because its event-consumption rules prevent the simulator from producing any output description that has a time value larger than its own minimum input time. A circuit of these macro-element simulators will deadlock unless a set of alternative consumption rules is used to produce a positive delay.

"The event with the smallest simulated time will be delivered first" is merely a convenient consumption rule that satisfies the following correctness conditions for a sequential simulator. When an event is delivered to an element:

1. The event will not need to be recalled, and

2. No future events for the element will have a smaller event time.

We can satisfy both conditions and provide a non-zero delay by sorting events according to the following ordered pair:

$$key = (t_e + d_m, t_e)$$

where t_e is the event time, and d_m (the `m_delay`) is the delay of a minimum-delay path (the *shortest path*) between the destination element of the event and any macro-element output. Macro-element output-events therefore have a d_m of 0. The first member of a key is the dominant member when keys are to be compared.

$$key_1 > key_2 \longleftrightarrow \begin{cases} key_1[1] > key_2[1] \\ key_1[1] = key_2[1] \; and \; key_1[2] > key_2[2] \end{cases}$$

Intuitively, if input events for an element are ordered according to this key, they are ordered in t_e as well, because d_m is the same for all input events of the element. An event whose destination element has an `m_delay` of d_m can be deferred in the event list by d_m amount of time relative to those events for the macro-element outputs because its effects cannot propagate to the outputs before $t_e + d_m$. The effective delay of a macro element is therefore the minimum `m_delay` of its macro-element inputs.

Theorem 7.1: An event produced by an element with a positive delay must have a key that is larger then the key of the event that triggers it.

Proof: Let the delay of the element be δ, the time of the input event be t_e, and the `m_delay` of the element be d_m.

By the definition of element delay, any output event triggered by this input event must have a time value of at least $t_e + \delta$. By the definition of `m_delay`, the destination element of the output event must have an `m_delay` of a least $d_m - \delta$. Therefore, the first part of the key for the output event must be no less than $d_m - \delta + t_e + \delta$, or $t_e + d_m$, which is equal to the first part of the key for the input event.

The second part of the key for the output event is $t_e + \delta$, which is greater than the second part of the key of the input event. Therefore, the key of the output event must be larger than the key of the input event.

Theorem 7.2: Any event appearing at the top of the event list is valid.

Proof: An event must come either from another element in the same macro element or from another macro element. Events from other macro elements are assumed to be correct because the macro-element simulators follow the rules of a CMB-variant simulator.

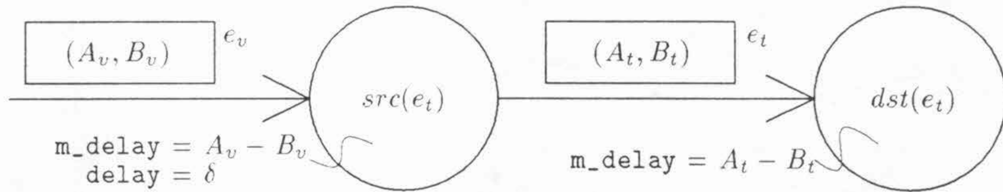

Figure 7.1 An event that invalidates another event.

If the event is produced locally, let the event at the top of the list be e_t, and let (A_t, B_t) be the key of that event. Let e_v be the event that an element consumes to invalidate e_t, and let (A_v, B_v) be its key.

By the definition of a key, $A_t - B_t$ is the `m_delay` of $dst(e_t)$, and $A_v - B_v$ is the `m_delay` of $src(e_t)$. Let δ be the delay of $src(e_t)$. By the definition of `m_delay`, we have the inequality:

$$A_t - B_t \geq A_v - B_v - \delta$$

which we can rearrange into:

$$\delta \geq (A_v - A_t) + (B_t - B_v)$$

We also have $(B_t - B_v) \geq \delta$, because the delay of $src(e_t)$ is δ; and $(A_v - A_t) \geq 0$, because e_t is the event at the top of the event list. The only solution to the inequality above is $(A_v - A_t) = 0$ and $(B_t - B_v) = \delta$.

Since the key of e_t is no greater than the key of e_v, it follows that δ must be zero and that the two events must have the same event time. Since

the ordering of the two events is beyond the ability of the model to resolve, it is correct to assume in this case that e_t is earlier in time, and is therefore valid.

Suppose e_t is the event at the top of the event list, and let the first part of its key be called the *event-list time*. Since all macro-element output events have an m_delay of zero, and since all new events have keys that are at least as large as the key of e_t, the state of all macro-element outputs is known up to the event-list time. The effective delay of a macro element is therefore equal to the delay of the shortest path between any macro-element input and output.

7.1.3 The simulator mechanism

The sequential-simulator discussion in section 6.2 hints that complexities are being moved into the message system of the reactive kernel (the kernel of the light-weight, reactive element processes). When a reactive kernel needs an event, its message system provides the event with the smallest time value of all events in the message system.

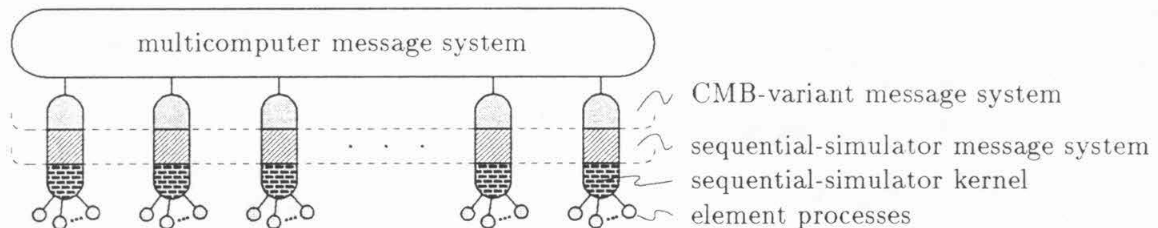

Figure 7.2 Layering in the hybrid-1 simulator.

In hybrid-1, the message system of a sequential simulator is sandwiched between the message system of a CMB-variant simulator and the kernel of the sequential simulator. When the kernel needs an event, its message system provides that event having the smallest key, as long as that event is not a stopper. If it is, the message system waits for the stopper to be relocated. When the message system of the CMB-variant simulator receives more fragments, it moves the stoppers. The hybrid-1 simulator can therefore be constructed by layering reactive kernels.

```
 1  struct { int    (*entry)();
 2           char   *data     ; } PROCESS;

 4  struct { int        pid2  ;
 5           char  msg_body[]; } MESSAGE;

 7  SIM_DATA *simulator_data;

 9  sequential_simulator_main_loop(simp,mesg)
10      PROCESS *simp;
11      MESSAGE *mesg;
12  {
13      PROCESS *proc;

15      simulator_data = (SIM_DATA *)(simp->data);

17      proc = simulator_data->process_table + mesg->pid2;
18      (*proc->entry)(proc, mesg->msg_body);
19  }
```

Listing 7.1 Hybrid-1 main loop.

The kernel of the sequential-simulator main loop can be expressed as the light-weight, reactive-process program shown in Listing 7.1. It returns to its message system for more events. The message-system layer for the sequential simulator (Listing 7.2) takes care of sorting the events and getting external events from the message system of a CMB-variant simulator. The message system of the sequential simulator is also a light-weight reactive process:

```
 1  PROCESS *seqsim;  /* Sequential simulator process structure (only 1) */

 3  sequential_simulator_message_system(msys, sb)
 4      PROCESS         *msys;
 5      STATE_FRAGMENT   *sb;
 6  {
 7      break_state_fragment_into_events(msys,sb);
 8      free_fragment(sb);

10      while(top_of_list_event_is_not_stopper(msys))
11      {
12          (seqsim->entry)(seqsim,get_top_of_list_event(msys));
13      }
14  }
```

Listing 7.2 Hybrid-1 embedded message system.

It returns to the message system of the CMB-variant simulator for a fragment, which it digests into individual events. After that, as long as the event with the smallest time is not a stopper, the message system will remove the event from the event list and deliver it to the sequential-simulator kernel.

7.1.4 The simulator output

Sending only the macro-element output events is not enough to satisfy the requirements for a CMB-variant simulator. Whenever the event-list time has increased, more is known about the outputs, even if no output event has been produced. The rule for eventual delivery requires that null messages be generated.

Like the CMB-variant simulator, several variants of the hybrid-1 simulator have been created, and they are characterized by how and when messages are sent. Eventual delivery is also assured by the same indefinite-lazy evaluation mechanism (not shown in the listings above). Three adjustable parameters are available for the hybrid-1 simulator:

Queue-limiting: Messages are sent when an adjustable limit on the number of queued output events is reached, or when null is returned by `xrecv`.

Demand-driven: Demand messages are sent after an adjustable delay, as measured by the number of successive nulls returned by `xrecv` while a macro-element simulator is waiting for more inputs. Demand messages are sent to the source nodes of the inputs whose stoppers are at the top of the event list. Queued messages for that output addressed by the demand message are sent when a demand message is received.

Eager-message: Each output has a `prompter` event that stores the sum of an adjustable value and the simulated time of the last output action. When a prompter event reaches the top of the event list, messages are sent for that output and the prompter is rescheduled.

7.1.5 Expectation

Tight synchronization between elements in the same computing node greatly reduces the volume of internode messages, especially null messages, by combining internode arcs having common source and destination nodes into one single arc. Tight synchronization, however, can also reduce concurrency. When a simulator process is blocked because of a stopper appearing at the top of the event list, elements that do not depend on the input of that

stopper are also prevented from making progress. Concurrency is reduced because this forces different sub-circuits in the same node to progress at the same rate, and ignores non-strict input conditions in which an element can still make progress when some of its inputs are blocked.

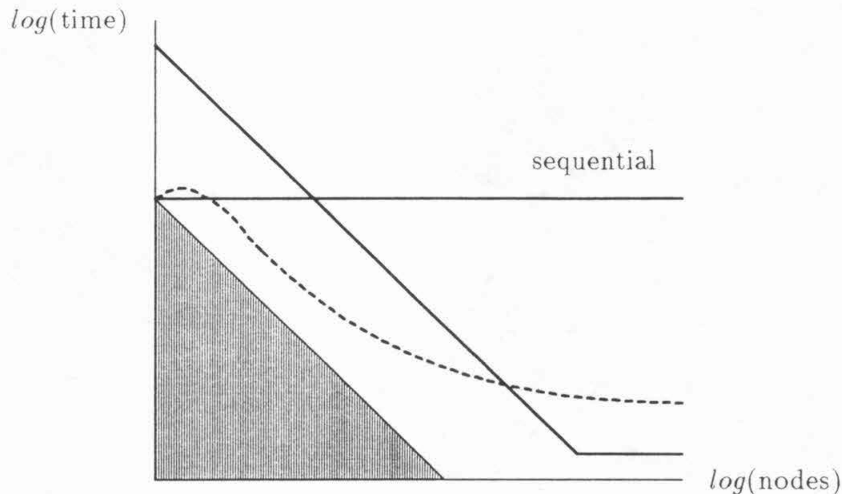

Figure 7.3 Expected performance of the hybrid-1 simulator.

The purpose of this experiment is to construct a simulator that will do as little work as possible at small N rather than be as efficient as the CMB-variant simulators at large N. After all, we can already get CMB-variant-simulator performance by running a CMB-variant simulator. We expect the simulator performance graph to start at $N = 1$ at sequential simulator speed. We expect to see sub-linear speedup due to the lost concurrency, load imbalance, and extra work required to deal with the message system. We then expect the performance to bottom out at a level above the CMB-variant simulator when N is large.

7.1.6 Experimental results

Like the CMB-variant simulator and the sequential simulator, hybrid-1 is also written in the form of a reactive program, making it suitable for sweep-mode simulation; however, a sweep-mode simulator has not been implemented. The real-mode simulator has been implemented, and a 64-node Symult 2010 was used as the primary test vehicle. Although simulation was performed using a multitude of simulation parameters, only a handful will be shown because related variants produce similar results. The variants are:

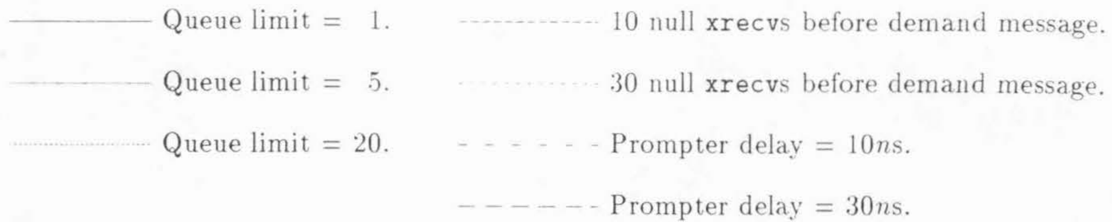

Figure 7.4 contains the simulation result of a 14×14 array-multiplier running on a 64-node S2010 for $100\mu s$ simulated time. It is shown alone (left) and superimposed over the CMB-variant result (right).

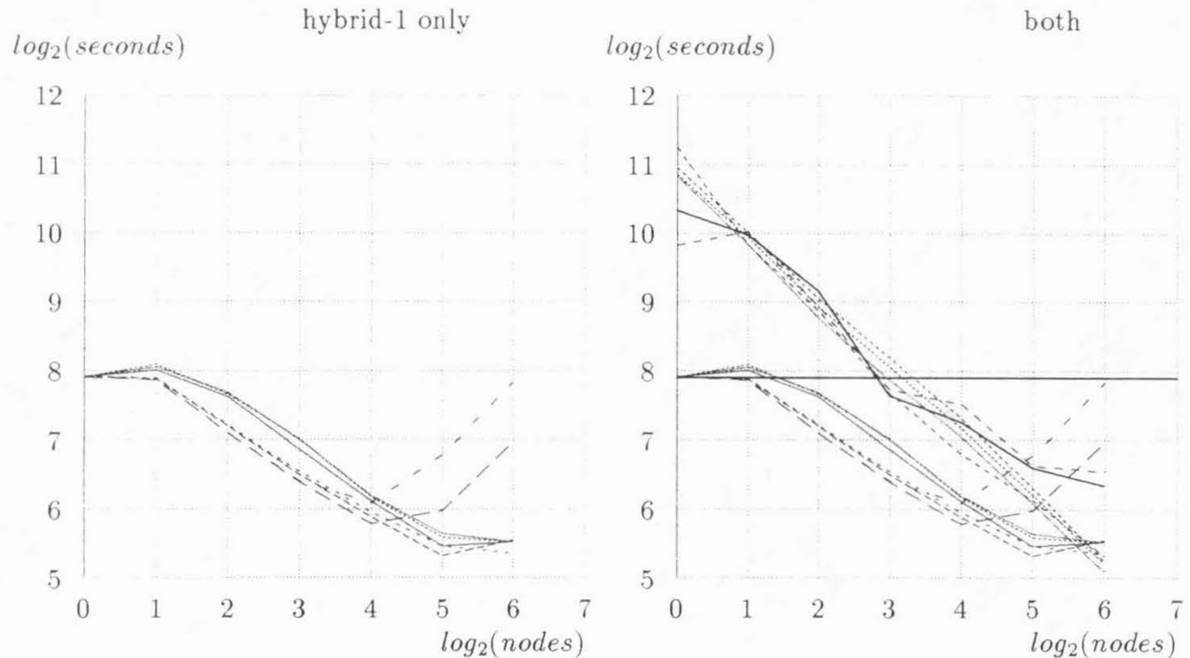

Figure 7.4 A 1376-gate multiplier for $100\mu s$ on a Symult 2010.

The general characteristic of these curves matches our expectation. In the multiplier example, the extra work that the simulator has to do and the difficulty it has in subdividing the multiplier for load balancing result in no speedup from $N = 1$ to 2. For larger N, the curves show a slope of $\approx 1/2$ until $N = 32$, where the curves level out. Between $N = 32$ and 64, the curves cross over those of the CMB-variant simulator. The demand-driven modes perform consistently better than the queue-limiting modes. The eager-message modes perform well for small N, but they bend upward for large N due to an excess of null messages. The more eager of the two curves bends upward sooner than the less-eager one.

Due to the combining of arcs, hybrid-1 curves are strongly influenced by element distribution only when N is large. Figure 7.5 contains results of simulation using randomized element placement. Compared to Figure 7.4, the CMB-variant curves are shifted upward uniformly for all N, and the hybrid-1 curves are bent upward when N is large. The hybrid-1 curves show little change when N is small.

