MIT/LCS/TR-380

Logic Simulation on a Multiprocessor

Elizabeth Bradley

November 1986

This blank page was inserted to preserve pagination.

Logic Simulation on a Multiprocessor

by

Elizabeth Bradley

© Massachusetts Institute of Technology 1986

This research was supported by the Defense Advanced Research Projects Agency and was monitored by the Office of Naval Research under contract number N00014-83-K-0125.

Logic Simulation on a Multiprocessor

by

Elizabeth Bradley

Submitted to the department of Electrical Engineering and Computer Science
on May 9, 1986 in partial fulfillment of the requirements for the degree of
Master of Science
in Electrical Engineering and Computer Science

ABSTRACT

The performance of circuit simulators running on SISD computers is fundamentally limited by the Von Neumann bottleneck. Multiprocessors do not share this limitation. The task of solving the equations for the many parallel signal paths found in most circuits lends itself readily to concurrent computation. For both of these reasons, parallel processing is a highly promising approach to circuit simulation. This thesis explores several facets of this problem.

The logic simulator CONSIM was implemented in the parallel language Multilisp, which contains special constructs for dispatching tasks in parallel. A model for the simulator's behavior was developed using a series of experiments. The analysis explains the effects upon CONSIM's performance of several parameters, including: the number of nodes in the multiprocessor, circuit size and topology, and the algorithms for generating the simulation code and for taking advantage of its inherent parallelism. The final generation of these algorithms exposed and exploited significant parallelism, but did not attain linear speedup.

Name and Title of Thesis Supervisor:

Dr. Robert H. Halstead, Jr.
Assistant Professor of Computer Science and Engineering

Key Words and Phrases:

simulation, logic simulation, Lisp, multiprocessing, parallel processing

Acknowledgments

Many people contributed directly and indirectly to this project. I would like to give my special thanks to these particular people and dedicate this thesis to them:

To Bert Halstead, my thesis advisor, for patiently leading me through the Computer Science maze. His sense of direction drew this research out of many blind alleys. His understanding and encouragement have also extended beyond MIT and fostered my rowing career.

To my officemates, Peter Nuth and Dan Nussbaum, who helped keep me sane through research, thesis, and orals.

To Sharon Gray, for sharing her insights into the heuristics and mechanisms of *future* placement.

To John Wyatt, for ongoing professional guidance and a careful proofreading of chapter 2.

To all members, past and present, of the Real Time Systems group, especially its leaders and its "den mother," for making RTS a wonderful place to work.

To my parents, for constant support, love, encouragement, and the occasional necessary reminders that rowing is not the only important thing in the world.

To Jeanne Flanagan, for all the same things (with an alternate opinion on the last issue.)

TABLE OF CONTENTS

TABLE OF FIGURES

TABLE OF TABLES

Chapter 1

Introduction

Using computer simulation, a designer can explore the stresses in a bridge, the aerodynamic drag coefficient of an automobile, or the performance of a circuit without having to first construct a prototype or model. The economic advantages of this are obvious — building and discarding bridges or cars can become quite expensive. Eliminating the time lag caused by prototype construction also allows creativity to flow more freely through the design process. The circuit application, although driven less by *economic* pressure than some other cases, is nonetheless complex enough to require powerful simulation tools to keep its architects productive and sane. When circuit size exceeds a few dozen devices, solving the equations by hand becomes tedious and simulation tools are useful. In modern VLSI circuits, where sizes regularly exceed 100,000 transistors, these tools are not merely useful, but downright vital. Besides saving the designer from having to solve thousands of equations in thousands of unknowns, the immediate feedback saves ideas that might otherwise be lost while the IC masks are at the foundry or the PC board is being etched.

The computer power required to simulate a circuit increases, linearly or worse, with circuit size. If simulation technology is to keep pace with circuit size, the simulation time must not increase accordingly. To simulate a larger circuit in the same amount of time, we must either focus more CPU power on the task or somehow selectively lower the complexity of the task itself by finding shortcuts. The ubiquitous cost/performance tradeoff is at the root of this problem. In this particular case, the "cost" is computer time and the "performance" is accuracy and speed.

The arsenal of simulation theory abounds with techniques which attack one end or the other of this tradeoff. Classical theory has largely focused on finding and exploiting shortcuts in the algorithms. For example, certain variables may remain unchanged during some time interval in a simulation run. Accuracy can be locally downgraded in that interval and thus traded for overall speed. Variations on this

type of technique are discussed later in this paper.

Squeezing the other end of the tradeoff is less common and more powerful. Perhaps the most obvious way to apply more computer power to a problem in a given amount of time is to use a faster computer. The speeds of Von Neumann computers are fast approaching their fundamental limits: more and more research and design effort is required to gain an increment of performance improvement. An intriguing and highly attractive alternative is the use of multiprocessors.

There are two angles of thought which lead to this approach. The first leads from system performance considerations. The fundamental "bandwidths" of multiprocessors are much higher than those of Von Neumann machines. A multiprocessor can deliver more cycles per second, which is precisely the desired result. A simple observation about circuit topology also leads naturally to a parallel-processing approach to simulation. Circuits contain many parallel signal paths, while a simulation algorithm running on a SISD computer is purely sequential in nature. The concurrency of the circuit is not duplicated in the simulator, which causes the program to run slower than the hardware by a factor proportional to the amount of parallelism in the circuit. Partitioning the circuit and distributing the task among the nodes of a multiprocessor is an esthetically pleasing and hopefully efficient solution to this problem. Note that the underlying cost/performance tradeoff has been sidestepped but not completely avoided. The *real time* required to run a particular simulation decreases, but the *CPU time* remains constant, since many computers are working in parallel on the problem.[1]

Partitioning a complex circuit, such as a CPU, so as to facilitate faster simulation is not at all trivial. The boundaries of blocks in a partitioning that streamlines the computation may not resemble hardware boundaries at all — for example, a designer may find it easier to follow the operation of her circuit by thinking of a block as an "adder," while an algorithm that treats that block as a network of XOR gates may run much faster due to the behavior of the computer system upon which it executes. The mathematics behind a successful partitioning is a combination of circuit theory, simulation theory, programming technique, and the architecture of the computer involved.