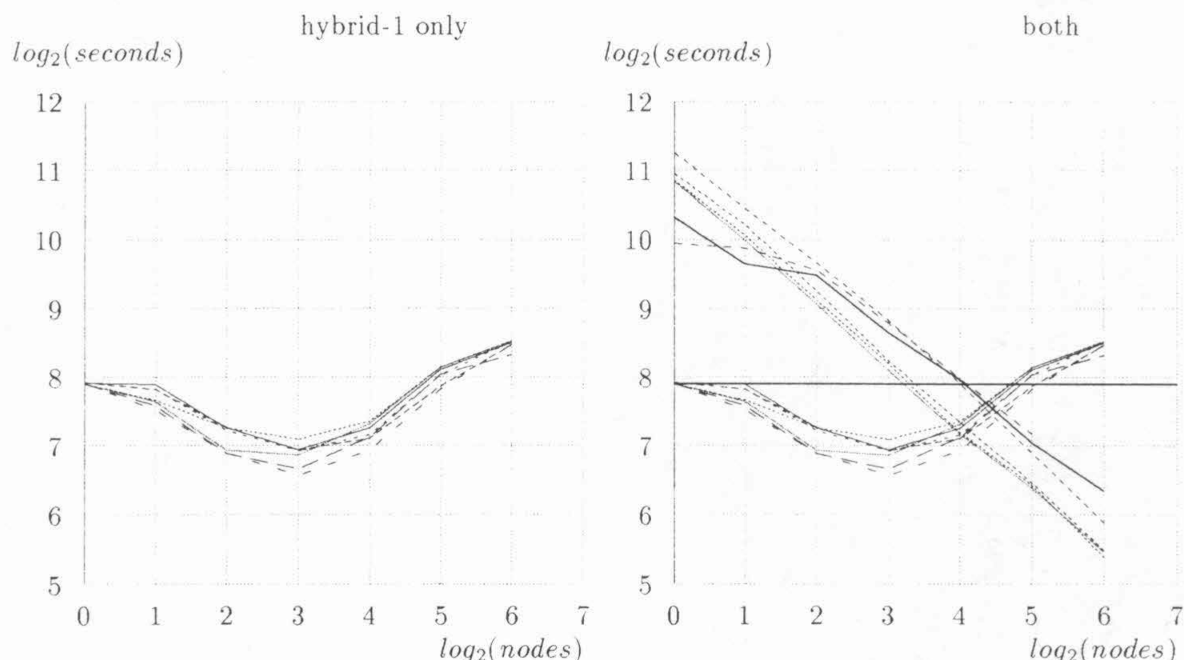

Figure 7.5 A 1376-gate multiplier for $100\mu s$ on a Symult 2010 with random placement.

Since one end of the hybrid-1 curves is pegged to the sequential simulator time, we can also expect a larger change for the hybrid-1 simulator than for the CMB-variant simulator when we increase the circuit activity level. Figure 7.6 contains the results of simulation using the same multiplier circuit that is operated at a higher oscillation frequency. The hybrid-1 curves are shifted upward by two octaves, while the CMB-variant curves are shifted only by one octave. A high activity level is more favorable to the CMB-variant simulator because fewer of the messages are null messages.

Results from the multiplier example in this chapter, and better results from other circuits to be shown in Chapter 8, have confirmed that the hybrid-1 simulator is working and performing to our expectation. Our next step is to go beyond the limitations of the

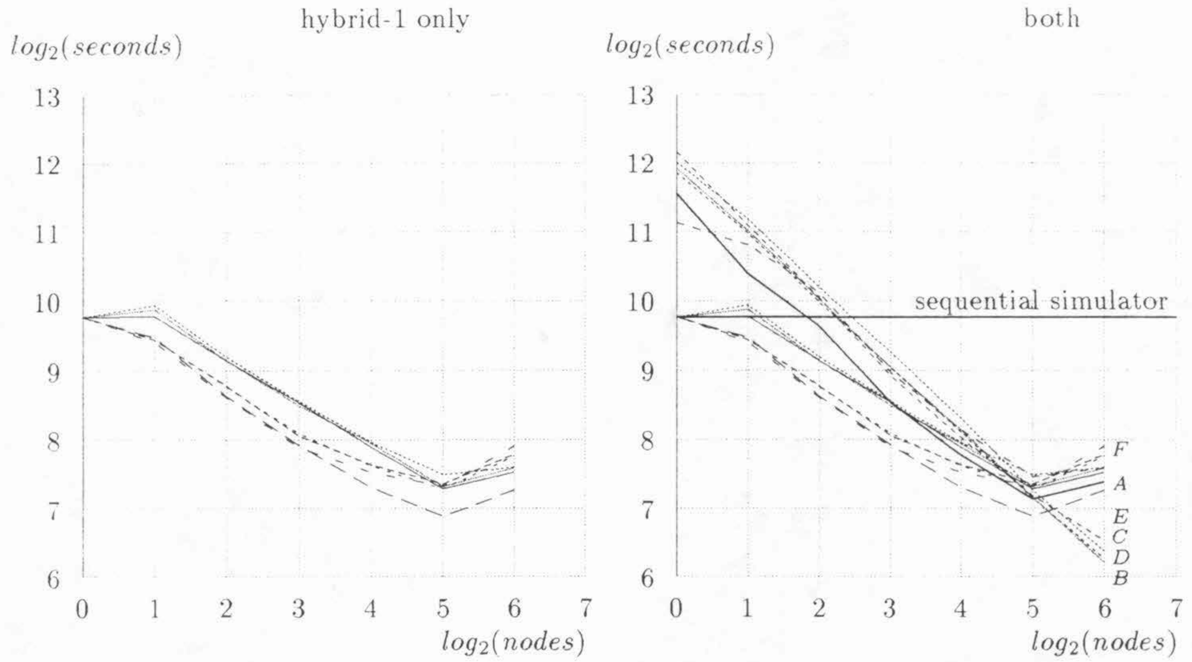

Figure 7.6 A faster oscillating 1376-gate multiplier for $100\mu s$ on a Symult 2010.

hybrid-1 simulator to construct a new hybrid simulator that will converge to the CMB-variant simulators when N is large.

Section 7.2 Progressive Hybrid Simulator (Hybrid-2)

The hybrid-1 simulator cannot achieve CMB-variant performance at large N because potential concurrency is lost when non-strict conditions are ignored and elements in a macro element are synchronized. Two separate mechanisms are used to recover the lost concurrency: First, when an input of an element becomes blocked, it must be allowed to continue if it can still make progress (due to a non-strict input condition). Second, when some elements are blocked, we must allow those that are not blocked to continue ahead of the blocked elements.

When a stopper appears at the top of the event list, elements connected to the input of the stopper may be blocked. Since hybrid-1 macro elements are simulated by sequential simulators, when an element in a macro element becomes blocked, the entire macro element is blocked. When an element becomes blocked in hybrid-2, the macro element is, in effect, reorganized by moving the blocked element out of the macro element. More blocked elements may result due to arcs leading from the blocked element to the new macro element. When only unblocked elements remain, however, the macro-element simulator can continue to make progress. When a blocked element has received more inputs and becomes unblocked, it is put back into the macro element.

To take advantage of non-strict input conditions, stoppers in hybrid-1 are replaced by blocker events in hybrid-2. A blocker appearing at the top of the event list does not cause the simulator process to stop; instead, it is delivered like a normal event. For every blocker, there is a matching anti-blocker; it has the same simulation time as the blocker and they annihilate each other in the simulator. Macro-element inputs produce both blockers and anti-blockers. Whereas the hybrid-1 simulator relocates the stopper as more state fragments are received, the hybrid-2 simulator instead adds an anti-blocker with a time value equal to the previous blocker, adds any events carried by the fragment, and adds a blocker with the time equal to the new time of the hybrid-1 stopper.

When an element receives either a blocker, an anti-blocker, or a normal event, the element determines whether it is blocked. It is not blocked if all of its inputs are unblocked or if its remaining unblocked inputs contain a non-strict input condition; it is blocked otherwise. When an unblocked element becomes blocked, it sends a blocker with a time equal to the current input event. When a blocked element becomes unblocked, it sends an anti-blocker with a time equal to the previous blocker.

In a hybrid-2 simulator, when N is small, most of the element inputs are not blocked, and the simulation takes on the characteristics of a hybrid-1 simulator. When N is large, many of the element inputs are blocked, and the simulation produces the efficiency of a CMB-variant simulator. However, one clear disadvantage of hybrid-2, compared to hybrid-1, is that internode arc merging is no longer possible, and the simulator is potentially more sensitive to element placement.

7.2.1 The mechanism

```
 1  struct EVENT {  int    e_type;      /* type of the event.         */
 2                  int input_id;       /* id of the element input.  */
 3                  int       time; } ; /* time of the event.         */

 5  generic_gate(pp,ep)
 6      PROCESS *pp;
 7      EVENT   *ep;
    {
 9      if(ep->time < element_time(pp)) ep->time = element_time(pp);

11      set_input_bits(pp,ep);
12      compute_state_and_blockage(pp);

14      if( was_blocked(pp) && !is_blocked(pp)) add_anti_blocker(pp,ep->time);
15      if( old_output (pp) !=  new_output(pp)) add_output_event(pp,ep->time);
16      if(!was_blocked(pp) &&  is_blocked(pp)) add_blocker     (pp,ep->time);

18      save_new_state(pp);
19      free_event(ep);
20  }
```

Listing 7.3 Generic logic-gate handler for hybrid-2.

A sample element entry function appears in Listing 7.3. In addition to the usual `input_id` and `time` fields, the hybrid-2 event structure also contains an `e_type` field to distinguish among normal events, blockers, and anti-blockers. Since non-strict input conditions are

utilized, it is now possible for an element to receive events with a time value smaller than the time of the element. These events are for inputs that were previously blocked, but the element was able to progress further because a non-strict input condition was present. These events do not contribute to the operation of the element, other than to determine the current input state of the element. Therefore, when such an event is received, its event time is simply set to the element time (9) before it is processed like other events.

Each element input contains a pair of variables: One indicates the state, the other indicates blockage. Each output contains two pairs of variables, one for the old state and blockage, and one for the new state and blockage. When an event is received by the process, the `set_input_bits` function is called to set or clear the affected bits in the input structure of the element. The new output state and blockage are then computed from the new input state and blockage (12). If the element has become unblocked due to the event (14), an anti-blocker is sent. If the element has changed state (15), a normal event is sent. If the element has become blocked (16), a blocker is sent. The ordering of lines 14–16 assures that the event following a blocker is an anti-blocker.

The sequential-simulator main loop, the kernel to these element processes, tests the blockage flag before and after an entry function is called; blocked elements are separated from unblocked elements by treating them differently. Listing 7.4 is the kernel written as a heavy-weight reactive process:

```
 1  sequential_simulator_main_loop()
 2  {
 3      MESSAGE *mesg;
 4      PROCESS *proc;

 6      mesg = get_next_event();
 7      proc = process_table + mesg->pid2;

 9      if(!blocked(proc))
10      {
11          (*proc->entry)(proc, mesg->msg_body);

13      } else
14      {
15          if(event_time(mesg) > element_time(proc))
16          {
17              queue_event(proc,mesg);
```

```
19          } else
20          {
21              (*proc->entry)(proc, mesg->msg_body);
22              if(!blocked(proc)) move_queued_events_back_to_event_list(pp);
23          }
24      }
25  }
```

Listing 7.4 Hybrid-2 main loop.

When an event is returned from the message system (which contains the event list), the main loop identifies the receiver of the event (7) and checks its blockage flag (9). If the element is not blocked, it is in the sequential-simulator domain and the event is delivered to it as if it were in a normal sequential simulator (11).

If the element is blocked, the main loop checks its readiness to consume the event. The event cannot be consumed if its time is larger than the time of the element. The element lacks information about the future state of its blocked inputs necessary to consume an event that arrives at a future time. The event is queued for the element (17). If the event time is less than or equal to the element time, the element has enough information to consume the event, and the event is sent to the element (21). If the element is now unblocked, its queued events are moved back into the event list to be delivered again for the element.

Queued events cannot be delivered directly to the element when the element becomes unblocked because they are ones that arrived while some inputs of the element were blocked. There may be events for the blocked inputs that have yet to arrive and that need to be delivered in the proper order (with respect to the queued events) when the element becomes unblocked. Moving all queued events back into the event list is inefficient when the queue is long and when moves have to be done frequently. The actual implementation of the hybrid-2 simulator contains an elaborate mechanism for minimizing wasted efforts, and this accounts for the largest difference between the hybrid-2 presented here and the actual implementation.

7.2.2 Experimental results

Like the other simulators, hybrid-2 is written in the form of a reactive-process program, making it suitable for sweep-mode simulation; but, as in the case of hybrid-1, a sweep-mode simulator has not been created. Figure 7.7 contains the simulation results of a 14×14 array-multiplier running on a 64-node S2010 for $100\mu s$ simulated time. It is shown alone (left) and superimposed over both the CMB-variant result and the hybrid-1 result (right).

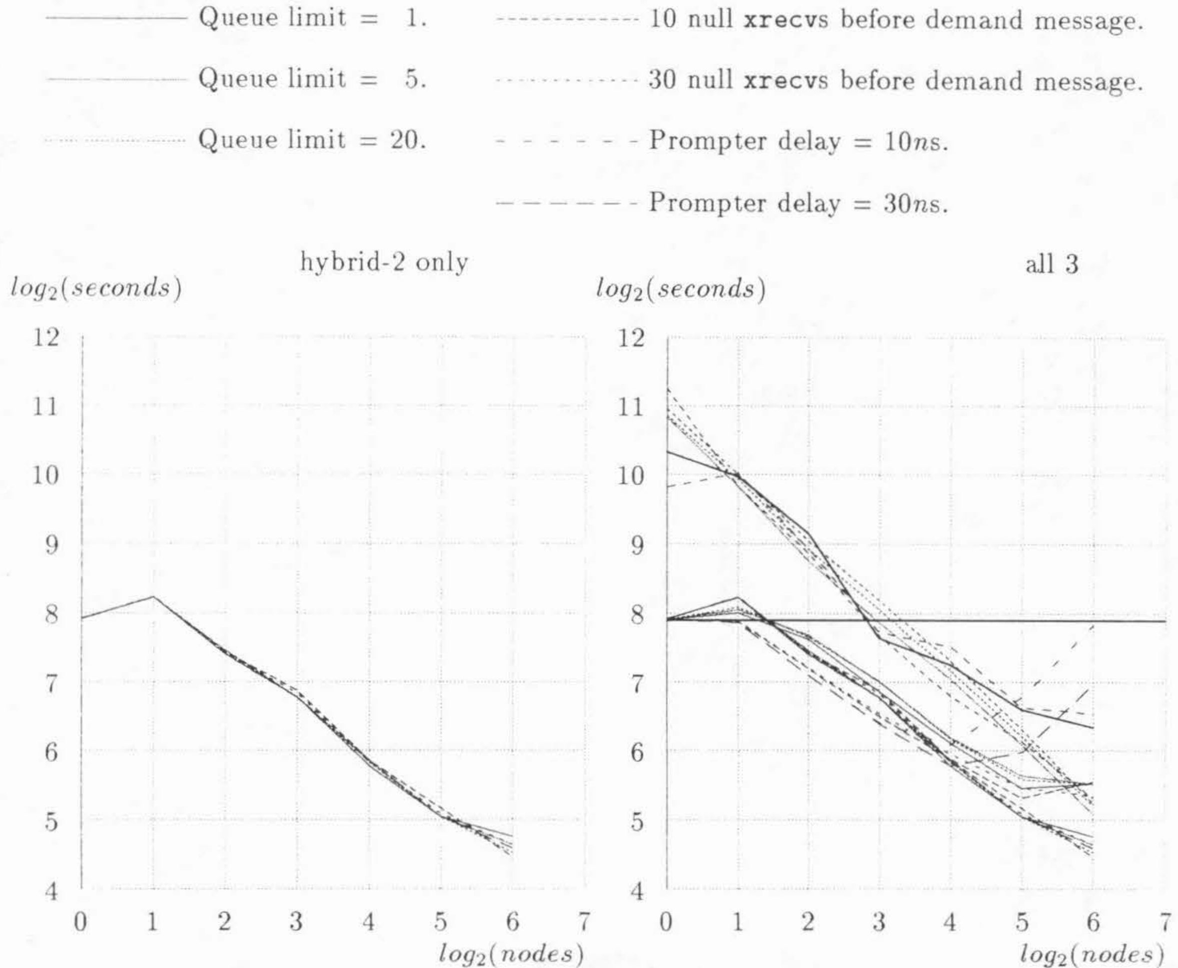

Figure 7.7 A 1376-gate multiplier for $100\mu s$ on a Symult 2010.

The most noticeable difference between hybrid-1 and hybrid-2 curves in this graph is that whereas hybrid-1 curves level off at large N, hybrid-2 curves keep going down. Hybrid-2 curves start out very much like hybrid-1 curves because most of the elements in the hybrid-2 simulators are running under the hybrid-1 mode. As more and more nodes are used in

the simulation, hybrid-1 element simulators start to become idle more frequently, and their curves start to level off. In the hybrid-2 simulator, instead of becoming idle, more of the elements enter the CMB-variant mode to provide additional speedup over hybrid-1.

The other remarkable aspect of hybrid-2 curves is that they are all very much alike until that point where most of the hybrid-1 curves level off. It is after this transition point that progress-promoting actions begin to dominate, and a variety of different performance results are produced, depending on the properties of the progress-promoting action in use.

The hybrid-2 curves appear to converge toward the CMB-variant curves, but nothing conclusive can be deduced from this graph because a 64-node machine lacks sufficient nodes to demonstrate this effect. The convergence is much more obvious when elements are placed randomly. Placement has a much stronger effect on the hybrid-2 simulator than it does on the hybrid-1 simulator because random element placement greatly increases the number of internode arcs for the hybrid-2 simulator.

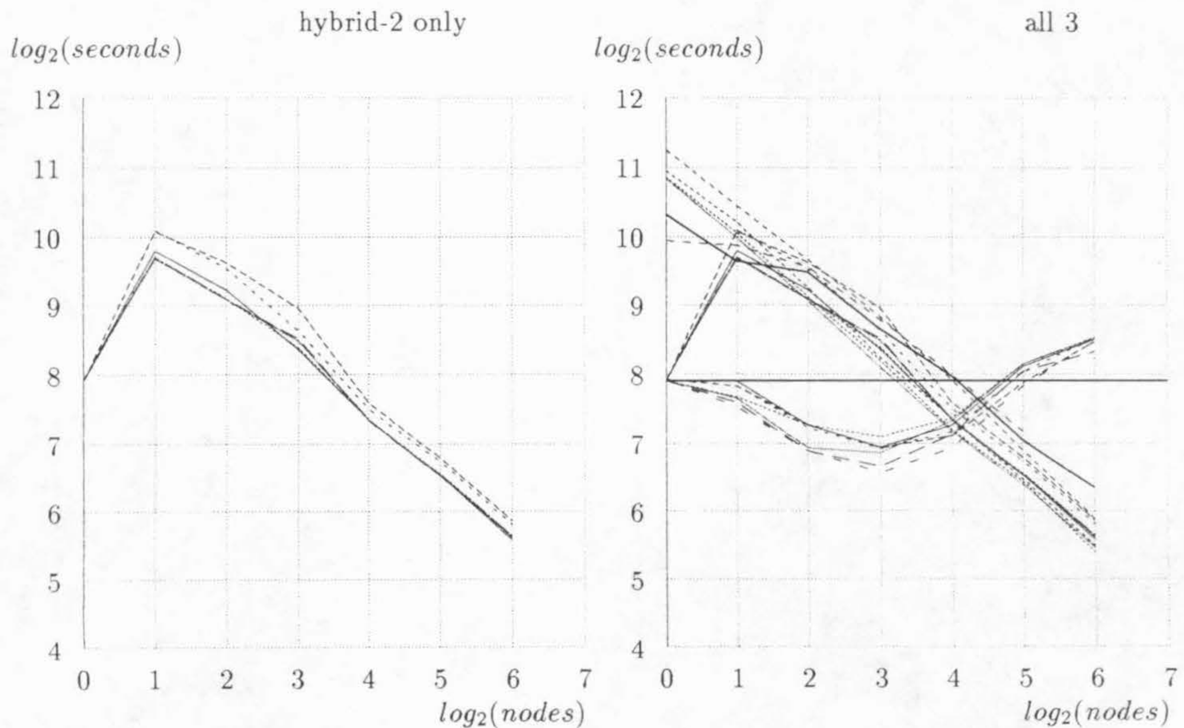

Figure 7.8 A 1376-gate multiplier for $100\mu s$ on a Symult 2010 with random placement.

Figure 7.8 shows the result of random element placement (same placement for all simulations shown in this graph). The hybrid-2 curves converge immediately to the CMB-variant curves at $N = 2$. Reduction in internode null messages by bundling internode arcs allows the hybrid-1 simulator to show a small speedup at small N.

Convergence is also more evident when we increase the circuit activity level. Figure 7.9 shows the results of simulating the multiplier with enhanced activity level. Convergence begins at a smaller N because the sequential-simulator time is now closer to the CMB-variant time when $N = 1$. The hybrid-2 curves start out closer to the CMB-variant curves, and they converge to the CMB-variant curves at $N = 32$.

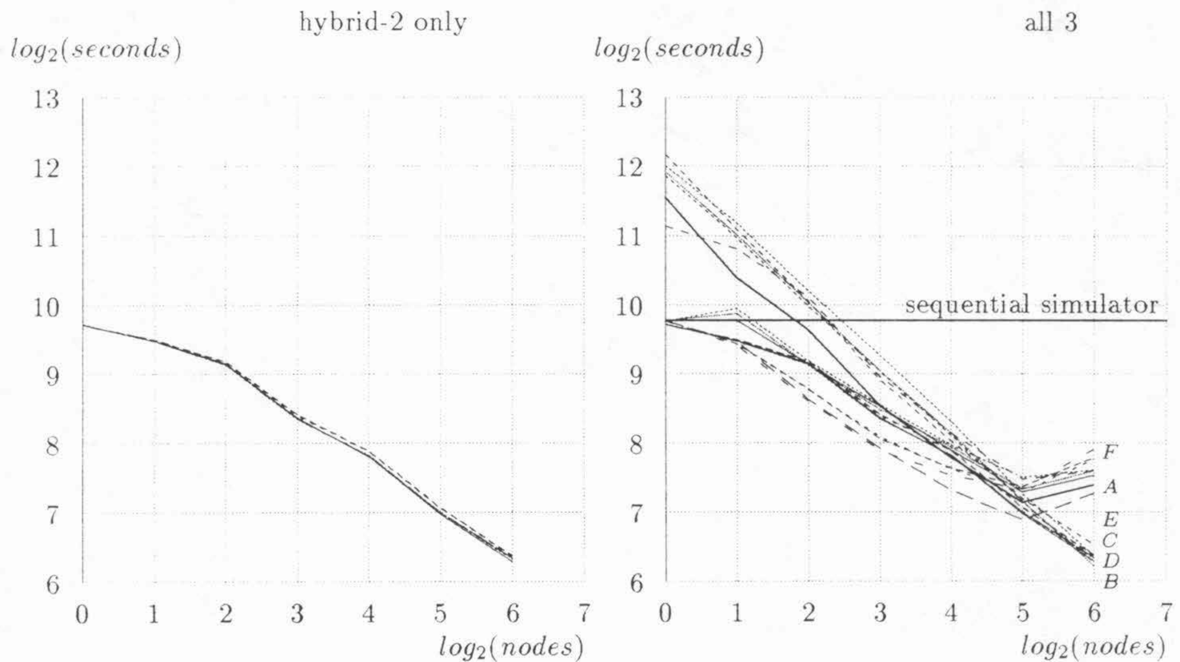

Figure 7.9 A faster-oscillating 1376-gate multiplier for $100\mu s$ on a Symult 2010.

Although we do not have a larger machine for looking at cases where there are fewer elements per node, we can reduce the number of elements per node by using smaller test circuits. We tested a 4×4 array-multiplier that contains 116 gates. At $N = 64$, there are no more than two gates in each node.

The CMB-variant curves diverge wildly; some of them do better than the hybrid-2

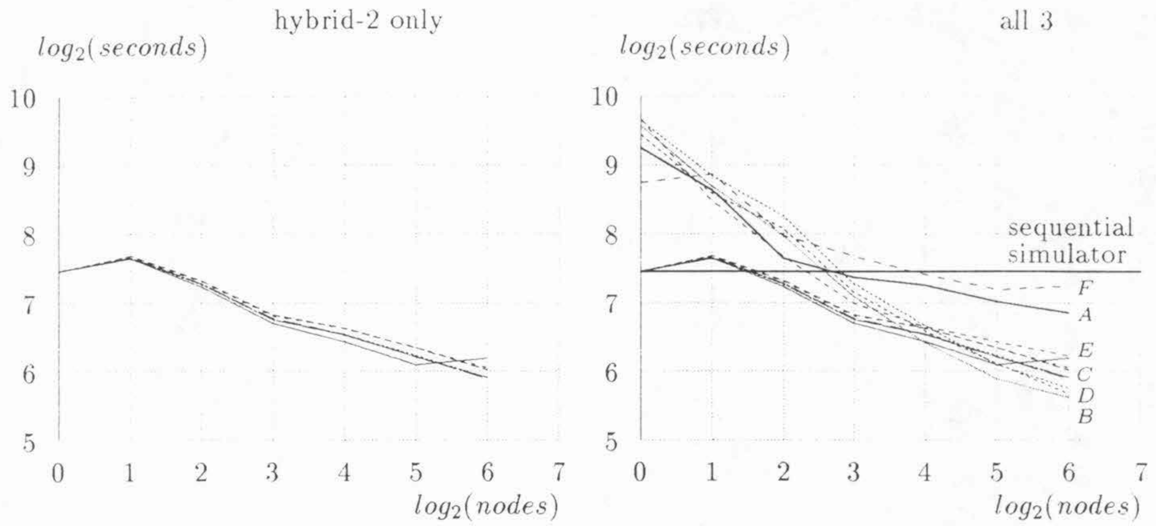

Figure 7.10 A 116-gate multiplier for $400\mu s$ on a Symult 2010.

curves and some do worse. Overall, the hybrid-2 curves seem to follow the better CMB-variant curves.

Chapter 8 Additional Performance Results

This chapter summarizes the simulation results of a few selected circuits that were used in this research. They are generally presented in the following order:

1. Description of the circuits.
2. Sweep-mode simulation results on an emulated multicomputer.
3. Real-mode simulation on a Symult 2010 with systematic element distribution.
4. Real-mode simulation on a Symult 2010 with random element distribution.
5. A few sets of real-mode simulation on smaller circuits of the same type.

Each set of real-mode simulations contains results from running the CMB-variant simulator, the hybrid-1 simulator, and the hybrid-2 simulator. Results from other multicomputers are similar and are not shown.

Section 8.1 2-D Clock Network

8.1.1 Description

A clock network is an arbitrarily extensible array of logic gates that oscillates when properly initialized. The frequency of the oscillation is determined by local characteristics, and the phase at any node in the network is locked to the phase of the adjacent nodes. A clock network can be used to provide synchronous communication for an arbitrarily large, bounded-degree multicomputer network.

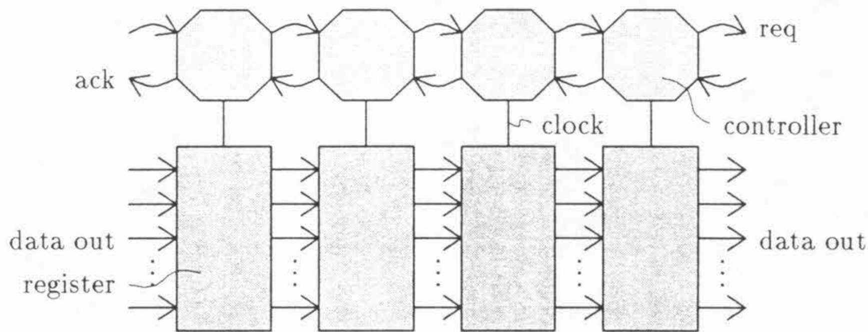

Figure 8.1 A FIFO consisting of 4 units.

A clock network is a generalized self-timed FIFO circuit. As shown in Figure 8.1, a FIFO is made of a number of FIFO units connected into a chain; a FIFO unit contains a controller and a register. The registers in a FIFO are connected in a chain via their data inputs and outputs; the controllers are connected via their request and acknowledge signals. Each controller provides a clock signal to enable and disable the latches in its register. The acknowledge and request signals allow the controllers to determine when the FIFO unit immediately preceding it has data for it, and when the FIFO unit immediately following it has taken the data from it.

Each FIFO unit leads but is never more than a half cycle ahead of the following unit, and lags but is never more than a half cycle behind the preceding unit. Thus, if registers were computers and register-to-register links were communication channels, the data one computer latches in at its kth clock tick is the data put out by the preceding computer at

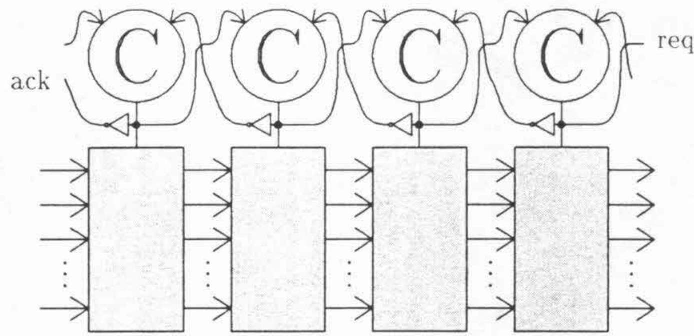

Figure 8.2 A C-element FIFO consisting of 4 units.

that computer's kth clock tick. With a little extra delay, synchronous communication can also take place in the reverse direction.

A simple FIFO control can be constructed using a C element and an inverter. A C element is a state-storage device such that when all of its inputs are high, the output becomes high; when all of its inputs are low, the output becomes low; and the output remains unchanged otherwise. In the FIFO shown in Figure 8.2, the output of a C element is connected to an input of the C element in the following unit. The inverted output of a C element is connected to an input of the C element in the preceding unit. The output of the C element is also used as the clock to the register.

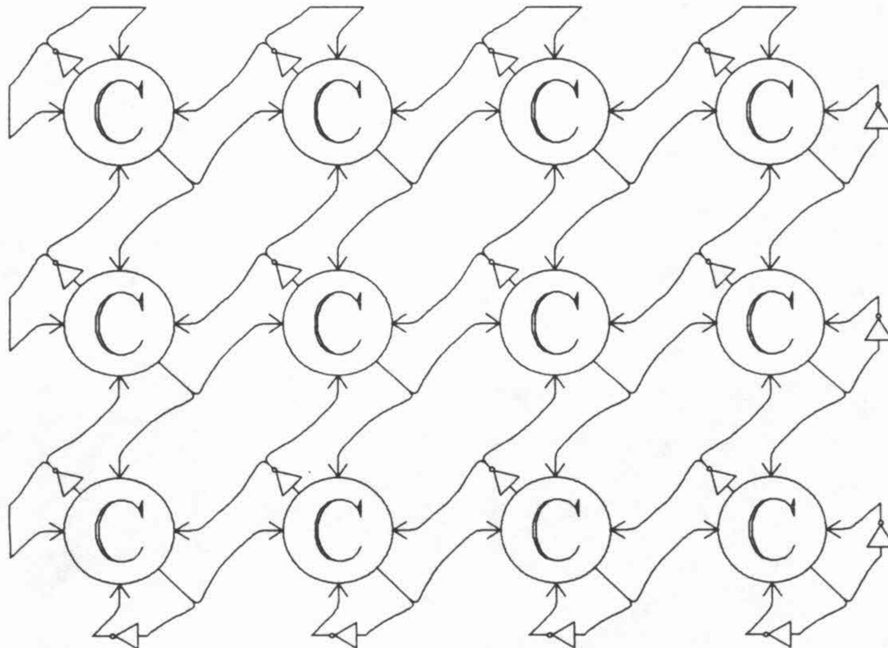

Figure 8.3 A 3×4 array of self-oscillating FIFO units.

The FIFO structure can be extended to a higher dimension by cross-connecting a set of FIFO controls with another set of FIFO controls. Figure 8.3 contains a two-dimensional array of 12 FIFO units with the registers omitted. The edges are terminated in such a way that the array will oscillate. This is essentially the same network that is used in the clock-network simulation, except that each 4-input C element is replaced by a 12-gate circuit. The circuit in Figure 8.3 has 151 gates.

8.1.2 Sweep-mode results

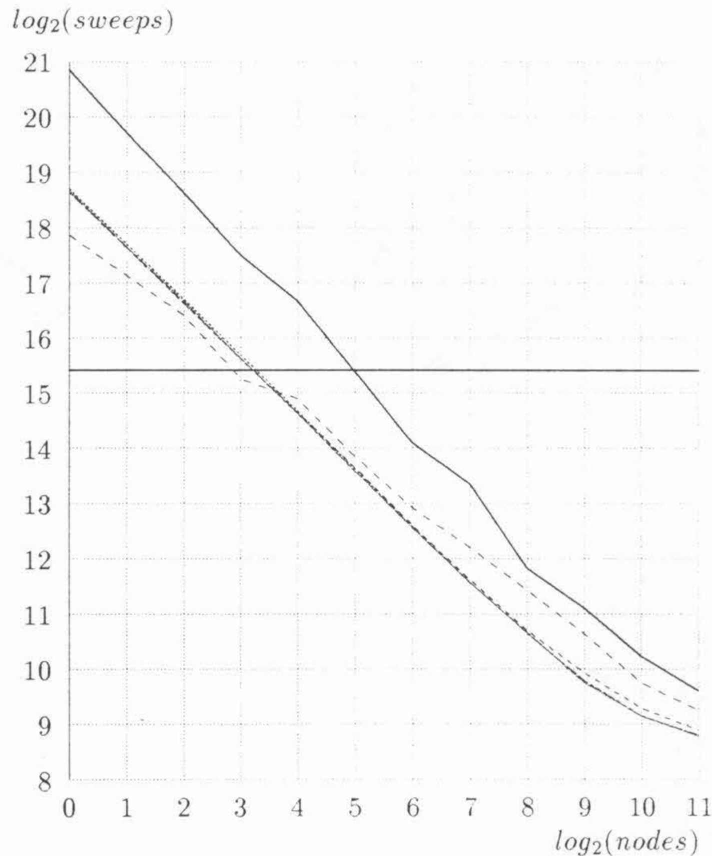

Figure 8.4 Sweep-mode CMB-variant simulation of an 1841-gate clock network.

Figure 8.4 contains the sweep-mode results of an 8×16 clock-network containing 1841 logic gates. The speedup is linear until there are fewer than 4 elements in each node. The null message overhead is a little larger than 8 at $N = 1$, and the crossover occurs between $N = 8$ and $N = 16$. Unlike the multiplier example we used in previous chapters, the clock network shows a much greater difference between the most-eager variant and the lazier variants. This

is typical of circuits with many tight loops, where unnecessary null messages can persist as they travel around the loops. The lazier variants annihilate such null messages to achieve an improved performance over the most-eager variant.

Also, unlike the multiplier example, load balancing is simple because a clock network shows a steady and uniform activity level at every part of the circuit. Although the CMB-variant simulators are relatively insensitive to the effect of load balance and activity level, the hybrid simulators are more favorably influenced, as we can see in Figure 8.4.

8.1.3 Real-mode results

The performance at $N = 1$ and the linear speedup for most of the lazier CMB variants fit the sweep-mode prediction well. The real-mode curves differ from the prediction in that the eager CMB-variant curve is not uniformly worse over all N, and the curve for the adaptive demand-driven variant worsens more rapidly than predicted. These two CMB variants are not robust in circuits that contain many closed loops where null messages can circulate, because the persistence of the null messages depends on run-time conditions such as congestion and order of message arrival. As a consequence, the result of the simulation can vary significantly from run to run, but when N is small, the behavior is more restricted, and the prediction of the sweep-mode simulation prevails.