The logic simulator CONSIM, described in this thesis, investigates these issues. The effects of coding algorithms, partitioning schemes, parallelism exploitations,

[1] The actual CPU time may even increase over the sequential case due to the overhead involved in orchestrating the parallel tasks.

and circuit size and topology upon simulator speed were isolated. Using these results, CONSIM was tailored to take maximum possible advantage of the parallelism made available from its target machine — the Concert multiprocessor — by the language Multilisp.

Concert [3] was constructed by the Real Time Systems Group of the Laboratory for Computer Science at MIT. It consists of an array of Motorola 68000 microprocessors and some shared memory. The processors are organized in eight groups or "slices" and connected on a common Multibus. Several slices are linked using an interconnection scheme called the RingBus. As of this writing, 14 processors on 5 slices are functional.

Multilisp, an extension of the Lisp dialect Scheme [1], has been implemented on Concert [14]. Multilisp shares much of the function and semantics of Lisp, but incorporates additional operators and semantics to take advantage of the parallelism of its implementation. Examples of these special constructs are (1) the *pcall* construct, in which the programmer specifies two or more expressions whose evaluations may proceed concurrently, and (2) the *future* construct, which allows a process to execute given a "promise to deliver" its arguments at a later time. The Multilisp implementation, without programmer intervention, takes care of the parceling out of tasks associated with (1) and the "filling in at a later time" associated with (2).

The intent of the Concert project was to provide researchers with a functional testbed for multiprocessor applications. Multilisp brings this functionality to the hands of the programmer. Together, Multilisp and Concert represent an ideal system on which to experiment with the problem at hand.

The main goal of this research was to answer some questions about the issues that affect performance of a simulator running on such a system: Does performance reflect hardware or software issues? How should the code and algorithms be structured to take maximum advantage of the multiprocessor system and its underlying hardware? How should tasks be divided to enhance parallelism? Of course, a welcome side-effect[2] was a functional logic simulator which can verify logic designs.

Several circuits, as small as a dozen gates and as large as several hundred, have been simulated under CONSIM on as many as 14 Concert processors. All show significant parallelism gains over their sequential execution times, but the gains fall short of the $t = \frac{1}{n}$ speedup theoretically available from n processors. For example, a simulation of a 58 gate ALU circuit runs 6.3 times faster on 14 processors as on one.

[2] Lisp occasionally *does* have side-effects.

Execution times fall as processors are added to the system, but this improvement is limited by the size of the circuit and the length of the simulation run, which together determine the number of tasks in the computation. The parallelism exploitation algorithms choose among these available tasks to fill the system's pipelines. Larger numbers of tasks allow the algorithms to keep these pipelines full and the throughput maximal. This has novel implications in the CAD world: a tool whose efficiency does not decrease radically with circuit size is highly uncommon and promising.

CONSIM is a logic simulator which was designed to run efficiently on a particular multiprocessor. It cannot compete outright with state-of-the-art, lightning-fast simulators. Its purpose is not to have the words "fastest, newest, improved product" splashed in bright colors across its package, but rather to explore how the particular flavor of parallelism made available by multiprocessors can be exploited to make a simulation algorithm run faster. Farther along this path lie simulators with truly blinding speed that will leave the current state of the art far behind.

Thesis Outline

By way of introduction, chapter 2 presents a brief review of the literature related to this endeavor. This establishes CONSIM's roots and reveals the novelty of its approach. Chapter 2 then presents an outline of the goals of the CONSIM project. Chapter 3 describes the environment, both hardware and software, in which the simulator operates. The software description presents the details of Multilisp on a level which presupposes some knowledge of Lisp or another programming language. Chapter 4 discusses the implementation of the simulator, beginning with the class of circuits to which it applies. CONSIM's general structure is then outlined, and the hardware description language (HDL) is defined and discussed, along with the algorithms that transform the hardware description into simulation code and other algorithms which exploit the parallelism in this code using Multilisp's special constructs. Chapter 4 concludes with a top-level CONSIM "user's manual," which ties together the aforementioned low-level description. The actual experiments performed are documented in chapter 5. The procedures, example circuits and issues are introduced, and experiments are proposed which investigate each issue. In each case, the results are presented and discussed in light of the underlying issues. Chapter 6 summarizes the conclusions of the research and proposes direction for future work.

Chapter 2

Approaches to Simulation

2.1 Previous Research

In circuit simulation, the network equations derived from device physics are repeatedly solved to determine the values of the variables (voltage, current, etc.) at the nodes and/or branches of a circuit. The values of the variables are computed for one time interval using the input values for that period. The internal and external variables are updated and the simulation continues at the next time point. The network equations may be a mixture of algebraic and differential equations; they will be nonlinear if the circuit contains nonlinear elements. For simulation purposes, the equations comprising a full description of an n node circuit are formulated in various ways, creating matrices whose dimensions are typically $O(n)$. An intermediate linearization step is required to get to these forms if the network equations are differential or nonlinear. The problem solution then reduces to a matrix inversion and multiplication. The computer time required to perform these operations can grow as badly as n^2, where n is the size of the matrix and, indirectly, of the circuit.

If simulators are to keep pace with increasing circuit size, a time penalty which grows as n^2 is unacceptable. For a circuit containing 10,000 devices, one signal input and two clock inputs, a 1,000 time step simulation run on a 370/168 takes approximately 10^5 seconds assuming 1 millisecond per device-iteration and three iterations required to reach convergence [27]. This amounts to about a day of CPU time. To simulate a circuit containing 100,000 devices, an example ten times as large, would take 10^2 times as long: about three *months* of continuous CPU time. In addition, the higher density devices may require more complex models because they are so small. This contributes to computational complexity and thus to device-iteration time. All things considered, the motivation for faster simulators is painfully obvious, and research has proceeded accordingly.

Techniques of improving simulator speed fall in two general categories. Those

in the first category use more efficient and intelligent algorithms, while those in the second attempt to improve the implementation upon which the algorithms run. Most of the research in the literature has focused on the former approach. Only recently, as alternative architectures appeared, have the limits of the implementation been explored.

Algorithm Improvements

Matrix mathematics provides many shortcuts which improve the n^2 performance lag. Gaussian elimination, LU decomposition, pivoting, node and branch tearing and sparse matrix manipulations are examples of this type of improvement. A full description of these methods is beyond the scope of this paper — the reader wishing further elaboration should refer to [13] for the tearing techniques or [32] for the others.

Circuit analysis programs, such as SPICE [21] linearize and solve the circuit's nonlinear differential equations using numerical methods, providing a wealth of information and chewing up large amounts of computer time in the process. SPICE, which grew out of another program called CANCER [22] at UC Berkeley during the late 1960s, contains many of the matrix-mathematical improvements described above. A serious problem with SPICE is the effect of the time step used. A badly-chosen time step can cause the "solution" to contain very real-looking transients that are purely artifacts of the numerical integration. This time step is chosen by compromise. If it is small enough to follow the fastest-changing node, the rest of the circuit will be oversampled and speed will suffer. If it is too large, artifacts and even nonconvergence can result.

IBM's ASTAP [33] relies upon an alternative formulation of the circuit matrix to achieve efficiency. SPICE, assuming that solution time and matrix size are closely linked, attempts to form the densest possible matrices. ASTAP places all possible information in a sparsely-populated "tableau" matrix to which special techniques, which preserve and exploit the sparsity throughout the computation process, can be applied. ASTAP includes another innovation which is based in the theory of numerical integration. The accuracy of the integration method (i.e., the number of terms used in the predictor and corrector) varies according to the needs of the circuit. The time step, as in SPICE, varies as well. The combination of these factors avoids unnecessary accuracy at points where circuit variables are changing slowly or not at all: the "bypass method." This is not pathological case — over 90 percent

of the transistors in a digital VLSI circuit often remain unchanged over a particular time step [26].

Timing simulators exploit this "time sparsity" by using simpler models for circuit elements. These simulators give approximate results using less CPU time, but at risk of neglecting important effects and giving erroneous or nonconvergent results. The first timing simulator, MOTIS [9], was developed at Bell Labs in the early 1970s. In MOTIS, feedback within devices (such as the Miller capacitance) is omitted from the models, which means that only a single iteration is required to reach a solution. The result is accurate if and only if the assumption is valid; if not, a second, more accurate level of simulation at that time step is required. One way to accomplish this is to allow the iteration to proceed beyond one step to absolute convergence if an error is found. The program must thus be able to estimate its own error [34]. Further model simplification adds yet more speed and intensifies accuracy and convergence problems. An example of a simulator which has succeeded with simpler models and found ways around the convergence problem is the MOS LSI simulator RSIM [31]. RSIM models transistors as resistors, whose values depend upon the logic state of the devices. The value of the capacitance in parallel with each of these resistors is extracted from the layout information (gate and interconnect parasitics) and an RC timing analysis is performed. This process chooses models accurate enough to avoid the pitfalls described above, but simple enough to speed computation. CRYSTAL [25] is a simplified timing simulator which identifies critical paths using static RC time constant analysis. The user may then choose to analyze the critical paths in a more accurate fashion.

In functional level or logic simulators, the model is simplified one step further. These programs, as their names suggest, simply verify the *function* of the circuit under test. Variables typically take on logic values: 0, 1, undetermined (X), and high impedance (Z). No timing information is computed. Since so much less information is involved, the program runs even faster than a timing simulator. For example, a timing simulation of a particular circuit took 2 hours of CPU time, while a functional level simulation took 10 minutes. [2]. In ESIM [31], the sister program to RSIM, transistors are abstracted yet one level higher than in RSIM and are treated simply as switches. Other functional level simulators are TEGAS [30], LAMP [8] and, indeed, CONSIM.

The bypass method capitalizes on time sparsity by varying accuracy with *time*; hybrid or mixed-mode simulators capitalize on the different levels of complexity within a circuit by varying the accuracy over *space*: each section of the circuit is

simulated at an appropriate level of detail. The linear region performance of a video amplifier, for example, is of much higher interest than the details of how a TTL output slews through the noise margins of the following input stage. A timing or even functional level simulation would be appropriate for the latter, while a full SPICE or ASTAP run is indicated for the former. One can envision a *really* mixed-mode simulator which simulates at all levels from functional to actual solution of Maxwell's equations at the device physics level! The critical paths in a logic circuit, perhaps identified by a timing simulator like CRYSTAL as described above, are candidates for a more detailed inspection. The interface between sections of the circuit which are to be simulated at different levels can be delicate, both to specify and to compute. The hardware description language used to describe the circuit to the program must incorporate different description levels; the program must perform complex variable coercions at the boundaries, where logic values meet voltages and times. SPLICE, also developed at Berkeley [24], combines logic and timing levels. Other combinations (i.e., timing and SPICE-level) also exist.