The hybrid-1 and hybrid-2 curves are similar to those of the multiplier circuit, except these curves show a greater speedup due to better load balance for the clock network. Thus, these curves are more similar to those of the multiplier with an enhanced activity level — there is no significant initial penalty at $N = 2$. The activity level for this multiplier is more uniform because a new wave of activities is injected into the multiplier before old ones have completed. The hybrid-1 curves flatten and bend upward between $N = 16$ and 32, while the hybrid-2 curves continue straight down as they close in toward the CMB-variant curves.

The next set of graphs shows the effect of randomized element distribution. The CMB-variant curves have shifted very little, but the hybrid-1 curves become much shallower, and the hybrid-2 curves show the characteristic upward hump for random element distribution.

Real-mode results for an 8×16 network

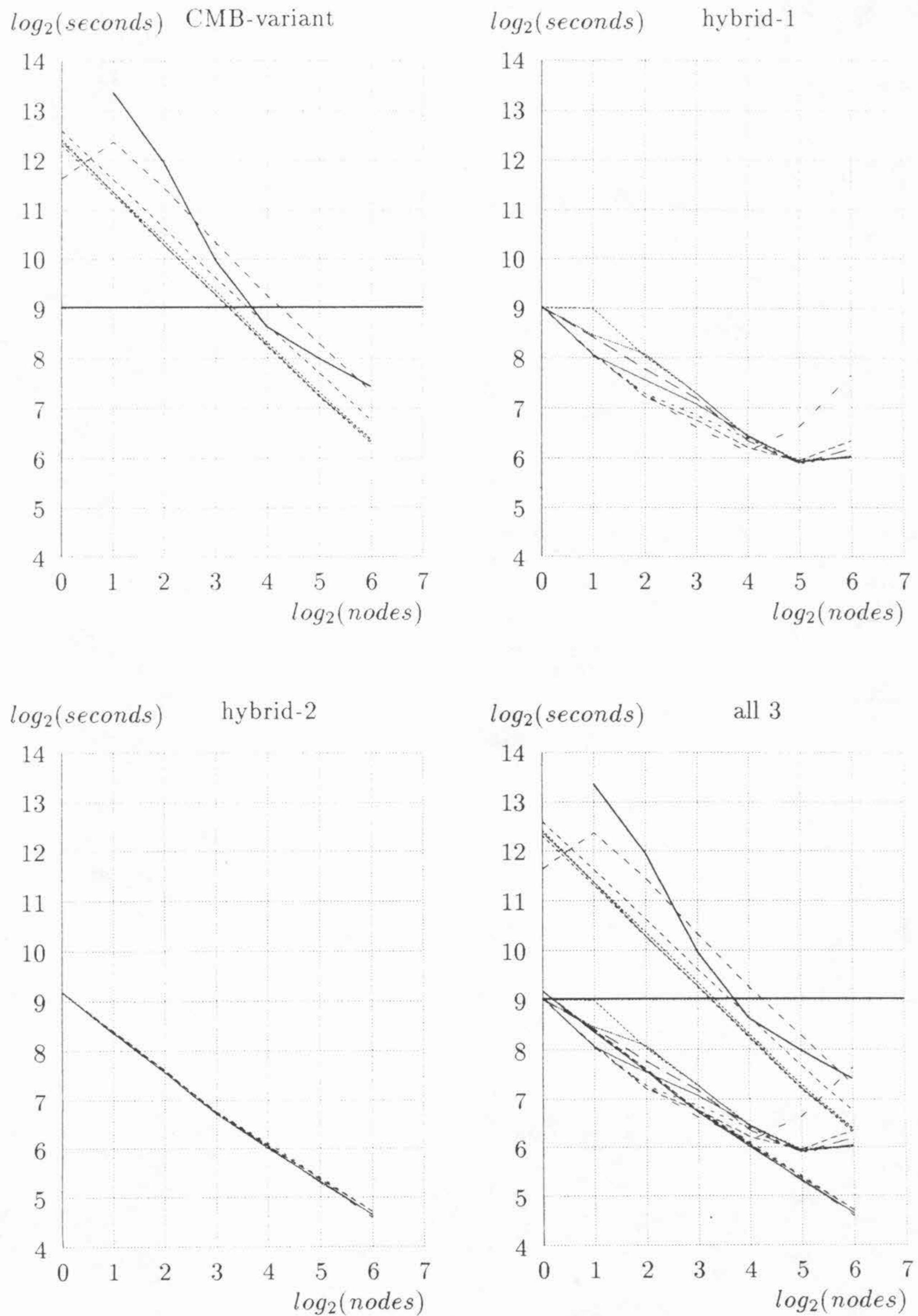

Figure 8.5 An 1841-gate clock network for $50\mu s$ on a Symult 2010.

Figures 8.4 and 8.5 show the results in regions where there are many more logic elements than nodes. The three additional sets of simulation results use progressively smaller clock circuits; the last one has, on average, one logic gate per node for $N = 64$. As the number of gates is reduced, speedup achieved by the hybrid simulators is reduced because the advantage that can be obtained from running sequential, macro-element simulators decreases. The CMB-variant simulators, which reflect the ratio of null messages and event messages, show very little change relative to the sequential simulator.

The lazy CMB-variants are hardy and robust simulators: They show good speedup relative to themselves all the way down to 1 element per node in a fashion consistent with the sweep-mode prediction.

Real-mode results with random element distribution

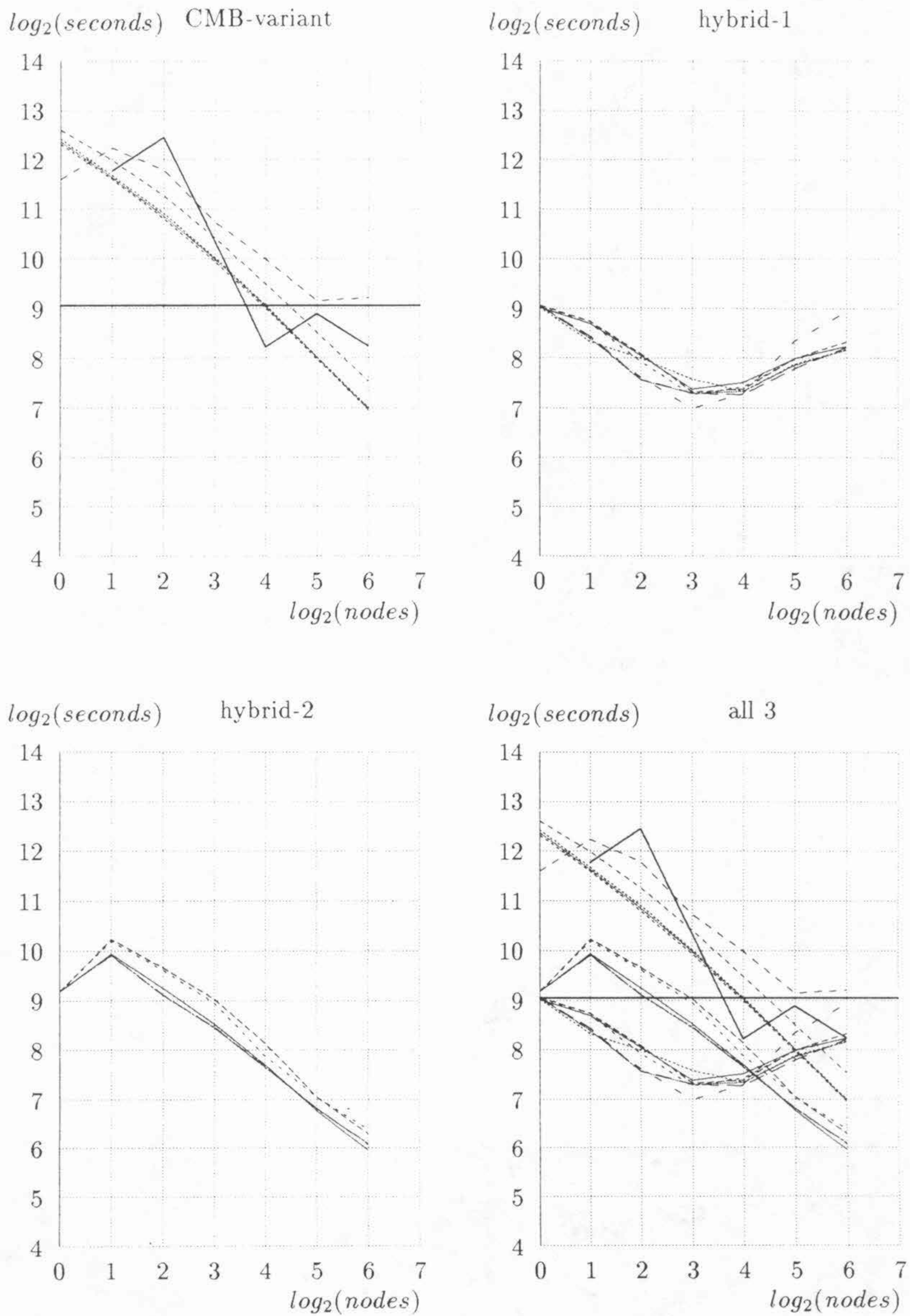

Figure 8.6 An 1841-gate clock network for $50\mu s$ on a Symult 2010.

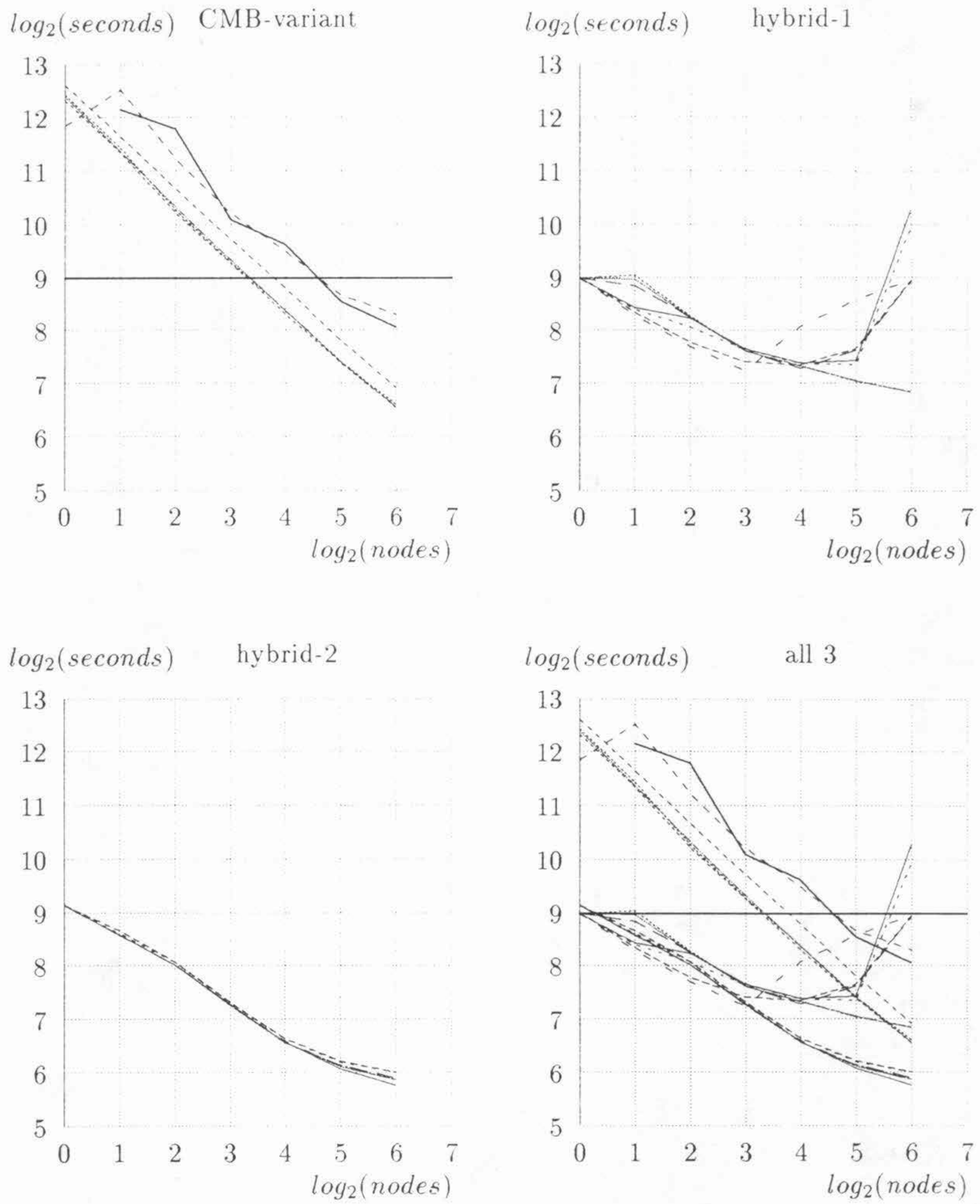

Figure 8.7 A 473-gate clock network for $200\mu s$ on a Symult 2010.

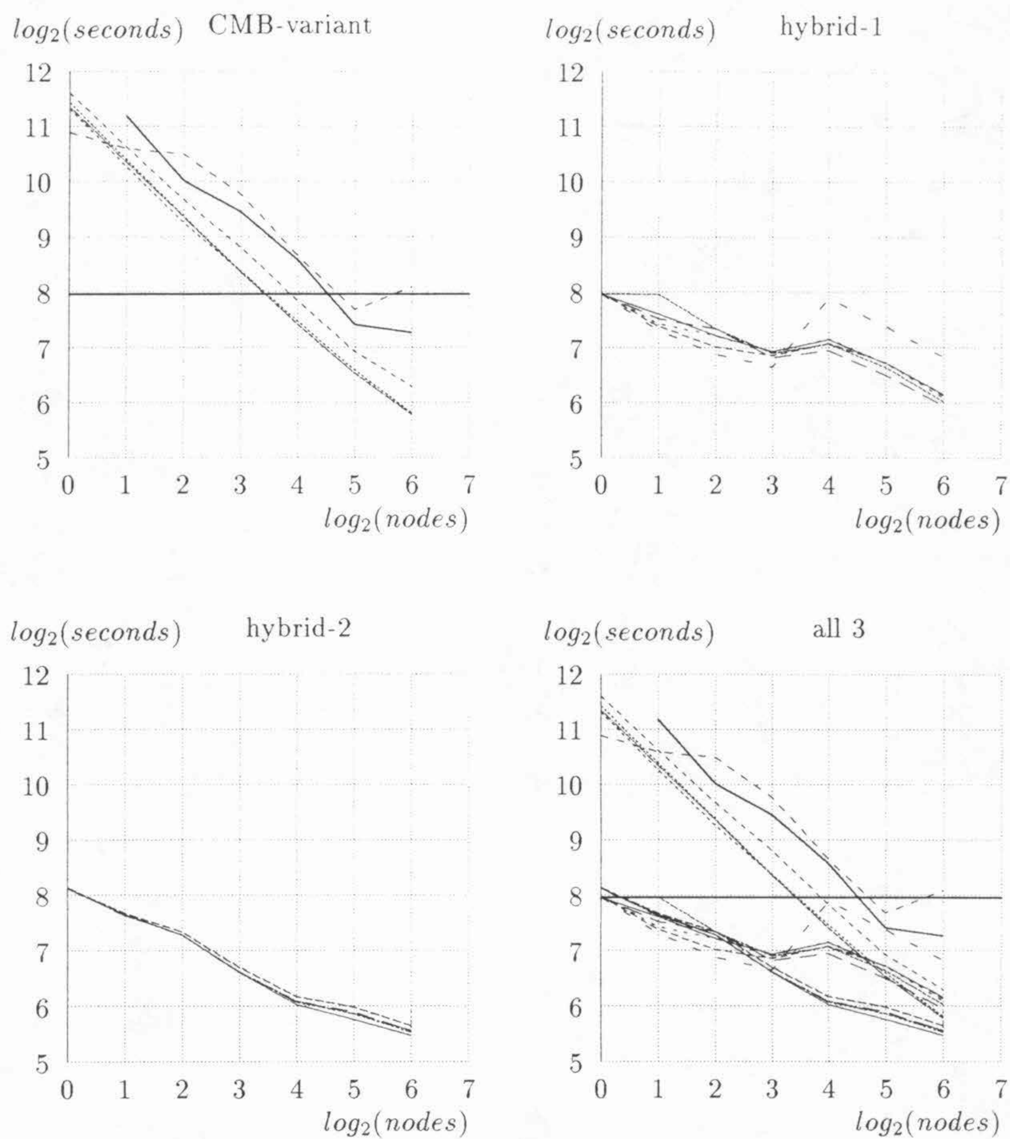

Figure 8.8 A 241-gate clock network for $200\mu s$ on a Symult 2010.