The notion of partitioning a circuit into sections and performing different operations as the sections dictate is extremely powerful. In the most general case, the simulator and not the designer should determine the partition. PowerSpice [20] conquers the SPICE problem of artifact due to bad time step choice by partitioning a circuit into various-size sections according to node time constants. During the simulation, the appropriate time step is then used for each individual section. In macromodeling [19], a "black box" is modeled at its terminals. The behavior of the circuitry within the box is characterized at the outset and then only its "boundary conditions" are used in subsequent computations. The partition is chosen so the black boxes enclose chunks which have complex, computationally-expensive internal interactions, which are bypassed by lumping them together, and so the interactions at the boundaries are easy to compute.

Implementation Improvements

Perhaps the most obvious way to improve a program's implementation is to buy a faster computer. However, the cost of a Cray or a Cyber computer reminds us of the ever-present cost/performance tradeoff. Tasks like astronomic simulations of entire galaxies do indeed mandate supercomputers; circuit simulation, since it is much better understood than astrophysics, lends itself readily to task-specific shortcuts in implementation, much as was the case with the algorithm improvements discussed above. This is based on the assumption that, since circuits are designed

by humans and galaxies aren't, humans can devise safer and more efficient shortcuts in the former case. Another reason for the use of supercomputers in astrophysical simulations is that such simulations are typically quite long, while some of the processes at work have short time constants.

Simulation is inherently repetitive. Large digital circuits are repetitive in *space*: they contain many instances of particular devices or subsystems. Simulating a circuit for a number of cycles is also repetitive in *time*: the same equations are resolved for every time step. Implementation modifications which exploit this quirk accelerate the simulation process.

The simulation software can be "tuned" to the underlying computer and to a typical circuit application. The sections of the code which are called most often, as determined by some sort of time histogram, may be written in assembly language or even microcode. The mixed-level Berkeley program, SPUDS [10], with sections in both assembly language and in microcode, shows a factor of 3 improvement over its high-level language twin. The drawback behind this approach is its unportability: it cannot be used on another computer without intricate modifications, and its efficiency may not be as high when it is used on different sorts of circuits.

Critical paths in the simulation can also be accelerated by special purpose computers. A vector computer can process large numbers of identical tasks, like solutions for RC delays, in parallel. For highly regular circuits, this can speed the simulation by a factor of 10 [26]. Manipulating arrays of numbers is the most time-consuming part of any circuit simulation; a special purpose array processor helps reduce this cost. General purpose improvements like floating point packages or accelerators obviously speed simulation runs as well.

Approaches which exploit the concurrency inherent in circuit behavior — signals propagating simultaneously along many parallel paths — have developed in the wake of recent parallel processor architectures. Parallelism can be mapped onto the nodes of a multiprocessor network in two ways: the processing units (PUs) may be specialized to perform a given task, or they may be entirely general.

An appealing, if unrelated, example of a dedicated multiprocessor application is the Digital Orrery [4]. This machine computes the orbits of the planets in a solar system. The computation involved in this "N-body problem" is greatly accelerated by partitioning: one PU is designated to simulate each planet. The specificity — planet radius, distances to other planets, etc. — is programmed into each PU.

A circuit-simulation analog of this is the Yorktown Simulation Engine (YSE)

[28]. Each PU in the YSE is a "Logic Processor" capable of simulating 4,096 gates. The gate functions to be simulated are microprogrammed into these modules. The YSE can simulate 2×10^9 gate operations per second — which translates to an instruction rate *faster than the native rate* [1] for some commercial processors. It cannot handle levels of abstraction, like lumping 2 gates into a flipflop. [2]

An array of unspecialized PUs may not offer the ultimate throughput possible with perfectly synchronized, specialized units, but interunit communication chores are more tractable if all units are identical. Tasks may be farmed out statically among the processors, as in PRSIM [5], which runs on Concert. Transistor circuits are modeled, as in RSIM, as RC networks. The circuits are then partitioned and each chunk is assigned to a particular processor. Tasks may also propagate dynamically, as in PowerSpice. Each subcircuit is handled by a single process which executes on the first available processor. CONSIM follows this approach.

The "Logic Simulation Machine" at Bell Labs [2] combines dedicated and general-purpose PUs and achieves 10^6 gate operations per second — an order of magnitude slower but somewhat more flexible than the YSE. Gate operations, designated "simple", are processed by PUs which are microprogrammed to perform a small number of fixed tasks. More complex, "functional" operations are handled as individual, dynamically assigned processes. For a "representative" mixture of gate and functional operations running on 8 general and 6 specialized PUs, the Bell Labs authors estimate that the simulation would run 30 to 60 times faster on this special setup than it would on a single SISD computer.

Summary

Perhaps the most persuasive argument for attacking simulator technology via implementation improvement is the Von Neumann limit: the fundamental "bandwidth" limit imposed by the processor-memory link through which all communication must pass. Any approach in which a Von Neumann computer, however fast (and expensive), is used will eventually fetch up against this limit. [3] Special purpose

[1] The rate at which instructions are performed by an actual chip operating at its maximum specified clock rate

[2] Another quirk of the YSE surfaces when it must be used to simulate a FET: a FET is modelled as an artificial collection of gates. Most simulators model gates as collections of **FETs**!