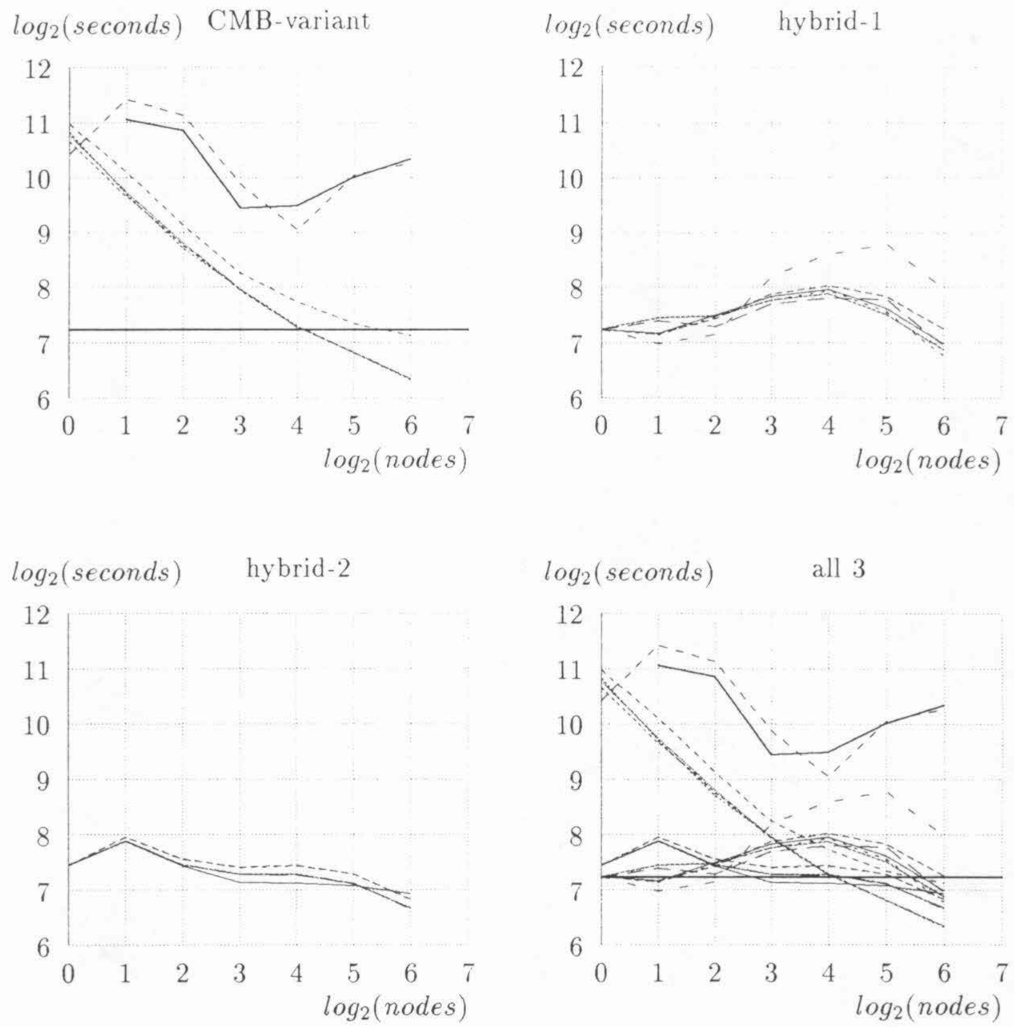

Figure 8.9 A 65-gate clock network for $500\mu s$ on a Symult 2010.

Section 8.2 Tree-Ring Example

8.2.1 Description

Unlike the multiplier and the clock network, the tree-ring circuit has no identifiable functions; it is one of the circuits we invented to test the simulator. It is made of a cycle of 1-to-8 pulse distributors whose outputs are then summed together by a ring of 3-input OR-gates. Each 1-to-8 pulse distributor is composed of seven 1-to-2 distributors connected in a tree structure. A test circuit with 12 distributors appears in Figure 8.10:

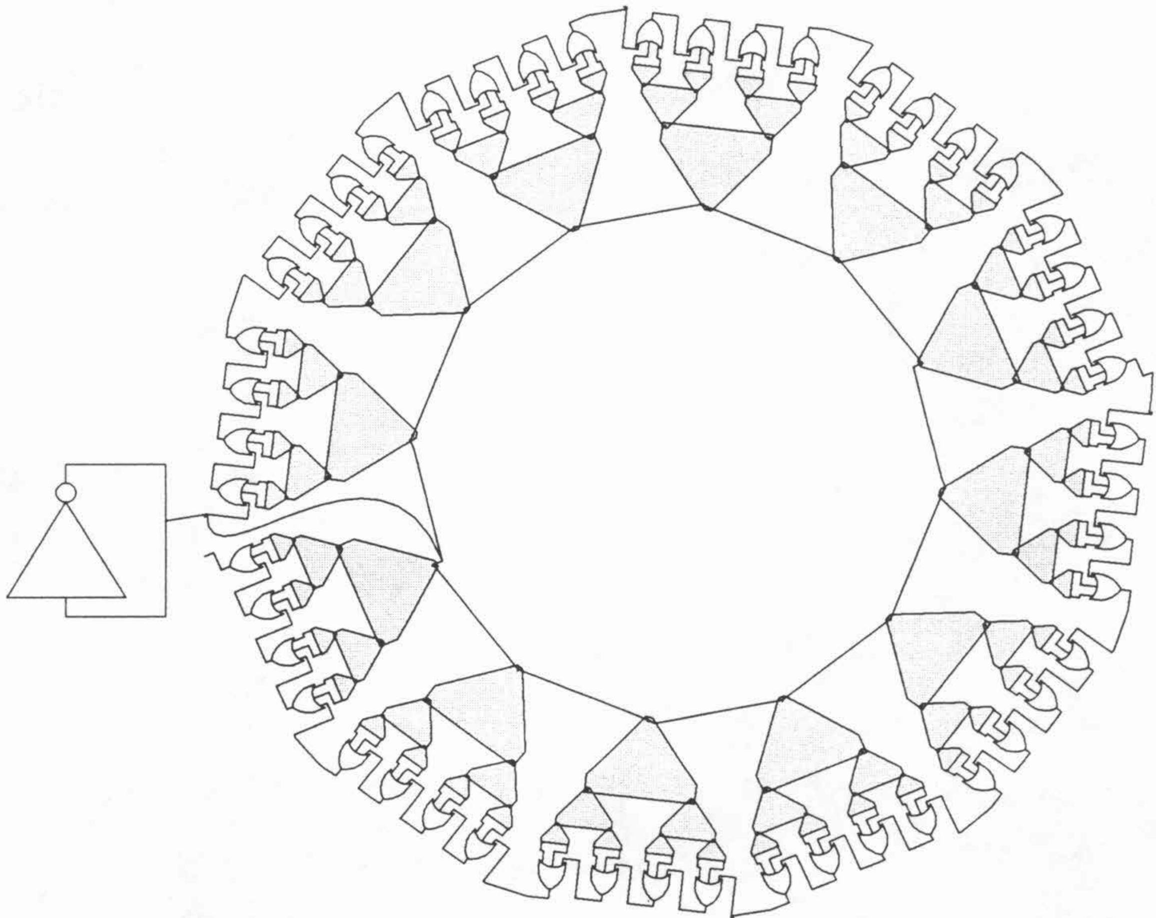

Figure 8.10 A 12-unit tree ring.

Each 1-to-2 pulse distributor has one input and two outputs. Pulses appearing at the

distributor's input are alternatively passed to one of its outputs. Thus, a 1-to-8 distributor spreads the pulses among its eight outputs. A 1-to-2 pulse distributor consists of a toggle flip flop, made of 9 logic gates, and a 1-to-2 demultiplexor, made of 4 logic gates:

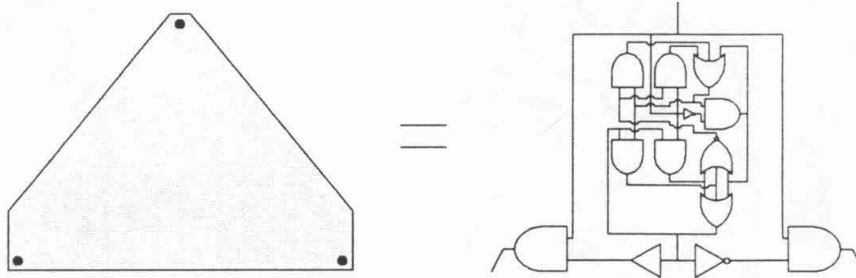

Figure 8.11 A 1-to-2 pulse-distributor circuit.

8.2.2 Simulation results

Sweep-mode simulation has not been done for this circuit. The graphs on the following pages are for the simulation of a 12-unit circuit, using both systematic and random element distribution; a 9-unit circuit; a 6-unit circuit; and, finally, a 3-unit circuit. Tree-ring circuits have a lower activity level than the others examined here because only one of the eight leaves in each unit can be active at any time. Accordingly, the CMB-variant curves show an overhead of four to five octaves relative to the sequential simulation results. The CMB-variant speedup is, otherwise, linear with respect to itself.

The hybrid-2 curves are not as smooth as those of the other circuits because each tree-ring circuit contains two sets of sub-circuits with very different properties (the pulse distributor and the ring of `OR`-gates). Partitioning of the circuit over different-sized multicomputers produces very different locality relations, which strongly affect the performance of the hybrid simulators. The effect of locality can also be seen in the simulation with random element distribution. While the hybrid-2 curves for the clock network merely worsen, those for this circuit converge immediately to the CMB-variant curves at $N = 2$. The CMB-variant simulator, however, is not strongly influenced by locality.

The CMB-variant curves, which are pegged to the ratio of null messages verses event-containing messages, show very little change as the size of the circuit is decreased. The

hybrid simulator curves show a steady flattening in slope, and hybrid-1 curves eventually lose all speedup when there are only 287 gates left in the circuit.

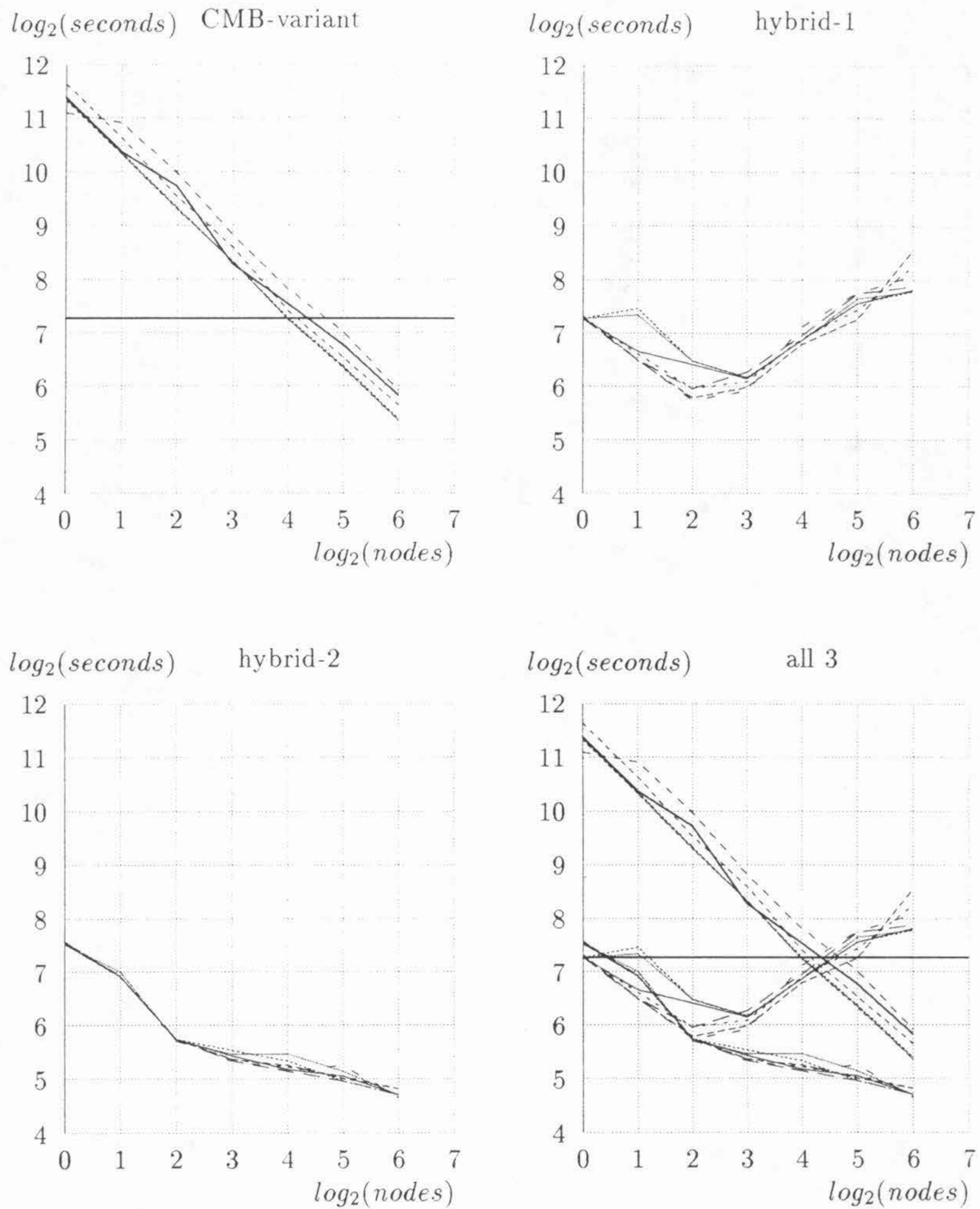

Figure 8.12 A 1142-gate tree network for $50\mu s$ on a Symult 2010.

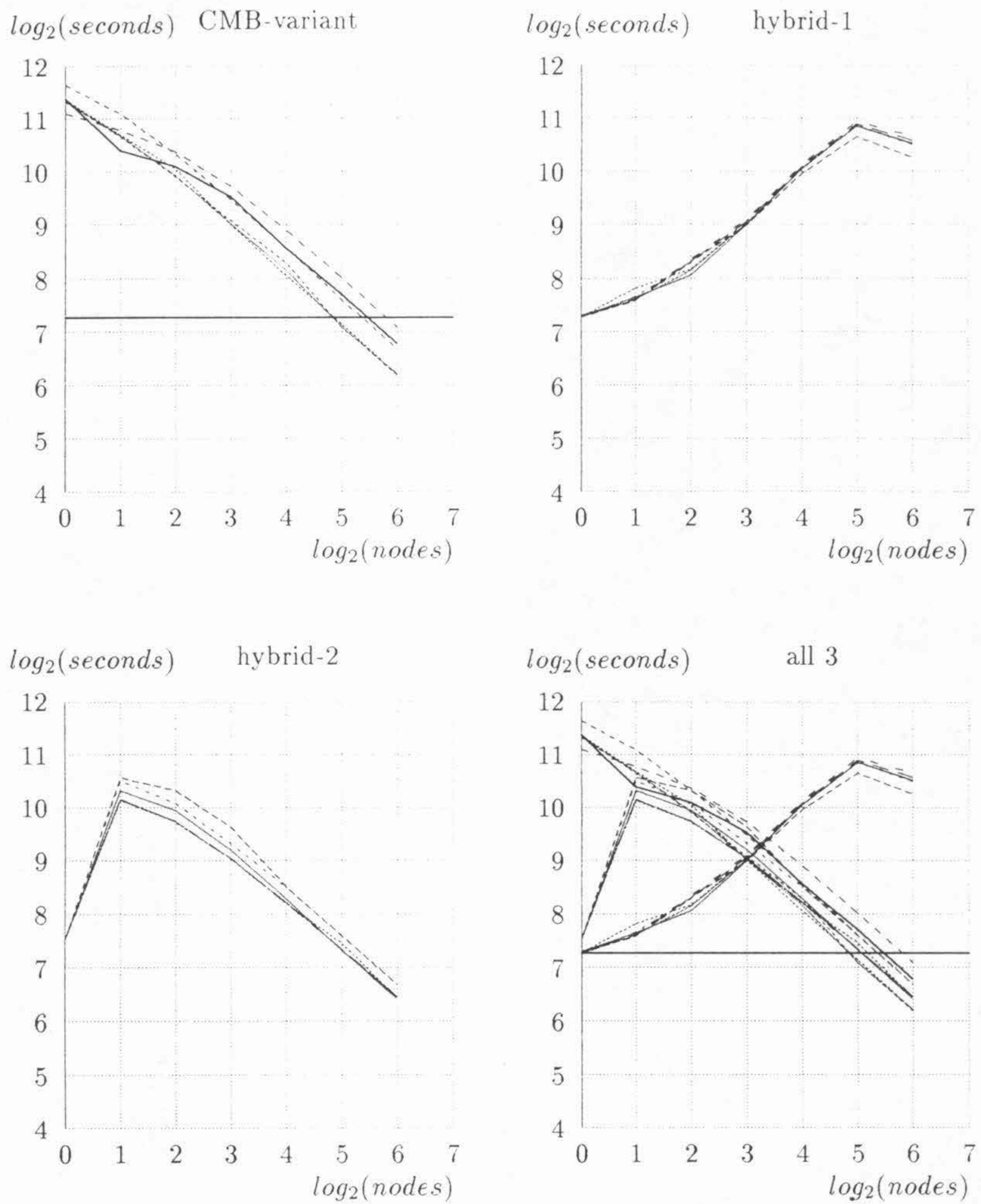

Figure 8.13 A 1142-gate tree network for $50\mu s$ on a Symult 2010.