[3] In fact, the expense is a result of the gymnastics required to get supercomputers

processing units improve this bandwidth by customizing the operations at both ends of the link and streamlining the information which travels between them. Special hardware serves the same function in a less flexible way, diverting certain parts of the information flow to special alternate "pipes" which are built to handle one kind of data very well. However, the huge systems of the future, containing "myriads" of processors, may not be able to afford the luxury of non-uniformity, either in the form of the data flowing between nodes, or in the nodes themselves. Communication chores may limit performance, and non-uniformity complicates communication.

Many of the techniques in this section apply to *circuits* as opposed to *logic*. The models in the former are much more complex, which tips the computation balance. "Simulating the wiring" is not necessarily a quick operation, since large chunks of information must be passed around between the subroutines that simulate the circuit's internal blocks. If those subroutines are so time-consuming as to completely dominate the simulation time, the time spent orchestrating their interactions is negligible. Conversely, the simplicity of the blocks makes this orchestration time dominate logic simulation.

2.2 Research Objectives

The logic level simulator CONSIM explores and extends one of the avenues described in the previous section: the use of a multiprocessor composed of non-specialized processing units to speed up a simulator. To attain performance gains, measured as "speedup factors" which reflect how much faster each program runs on several processors than on one, it is necessary to tailor the simulator to the capabilities of the system. In particular, the parallelism hidden in a logic circuit must be preserved or enhanced through all steps leading up to the simulation: description of the circuit to the computer and compilation of this description into a computer-palatable form. This parallelism can then be exploited in the final simulation run to lower the execution time.

The dominant effects in a multiprocessor-based simulator are very different from those in a classical simulator. Identifying and characterizing these effects are primary goals for early research in this area, such as the CONSIM project. This problem is extremely involved: size, character, and partitioning of the circuit to be simulated play roles in the amount of speedup gained by adding processors to the system, as do implementation issues like the structure of the simulation code,

---

as close as they are to the limit.

exploitation of parallelism in that code, process scheduling, etc. These issues are sorted out and explained, and trends are projected based on the analysis. Advances in VLSI processing technology and the resulting growth in IC gate count make the size effects particularly important.

The efficiency of a particular flavor of parallelism is highly system-specific. CONSIM is tailored to Multilisp running on Concert. Its choices, results, and predictions are not universal. For instance, Multilisp's speed in its present incarnation doesn't exactly rival that of ZetaLisp running on a Symbolics 3600. Waiting 1.5 seconds for a 20-cycle simulation of a 10-gate circuit doesn't call for celebration. Although it certainly can be (and has been) a useful design tool, especially when used on a faster machine and/or in tandem with other CAD tools like SCHEMA, CONSIM is *not* fast. Rather, it serves as a tool with which important effects in this relatively new realm can be investigated. It paves the way for future development.

Chapter 3

The CONSIM Environment

3.1 Concert

The growing interest in alternatives to Von Neumann architectures and the resulting wealth of orphan parallel software were the motivations behind the Concert project. Concert's designers recognized the need for an actual multiprocessor testbed for user programs — a need caused by the slowness inherent in a simulation of a parallel program on a sequential machine, and also by the danger that such a simulation might not unearth problems significant in real-time, like memory contention.

Concert is a shared-memory multiprocessor — details of its implementation may be found in [3]. Concert has a memory hierarchy with curious properties, explored in [17], but the details of Concert's architecture do not concern us here.

3.2 Multilisp

Multilisp is an extension of the Lisp dialect Scheme, with special constructs that allow a programmer to designate parts of the program that may run in parallel.

Implementation

A Multilisp program is compiled into the stack-oriented machine language MCODE[1], which is interpreted by a layer of C [18] code running on the Concert processors. A special Multilisp processor, under investigation by Peter Nuth of LCS, would speed matters greatly by removing this interpretive layer. MCODE has special provisions for *futures*, synchronization, and other special features that

[1] The compiler itself is written in Multilisp to further confuse straightforward types trying to understand this incestuous tangle.

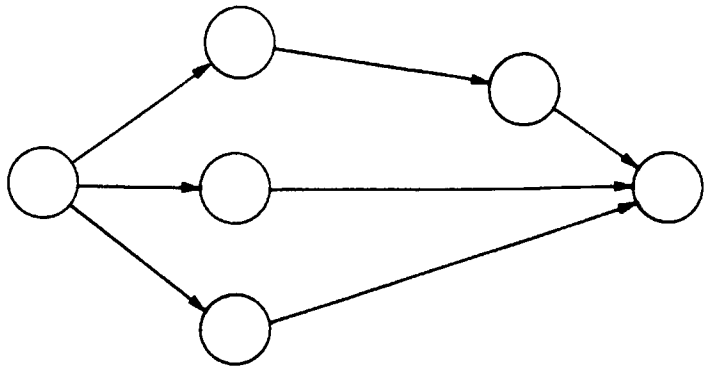
orchestrate concurrency. Garbage collection follows Baker's incremental model as described in [14].

Description

Like other members of the Lisp family, Multilisp is oriented towards symbolic, rather than numerical computation, maintains garbage-collected heap storage, and looks very strange indeed to those used to Pascal or Fortran. Scheme and Multilisp branch off from the Lisp family tree to incorporate lexical scoping, "first-class citizenship" for procedure values, and tail recursion. Multilisp branches yet again, incorporating special constructs which allow the parallelism of the underlying architecture to be exploited. A single processor machine, being sequential, allows no parallelism regardless of program structure. As more and more processors become available, the program execution can branch out if the data dependencies cooperate.

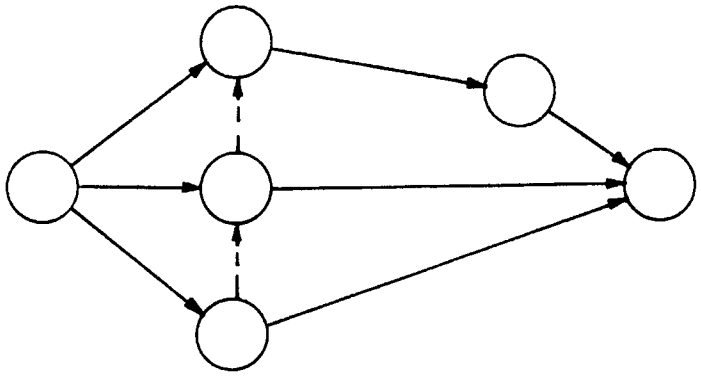
One such construct, the *future*, sends a chunk of computation off in the background in parallel, and at the same time returns a token which can be used as if it were the result of that computation. When the background computation terminates, the token is replaced with the real value. Operations, such as `list`, which do not require explicit values for their arguments can proceed using only the token. Other operations, which need the value for an argument midway through their execution, can at least proceed up to that midway point. When a point is reached in any operation where the token no longer suffices, the operation is suspended until the token is filled in.

Programs have built-in data dependency structures. Since operations depend upon the results of other operations, a partial ordering is imposed on their execution sequence as in Figure 3.1. Sequentialization imposes additional constraints, shown as the dotted lines in Figure 3.2, which comprise a total ordering on the sequence. Any partial ordering permits at least one total ordering and perhaps many more. For example, several otherwise independent operations may depend on the same result. The data-dependency graph would show these operations in parallel, following the operation that produced that result, but on a sequential computer they would be executed one after the other. The additional ordering within the parallel group is an artifact of the implementation. In this case, there are as many total orderings as there are orderings of the parallel group. *Futures* relax these added constraints, so Multilisp programs are bound only by their underlying data-dependencies and not by some tighter, artificial criterion imposed by the implementation.



Data Dependency

Figure 3.1
Data Dependency Graph



Additional Precedence From Sequentialization

Data Dependency

Figure 3.2
Precedence Graph for Sequentialized Program

The Multilisp statement

(future α)

sends the computation α off in parallel and returns a token for its value.

A second construct, the *pcall*, invokes simultaneous evaluation of its arguments. The statement

$$(pcall\ f\ \alpha\ \beta)$$

is equivalent to the call to procedure f

$$(f\ \alpha\ \beta)$$

except that the *pcall* causes concurrent evaluation of f, α, and β. *Pcalls* are not as general as *futures* because they constrain both fork and join of the operations they control. All threads of a *pcall* must terminate before the next program step can begin. This imposes additional precedence constraints, shown in Figure 3.3.

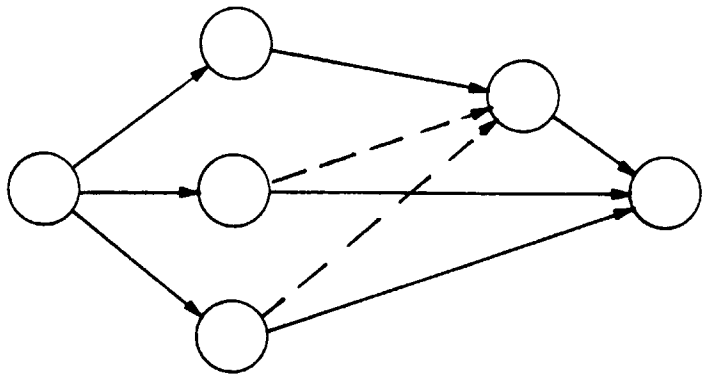


Figure 3.3
Precedence Graph for *Pcalls*

Although the type and mechanism of the extra constraints are very different from those imposed by sequentialization, the overall effect is the same: strictness added by the implementation. *Futures* specify the fork and leave the join up to the background, so the next task is only delayed by the spawning of the *future* and not by the entire computation within the *future*.

Since efficient use of a multiprocessor implies keeping all the task pipelines full, a parallel program should spawn as many tasks as possible. This translates to a program with *futures* in as many places as possible. If *futures* involved no overhead, a very simple heuristic would govern their placement: "put one around *every* expression." However, in the real world, *futures* have some finite overhead and must be placed intelligently. It is only worthwhile to enclose a computation in a *future* if the time gained by running that computation in parallel is greater than that lost via the overhead. It is also useless to enclose a computation in a *future* if the computation will immediately hang up. These issues are of great significance to CONSIM and are discussed in depth in chapters 4 and 5.

Computer Scientists like to abstract things and thus sanitize them, to put a messy thing in a well-defined box and then just interact with the box boundary. A *future* is a good example of such a box, where the insides of the box represent the actions required to send a task off to run in parallel. The book-keeping involved in orchestrating parallel processes is intricate and involved. Communication of arguments and results, keeping memory areas separate, synchronization, and a host of other problems contribute to this complexity. These issues are all hidden by *futures'* implementation. However, the required juggling of overhead, efficiency and data dependencies makes the *future* a less-abstract box. It is difficult to use, which contradicts the very nature of an abstraction. The ultimate abstraction in this case is one in which *future* placement is mechanized — the programmer doesn't have to use them at all and they are inserted by the system. An example of such an automated placement algorithm is discussed in section 4.4.

Chapter 4

Structure and Implementation

4.1 The Test Case

CONSIM's primary intent is to explore how concurrency in a simulation algorithm can best be exploited to accelerate the program on a particular multiprocessor system. A second goal is more practical: to be a useful design tool, a simulator ought to be flexible — to apply to a wide variety of problems. Both of these functions deeply influenced the design decision about the class of circuits to which it applies.