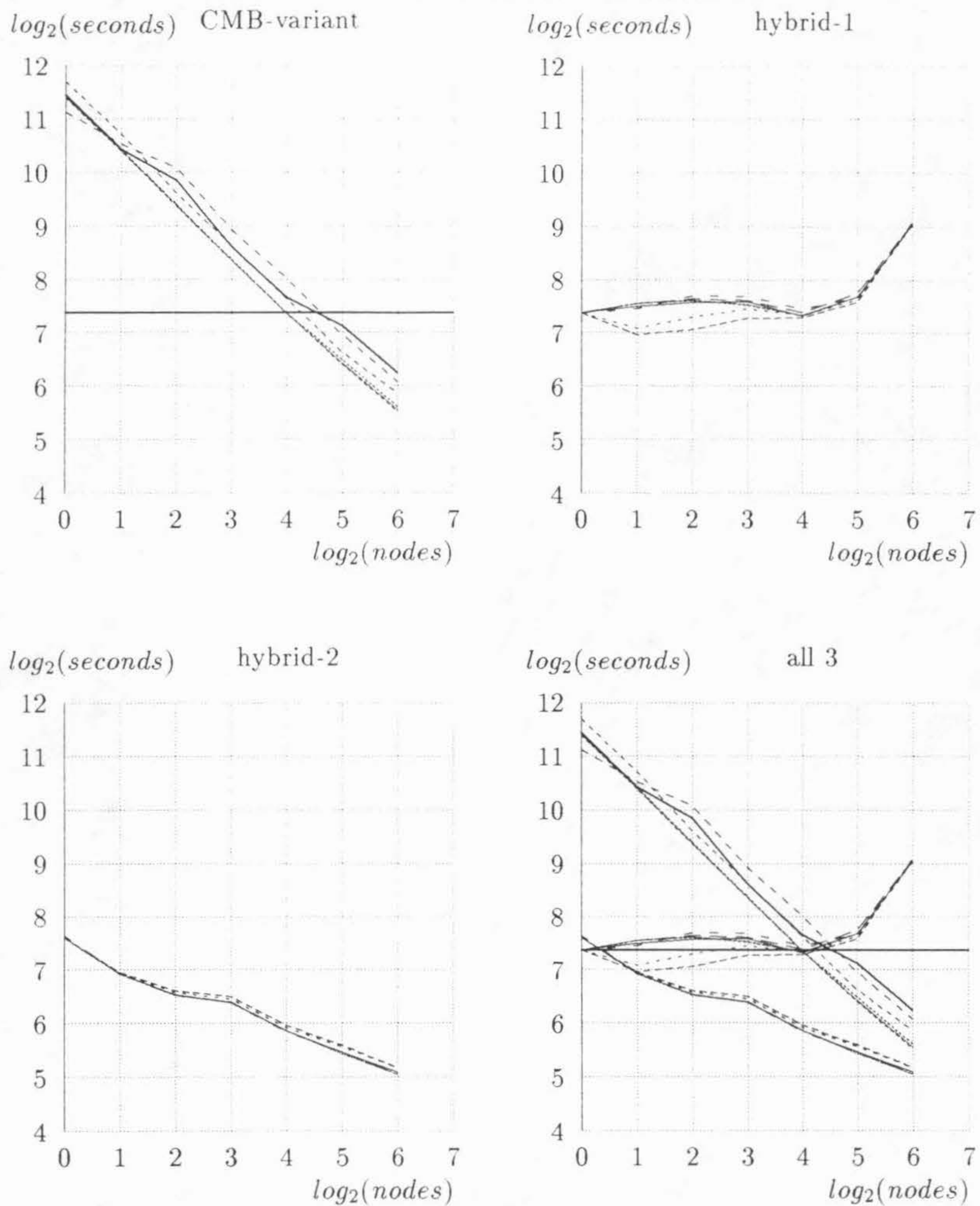

Figure 8.14 An 857-gate tree network for $70\mu s$ on a Symult 2010.

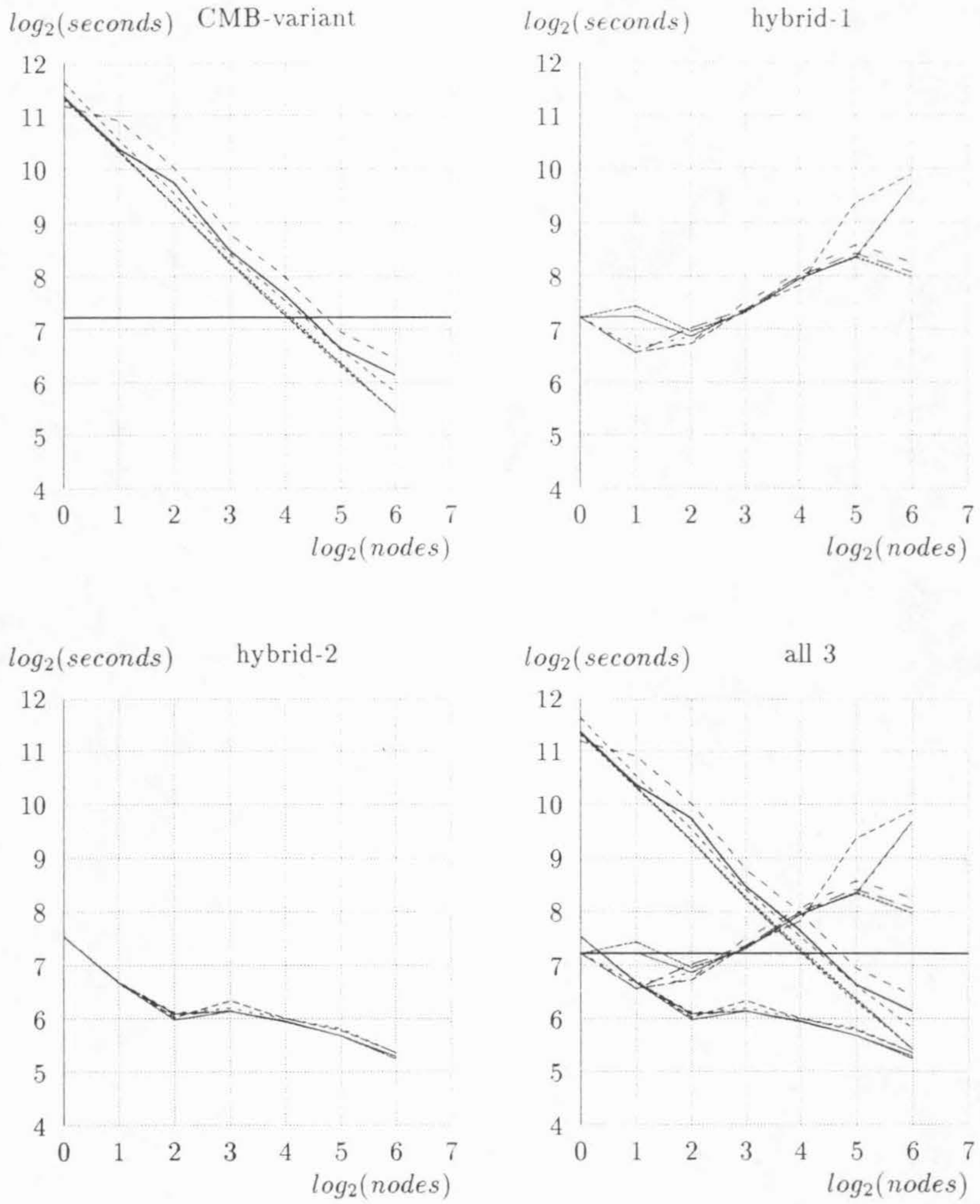

Figure 8.15 An 572-gate tree network for $100\mu s$ on a Symult 2010.

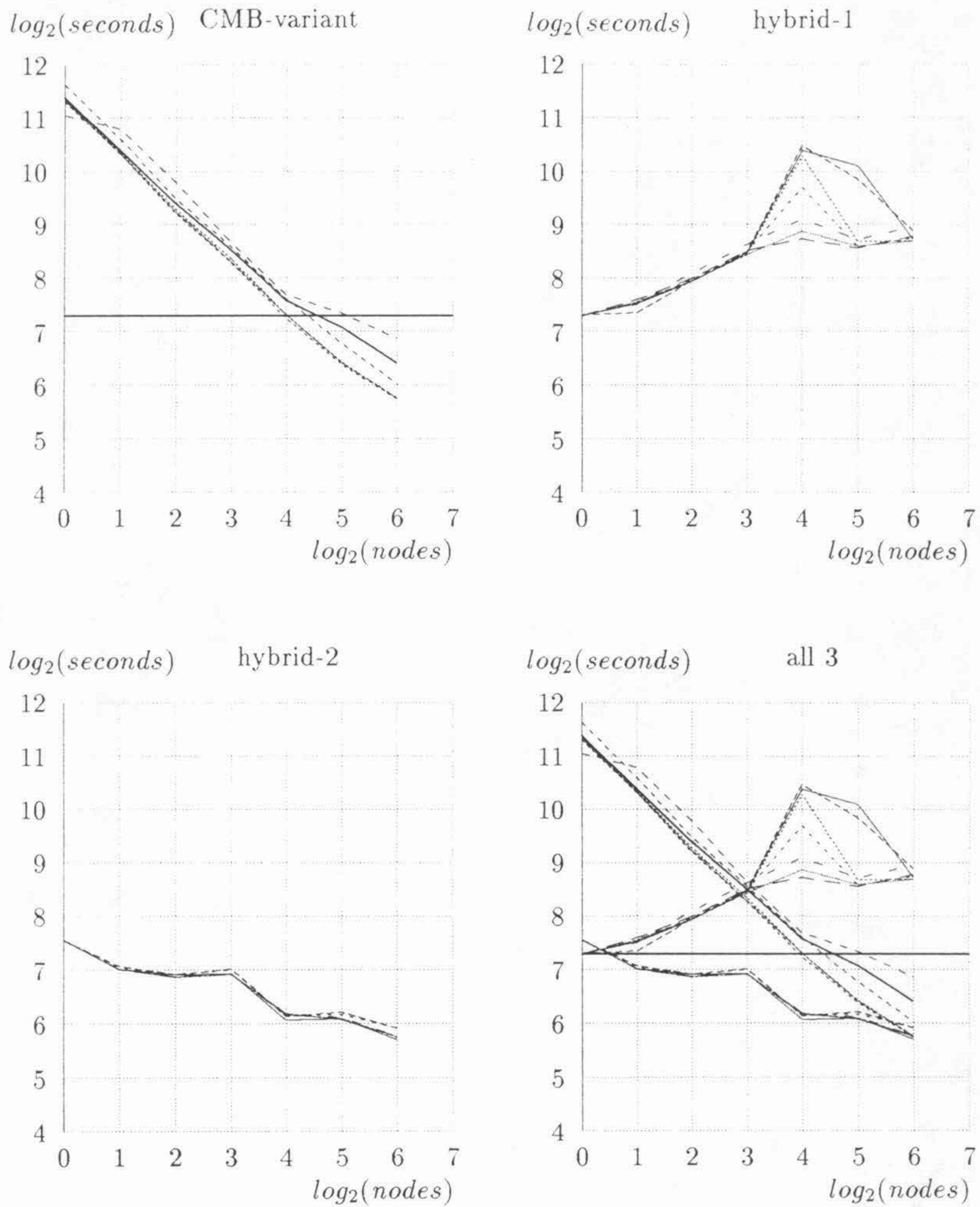

Figure 8.16 An 287-gate tree network for $200\mu s$ on a Symult 2010.

Section 8.3 FIFO Loop

8.3.1 Description

While the clock network example uses a 2-D array of cross-connected FIFO controllers, the FIFO loop example uses a circularly connected linear array of FIFO controllers and FIFO registers. (Refer to the figure in the clock network section.) The registers are made of a bank of 8 cross-coupled latches with clocked inputs. Each latch is made of 4 logic gates, as shown in Figure 8.17.

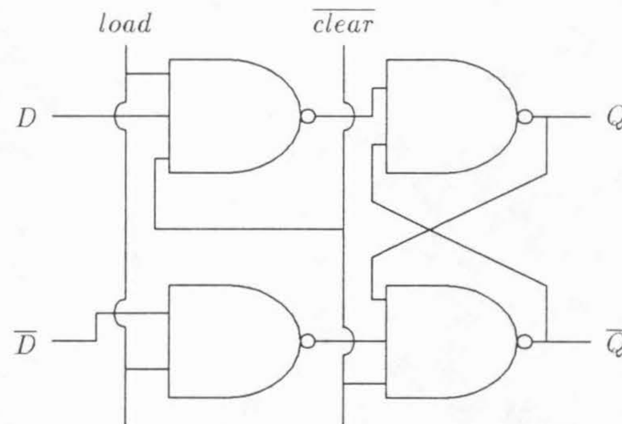

Figure 8.17 Circuit for one latch.

Since the design of the controller constrains the FIFO to contain no more than 1 unit of data for every pair of FIFO units, and since we chose to initialize the FIFO loop with alternating data units of all ones and all zeros, the number of FIFO units must be a multiple of four.

8.3.2 Simulation results

Figure 8.18 contains the CMB-variant sweep-mode simulation result using a loop of 28 FIFO units. The FIFO loop is an example with a lot of usable concurrency. However, unlike the clock network, the lazier simulation variants are not any better than the most eager simulation variant, evidently due to the majority of the circuit loops being found in the cross-coupled latches. Non-essential null messages do not remain long in the cross-coupled latch because the load signal and the reset signal must be long enough for the cross-coupled latch to settle down to a final value. In doing so, the input to one of the cross-coupled

latches is held low for a sufficiently long time that all free-running null messages in the cross-coupled latch are eliminated due to the non-strict input condition of the NAND-gates.

Yet, there are still essential null messages in the simulation, and the overhead estimate of the sweep-mode simulation is between 2 and 3 octaves. The curves should show a linear speedup up to $N = 256$ before they start to level off.

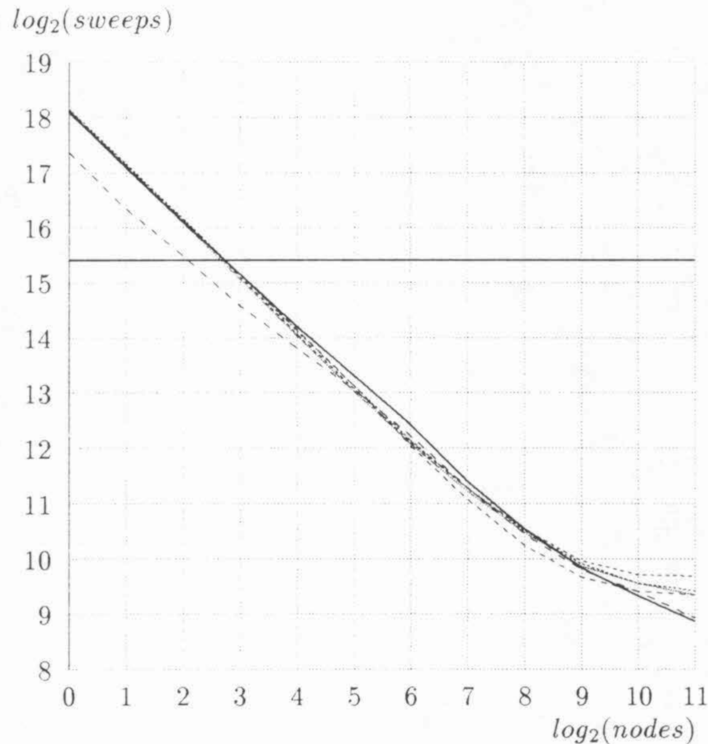

Figure 8.18 Sweep-mode CMB-variant simulation of an 1067-gate FIFO loop.

The real-mode CMB-variant curves for the FIFO loop circuit matches the sweep-mode predictions well. The curves for the hybrid simulators are also as expected. The hybrid-1 curves flatten out and cross over the CMB-variant curves earlier than they do in the previous examples because the gates in this circuit are under non-strict input conditions most of the time, and because hybrid-1 simulators are unable to make use of such conditions.

One unique characteristic of this circuit is that when the circuit size is reduced to 4 FIFO units, all three sets of results show evidence that the curves are bending upward at $N = 32$. This characteristic is not observed in the sweep-mode result, and is an indication that some tight loops are broken up and distributed across node boundaries. At $N = 64$,

there are 2 or 3 elements per node. With granularity approaching the number of gates in a cross-coupled latch, a misalignment in a systematic distribution will cause the majority of the cross-coupled latches to be split across node boundaries.