Experiments which investigate the fundamental effects of parallelism may well be clouded by the intricate problems of modern circuit simulation. Delicate models, variable levels of complexity, and all the other trappings of an advanced algorithm [1] may skew the resource usage and obscure the sources and effects of the parallelism. To truly isolate these effects, the problem must be pared down to the bare bones. Only when parallelism is understood in simple examples can it be extended to more complex cases, which are the real applications.

On the other hand, a simulator which only works on a limited number of simple circuits is fundamentally useless. Conclusions drawn from its behavior certainly cannot be held to be representative of the general class of simulators — those we hope to eventually speed up using the knowledge gained by this research.

CONSIM's range of applications — the circuits to which it applies — is thus a compromise between the complexity and generality of a flexible model and the ease of experimentation of a simple one. It is restricted to logic circuits which fit into the Finite State Machine (FSM) structure for sequential circuits. This model is by no means universal, but it *is* very widely applicable.

Any sequential (clocked) logic circuit can be modelled using the FSM model that

[1] Like those discussed in chapter 2.

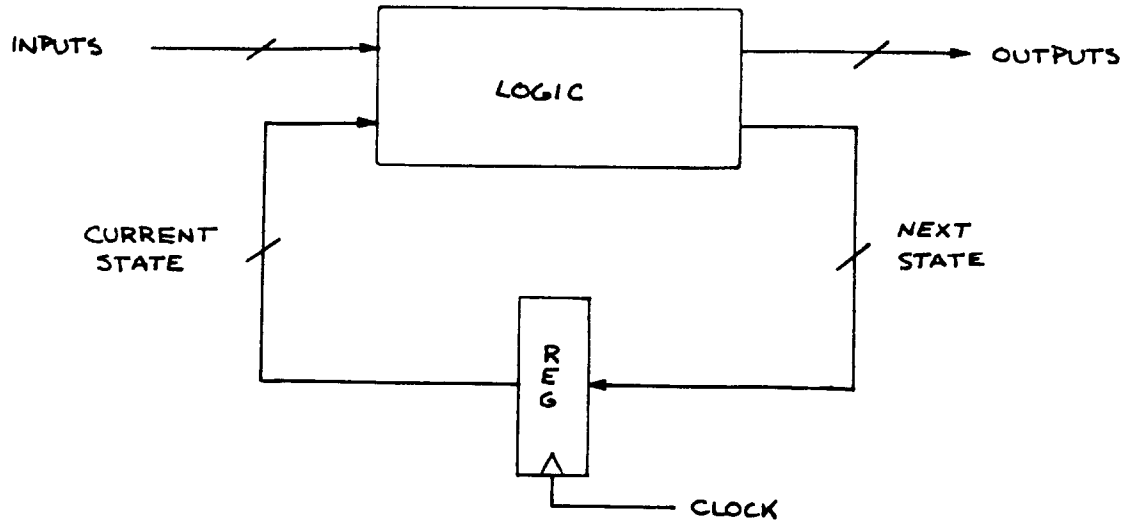


Figure 4.1
Finite State Machine Model

appears in Figure 4.1. The model consists of clocked registers [2] and combinational logic, which may not contain any signal path loops. The "state" of the circuit appears on the output lines of the registers; the "next state" is computed from the present state and the external inputs by the combinational logic and is presented on the input lines of the registers to be loaded on the next clock tick. A sequential circuit with N state variables has 2^N possible states. All of the blocks in this model are fully unilateral — the signal path flows only from input to output and not in reverse. This absence of feedback means that iteration is not required to reach the solution. FSMs fall in two categories according to the role played by their logic inputs. The outputs of Moore machines only depend upon the *state*, while in a Mealy machine the outputs are functions of the *current inputs* as well. The distinction is one of form only — a Moore machine can be constructed to fulfill any function that a Mealy machine performs, but it generally requires more states. Since this implies more labor for the designer, more hardware in the design, more power dissipated while the machine is running, and more effort involved in specifying and simulating the circuit with a CAD program, we choose to specify the Mealy model. Any logic circuit which can be cast into the form of Figure 4.1 can be simulated under CONSIM.

The FSM is not only a powerful model, but also an ideal test case for investiga-

[2] The degenerate case where the registers are missing is called the fundamental mode. Issues involving such circuits will not be addressed in this thesis.

tion of simulator concurrency. There is a great deal of parallel activity in all blocks just after the clock transition. In hardware, the clock period is chosen to synchronize this: by the time the next clock edge arrives, all transient activity is settled out and the outputs are ready. This behavior pattern has promising implications for concurrent simulation. CONSIM's task is to *expose* and *exploit* this parallelism in its most useful form.

4.2 General Structure of CONSIM

CONSIM consists of two main sections: the compiler and the simulator. The compiler translates the user's description of a circuit into an internal form used by the simulator. The simulator takes as inputs this "compiled" circuit, the initial state, and a list of inputs, organized by clock cycle. It returns a list of outputs (and/or machine state), also organized by clock cycle.