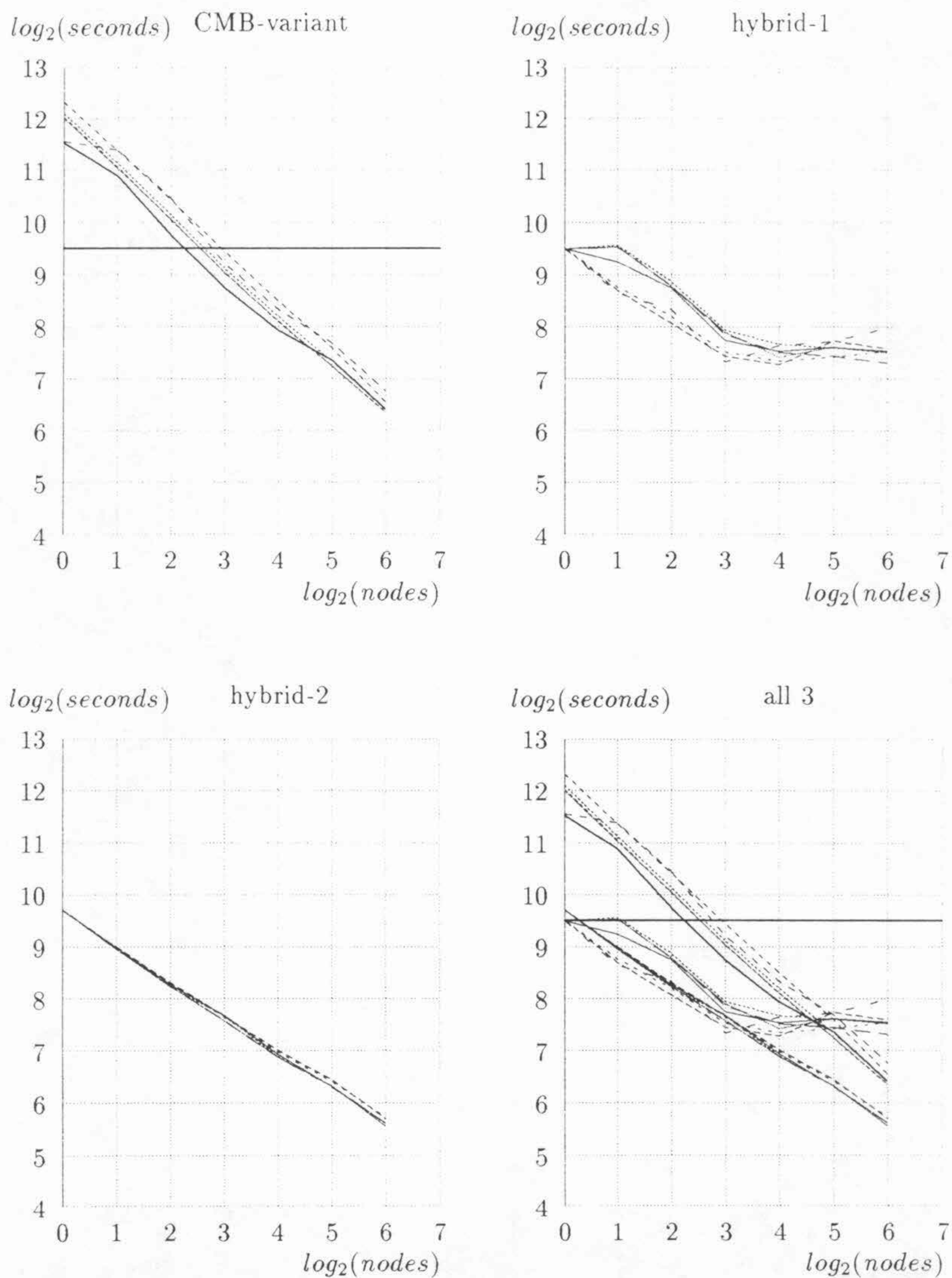

Figure 8.19 An 1067-gate FIFO loop for $100\mu s$ on a Symult 2010.

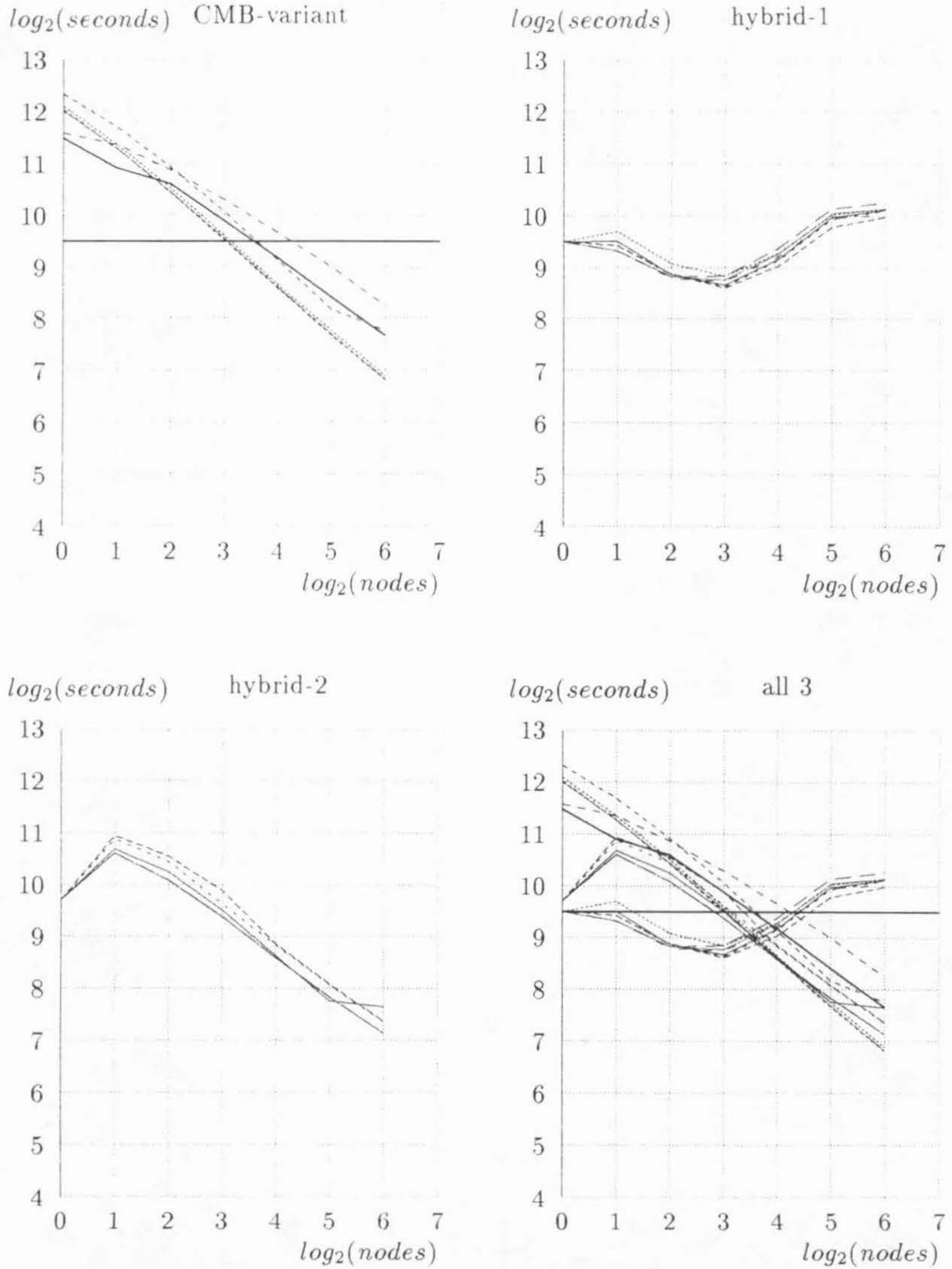

Figure 8.20 An 1067-gate FIFO loop for $100\mu s$ on a Symult 2010.

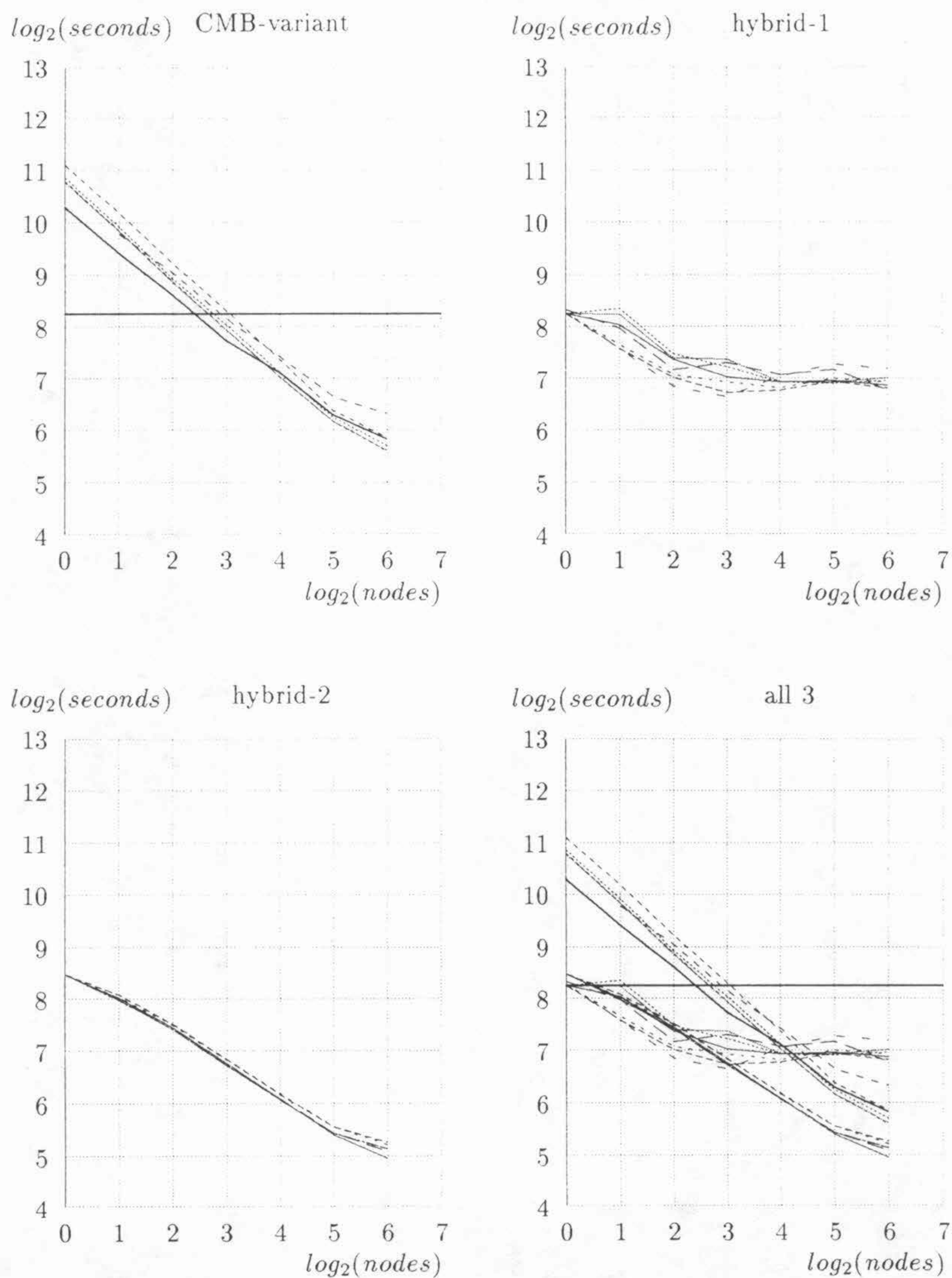

Figure 8.21 A 459-gate FIFO loop for $100\mu s$ on a Symult 2010.

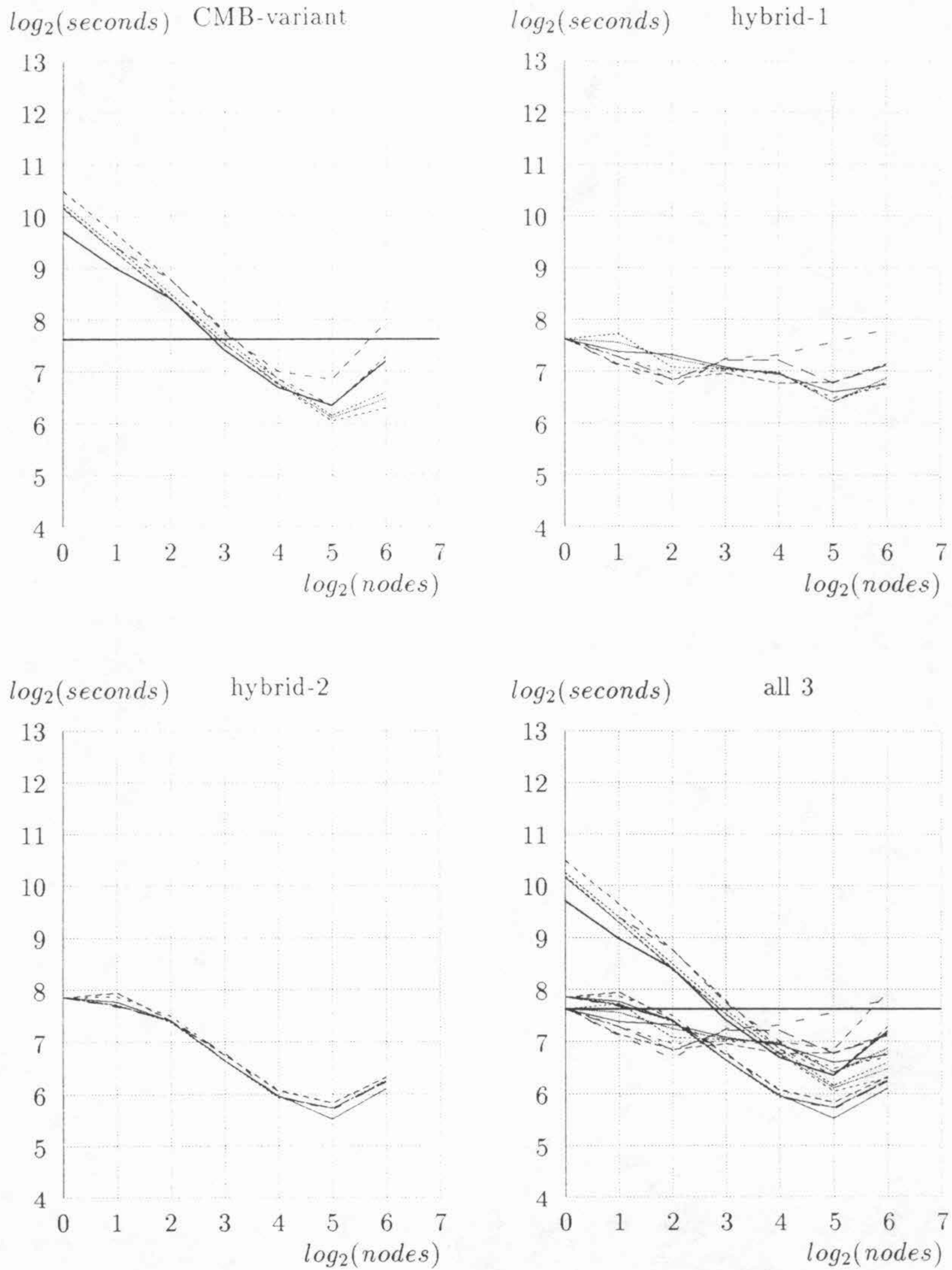

Figure 8.22 A 155-gate FIFO loop for $200\mu s$ on a Symult 2010.

Chapter 9 Summary

Section 9.1 Economy and Performance of a Multicomputer

Multicomputers are appealing because they improve (and, with advances in VLSI technology, promise to continue to improve) the two most prominent figures of merit of computing systems: *performance* and *economy*. Performance is proportional to the processing speed of a machine:

$$\text{Performance} \propto \text{processing speed}$$

Economy is inversely proportional to the cost of running a program; it is, therefore, both proportional to the processing speed and inversely proportional to the cost of the machine:

$$\text{Economy} \propto \frac{\text{processing speed}}{\text{machine cost}}$$

In most cases, performance and economy are at odds with each other because higher speed is achieved by using faster circuits; however, the increase in the machine cost is greater than the increase in the processing speed. In a multicomputer, speed is increased not by having faster circuits, but by having many cooperating computers. Hence, it is possible to improve economy by increasing performance without causing a proportionally larger increase in the machine cost.

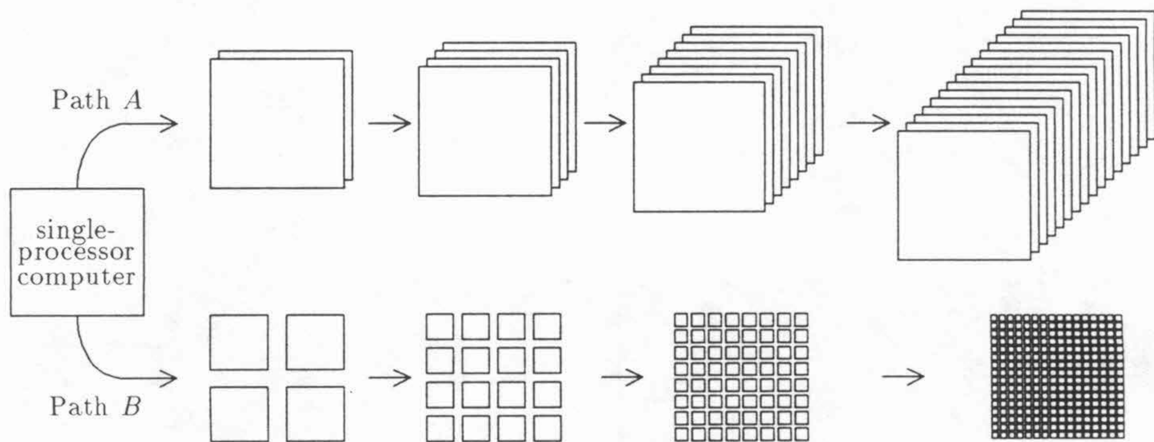

Figure 9.1 Two idealized multicomputer evolution paths.

Whether one agrees that economy can be improved, however, depends on how one sees the basic premise of multicomputing. Shown in Figure 9.1 are two idealized evolutionary

paths leading from the same single-node computer. We will, in our idealized model, consider computers to be made entirely of memory, because a fairly fast processor can be built in the area required for a few thousand bytes of fast memory. When we compare two single-processor computers, we compare two collections of memory attached to two identical, zero-sized processors. Thus, any two single-processor computers in our comparison have the same speed regardless of their size differences. We will also assume that programs do not take up more memory as they become more distributed.

Along path A, we build an N-node multicomputer by putting together N copies of the single-node computer. Performance has improved by a factor of N because there are now N single-node computers, and each is as fast as the original; economy has not changed because the total machine cost has increased by the same factor.

Along path B, the circuitry of a single-node computer is regrouped into N smaller nodes. Performance has improved by a factor of N because each of the N smaller nodes is as fast as the original; economy has also improved by a factor of N because performance has improved while the cost of the machine has remained constant.

These paths A and B also have a strong influence on multicomputer programming. The cost C of running a program, in this idealized model, is:

$$C = SNT$$

$S =$ Price per node per unit time ($\propto$ *size* of the node).

$N =$ Number of nodes in the machine.

$T =$ Time it takes for the program to complete.

When drawn as a 3-D *log-log-log* plot, which we call the *cost space*, the surfaces of constant cost are given by:

$$log(S) + log(N) + log(T) = log(C)$$

Constant-cost surfaces, called the C planes, appear as planes perpendicular to the (1,1,1) direction vector. Suppose we have an application whose single-node cost is marked by point P in Figure 9.2. If we can find a point that is lower than P for the same application,

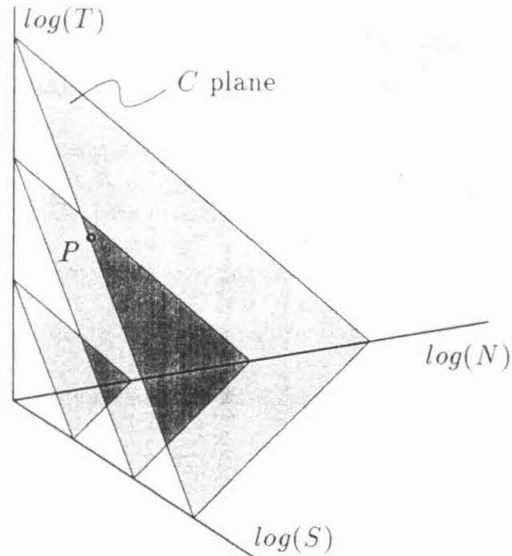

Figure 9.2 Multicomputer cost space.

we have found a point with higher performance; if we can find a point that is on a plane closer to the origin, we have found a point with lower cost.

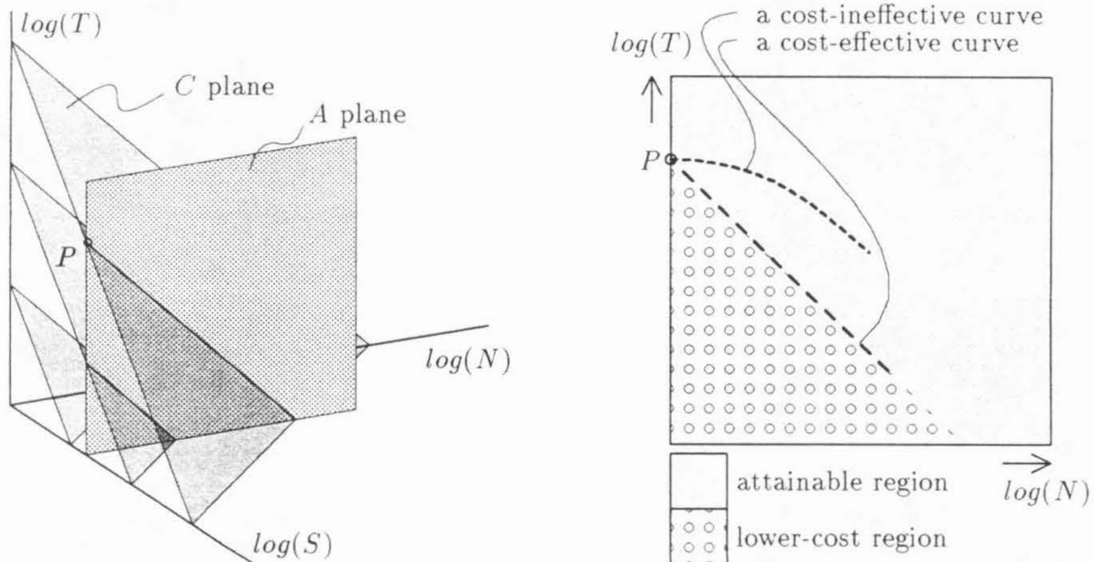

Figure 9.3 Intersection with A plane.

Surfaces corresponding to path A correspond to constant node cost; thus they appear as planes perpendicular to the S-axis. We call such a plane an A plane. Figure 9.3 shows the A plane containing P. The intersections of an A plane with C planes form lines of slope -1 on the A plane. Since super-linear speedup is impossible by our definition, the grey area shown in Figure 9.3 (right) is the possible range of N and T. The cheese area

is the range intersected by those C planes that are closer to the origin than the C plane containing P. The non-cheese area (which is the same as the grey area in this case) is the range intersected by those C planes that are further away from the origin. The only way to have the application be cost-effective is for it to exhibit a linear speedup starting at $N = 1$.

Any deviation from linear speedup means that the performance curve of the application has crossed into a C plane that is further away from the origin, and that the program will be more costly to run. In practice, there are many contributing factors to the actual cost of running a program that may more than make up for the inefficiency, but, in the long run, what we can afford to buy and what we are able to build will ultimately determine the performance improvement we can get by adding nodes.

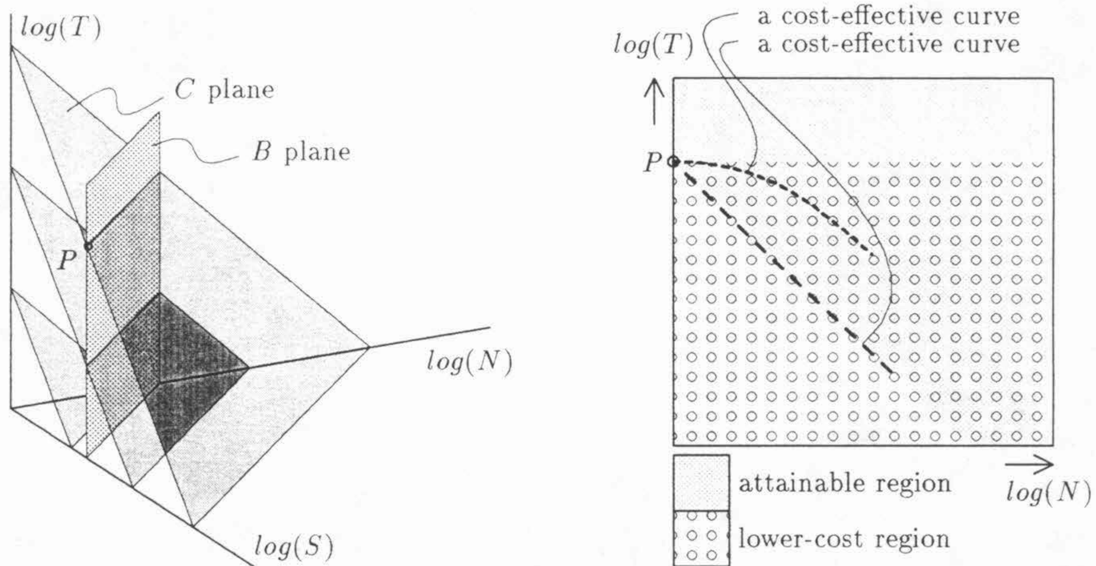

Figure 9.4 Intersection with B-plane.

Surfaces corresponding to path B appear as planes perpendicular to the (1,1,0) direction vector. We call such a plane a B plane. All points on a B plane have the same SN product, and correspond to multicomputers with the same total cost. The plane that contains P is shown in Figure 9.4. The intersections of a B plane with C planes form horizontal lines on the B plane. An application becomes cheaper to run if it shows *any* speedup relative to the 1-node case. Performance is improved because the time required to perform the computation is reduced. Cost is reduced because the computation is now on a C plane that

is closer to the origin. The area that is both grey and cheese is that range that is attainable by the application, and where both performance and economy are improved.

In practice, neither of the two paths can continue indefinitely. In path A, we are limited by the maximum physical size of a machine we are able to build, and by the amount of concurrency we can find in computations. In path B, we are limited by the minimum amount of hardware required to construct a node — computers are not made entirely of memory and most programs do take up more memory as they become more distributed.

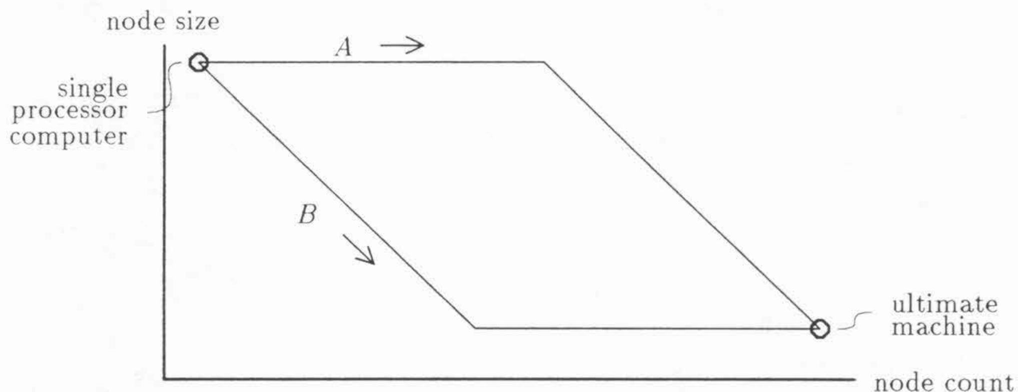

Figure 9.5 Two idealized multicomputer evolution paths in the path space.

To continue, path A must use smaller and smaller nodes and path B must use more and more hardware. The two paths (Figure 9.5) will eventually meet at the *ultimate machine* where all nodes are of a sensibly minimal size and the machine contains as many nodes as we can assemble in one machine.

Section 9.2 Overhead and Latency

Along path B, we encounter a series of multicomputers with progressively smaller nodes. Those with single-board nodes are called the *medium-grain multicomputers*; examples of medium-grain multicomputers are the Cosmic Cube, the iPSC/1, the iPSC/2, and the Symult 2010. Those with single-chip nodes are called the *fine-grain multicomputers*; an example of a fine-grain multicomputer is the Mosaic. Due to the reduced node cost when nodes become smaller and more abundant, the programming emphasis for a multicomputer shifts from one of achieving a linear speedup to one of exploiting the maximum concurrency.

Since medium-grain nodes are few and expensive, the primary goal of programming such multicomputers is to profitably utilize all available CPU cycles. Cycles can be lost to sources in the application itself: load-imbalance, extra synchronization, and insufficient concurrency; these internal delays are called *overheads*. Cycles can also be lost to sources in the system: message handling, kernel operation, and network congestion; these external delays are called *latencies*. In a medium-grain multicomputer, overheads and latencies are countered by employing at least several times more concurrency in the program than there are nodes in the multicomputer. The weak law of large numbers, together with the clustering of related elements, covers most of the problems. Nodes are seldom idle because the chance that all of their elements are blocked is low. The cost of message transactions is low because clustering causes most of the interactions to take place between elements of the same node.

To exploit more concurrency, we must use more nodes in the multicomputer and fewer program elements in each node. Although we can no longer overwhelm overheads and latencies by an abundance of concurrency, we no longer have to be obsessed with linear speedup, because nodes become cheaper as they decrease in size. Instead, programming for fine-grain multicomputers emphasizes the exploitation of all available concurrency in the program. Factors that prevent the exploitation of available concurrency are distinguished from factors that merely require the use of more nodes.

Latencies are factors that can prevent the full exploitation of concurrency. For example, when a message is delayed enroute to a waiting element, the element is blocked and the program may not progress as fast as it could. Overheads, on the other hand, do not prevent the full exploitation of concurrency. When an element is blocked waiting for a message that has not been produced, it is blocked only because the program has less concurrency than there are nodes. Synchronization operations, such as the use of null events in the conservative discrete-event simulators, are also overheads: They keep more of the nodes busy without interfering with the exploitation of concurrency in the system being simulated.

An element with unconsumed normal events may still be blocked awaiting a null event. If the required null event has been produced and sent, we would attribute the blockage to message latency; if the null event has not been produced, then we would attribute the blockage to lack of concurrency.

Section 9.3 Fine-Grain Multicomputer Programming

To fully exploit the concurrency of a program, we must remove all latencies and overheads. Overheads can be mitigated by putting one program element in each node, but latencies can only be reduced by careful hardware and software design.

On the hardware side, message latency can be reduced with high-speed routers. These routers move messages in the network via a modified form of circuit switching called *wormhole* or *cut-through* routing, which moves a message one step through the network in a time comparable to one memory cycle. Since a router is able to store and fetch messages at a rate close to the bandwidth of the memory, sending a message from one node to any other node is comparable to copying the same message from one buffer to another buffer.

On the software side, we must, without giving up generality, provide the thinnest cushion possible between the processes and the hardware. The Reactive Kernel and a fine-grain, light-weight programming environment, such as Reactive-`C` or `Cantor`, make an ideal combination because the program is never further than one function call away from the system. The execution units for these programming environments, especially the more restricted ones like `Cantor`, are small enough that nearly all of the concurrency in the program can be exploited.

We have aimed in the direction of fine-grain multicomputers in all of our research, and our work on the discrete-event simulation is no exception. The CMB-variant simulator is ideally suited for fine-grain machines because it is written in a fine-grain notation, and is able to fully exploit the concurrency of the system it simulates. The simulator takes on a large overhead at $N = 1$, but this overhead does not prevent the simulation from attaining

a large speedup at a large N. In many of the logic circuits we tested, near-linear speedup continues until there are only two or three elements in each node.

Since the CMB-variant simulator does not use any special techniques to reduce the overhead on a medium-grain multicomputer, the qualities that contribute to the performance characteristics of the simulator persist as the simulation becomes more distributed. The hybrid simulators were created to demonstrate the effect of those techniques. The overhead is reduced when N is small, but the effect of these techniques vanishes and the performance converges to that of the CMB-variant simulator when N is large.

Section 9.4 The Next Frontier

We have fully dispersed all available concurrency in a discrete-event simulation program when we put one element on each node. If there were more nodes in a multicomputer than elements in the simulation, we would not be able to utilize those leftover nodes. However, we can still change the program to one that contains more concurrency. In a medium-grain multicomputer, where it is necessary to use concurrency to overwhelm latencies and overheads, rollback simulators such as Time Warp seek to produce additional concurrency by computing on speculation.

The memory in each node of a fine-grain multicomputer is insufficient for storing the previous states of its element in a rollback simulator. However, when there are more nodes than elements, previous states can be stored on unused nodes. When an element has reached a synchronization point, where its future is to be decided by a message that has yet to arrive, the element picks a possible outcome and ships a copy of its old self to an unused node for storage. Alternatively, the element can make a copy of its *new* self, which it spawns and runs on an unused node. But rather than becoming dormant, the old self can continue to run and produce more copies until all possible outcomes have been exhausted. This is the *concurrent branch-and-bound simulator;* it is the next frontier to be explored when a fine-grain multicomputer becomes available.

Chapter 10 Bibliography

[1] G.A. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press, 1986.

[2] W.C. Athas, and C.L. Seitz, *Multicomputers: Message-Passing Concurrent Computers*, IEEE Computer, August 1988.

[3] C.L. Seitz, J. Seizovic, and W-K. Su, *The C Programmer's Abbreviated Guide to Multicomputer Programming,* Caltech-CS-TR:88-1, 1988.

[4] W-K. Su, R. Faucette, and C.L. Seitz, *C Programmer's Guide to the Cosmic Cube*, Caltech CS 5150:DF:84, 1984.

[5] J. Seizovic, *The Reactive Kernel*, Caltech-CS-TR-88-10, 1988.

[6] G.M. Birtwhistle, O-J Dahl, B. Myrhaug, and K. Nygaard, *Simula Begin*, Petrocelli, New York, 1973.

[7] Dan Ingalls, *The Smalltalk 76 Programming System: Design and Implementation*, Proceedings of the Fifth ACM Conference on Principles of Programming Systems, Janurary 1978.

[8] C.A.R. Hoare, *Communicating Sequential Processes,* CACM 21(8):666–677, August 1978.

[9] C.R. Lang, *The Extension of Object-Oriented Language to a Homogeneous, Concurrent Architecture,* Caltech-CS-TR-5014, May 1982.

[10] InMos, Ltd., *The Occam Programming Manual,* Prentice-Hall, 1985.

[11] William J. Dally, *VLSI Architecture for Concurrent Data Structure*, Caltech CS 5209:TR:86, 1986.

[12] R.E. Bryant, *Simulation of Packet Communication Architecture Computer Systems.* MIT/LCS/TR-188, November 1977.

[13] K.M. Chandy, and J. Misra, *Distributed Simulation: A Case Study in Design and Verification of Distributed Programs,* IEEE Software Engineering, September 1979.

[14] D.R. Jefferson, *Virtual Time,* ACM Transactions on Programming Languages and Systems, 7(3):404–425, July 1985.

[15] W.C. Athas, *Fine-Grain Concurrent Computations,* Caltech CS 5242:TR:87, 1987.

[16] Donald E. Knuth, *The Art of Computer Programming, V3, Sorting and Searching,* Addison-Wesley, 1973.

[17] M.R. Garey, and D.S. Johnson, *Computers and Intractability, A Guide to the Theory of NP-Completeness,* W.H. Freeman and Company, 1979.

[18] A.J. Martin, *A Message-Passing Model for Highly Concurrent Computation,* Caltech CS-TR-88-13, 1988.

[19] M. Schuster, R.E. Bryant, and D. Whiting, *MOSSIM II: A Switch-Level Simulator for MOS VLSI, User's Manual,* Caltech CS 5033:TR:82, 1982.