Circuit primitives within CONSIM are general clocked registers and functionally-described blocks of logic. Signal variables within and between the blocks take on logic level values: 0, 1, X (undetermined) and Z (high impedance.) No timing information is computed. The circuit is specified by the user in a hardware description language (HDL). The HDL is described in the next section. The compiler transforms this description into a Multilisp procedure which performs the function of the "combinational logic" block in the FSM model (Figure 4.1) during one clock cycle: given present state and inputs, it returns next state and outputs. The simulator maintains cycle information, synchronizes inputs, and calls this procedure once for each simulated cycle, feeding back each cycle's "next state" output to the following cycle's "current state" input. This structure exposes two levels of parallelism: signal-level parallelism (analogous to the circuit's inherent concurrency) *within* the single-cycle subroutine and a kind of "macroparallelism", unlike anything in the hardware behavior of the circuit, which results when entire *cycles* are computed in parallel. *Futures* solve the data dependency problem — that inputs to each cycle depend upon the last cycle and thus are not immediately available if these cycles are called concurrently.

4.3 The Hardware Description Language and the Compiler

A language used by humans to describe circuits to computers must be flexible enough to encompass all possible circuits that its users may wish to describe. It must also be precise, or the computer's understanding of the circuit may differ

vastly from the designer's. Since levels of abstraction are common and powerful design aids, the language must handle blocks of all sizes and levels of functionality with equal ease. If the block definition syntax becomes clumsy at either end of the complexity scale, the user interface suffers. If the implementation of the language requires an intermediate compilation step, like translating all logic into NAND gates, the computer performance (run time) suffers.

CONSIM's hardware description language (HDL) has a Lisp-like structure. The HDL compiler is written in Multilisp and thus operates most easily on Lisp data objects. Parsing issues are also much less of a problem in list structures.

A description of a circuit consists of three sections: the specification of the circuit name, the functional description, and the interconnect information. Information within these sections appears in the form of Lisp expressions. The first part identifies the circuit. The second part specifies the function, as defined from the terminals, of each block in the circuit. The third part specifies how those terminals are connected.

The HDL compiler translates this information into a Multilisp procedure which simulates the input/output behavior of the combinational logic block in the FSM model. This procedure, called the single-cycle procedure, is called once by the simulator for each cycle to be simulated. The mechanics of this translation are discussed later on in this section and the algorithms underlying the translation are discussed in the next section. [3]

A functional description of a block defines the names and number of its input and output terminals, the types of variables that exist at those terminals, the action the block performs upon those variables, and (of course) the block's name. A library of standard blocks exists within CONSIM; if a non-library block is used, it must be explicitly defined in a separate statement in the second section of the HDL description.

The syntax of a block definition statement is

```
(defmod name fcn-name (inputs) (input formats) (output formats) (body))
```

`Defmod` is the HDL module (block) definition command. `Name` is the block's name. `Fcn-name` is the name under which the Multilisp function is to be defined. The `format` fields, which are used during the compilation for error checking, specify the

[3] In the best incestuous traditions of computer science, the compiler itself contains *futures* to speed it on its way.

number and types of the inputs and outputs. Formats may be Boolean (B), integer (I), or character (C), depending upon the block. A gate would operate on Boolean signals, a 16×16 multiplier would multiply integers, and a dot-matrix display driver would use characters. Obviously, ANDing a 16-bit integer signal and an ASCII character is an error. These types of errors are detected by the compiler when it compares the format fields of all the blocks according to their interconnections. Body is the body of the Multilisp function the block is to perform, specified in Polish-ordered Boolean equations. When the module definition is executed, body is translated into Multilisp and encapsulated in the function `fcn-name`. A data structure called a block descriptor, used by the HDL compiler, which contains name, type and number information is also created by this call. An analogous statement, **defmodlib**, is used in a separate file to define library blocks. In cases where the block or function defined by the user carries the same name as a library version, the user's definition takes precedence. The structure of a Lisp environment can cause problems if more than one circuit is being simulated in one workspace and both contain different versions of the function "f" — the last-loaded version overwrites the other.

For example, a block named f which has two Boolean inputs x and y and one Boolean output z = (x AND (NOT y)) would be defined by the following statement:

```
(defmod f b (x y) (B B) (B) (and x (not y)))
```

This command creates the Multilisp function b:

```
(defun b (x y) (and x (not y)))
```

which returns z as specified and will be called every time the outputs of block f are to be computed.

Each line in the interconnect section lists the names of the variables connected to the terminals of a block. The order of the names within this list must follow the order of the terminals as defined in the **defmod** statement. There is one line for each block in the circuit and the lines may be entered in any order. Every inter-block variable must have a unique name — the HDL compiler error check procedure detects duplicates. It also detects block outputs that are not connected to any input, as well as block outputs that are connected together (unless they are tristate outputs). The type fields in the block descriptors for all the block terminals connected to each variable are compared to detect the type errors discussed above (i.e., ANDing an integer and a character, NANDing four signals with a three-input

gate or three signals with a four-input gate.) Note that the interblock data take many forms, and thus internal and external vectors of variables are heterogeneous. The external variables, which connect the circuit to the outside world, have special names [4] but they are otherwise treated just like the internal signals.

The general format of an interconnect line is the expression:

```
((block-id) block-type (inputs) (outputs))
```

`Block-id` is an optional identifier. It may be used to differentiate between different instances of a certain block type. For example, three XOR gates within a single circuit may be labelled "xor1", "xor2", and "xor3" for mnemonic purposes. `Block-type` is the name of the block, as given in the first field of the block definition in the first section of the HDL (or of the library definition.) If the block is not in the library and has not been explicitly defined in the first section, the compiler will complain. The input and output fields contain ordered lists of the inputs and outputs by name.

An example HDL interconnect line defining connections to a 2 input AND gate (called "gate1" because the designer wanted to differentiate it from some other 2-input AND gate which she named "gate2") with inputs m and n and output w is:

```
((gate1) and2 (m n) (w))
```

If no special mnemonic name is required, the HDL line would be:

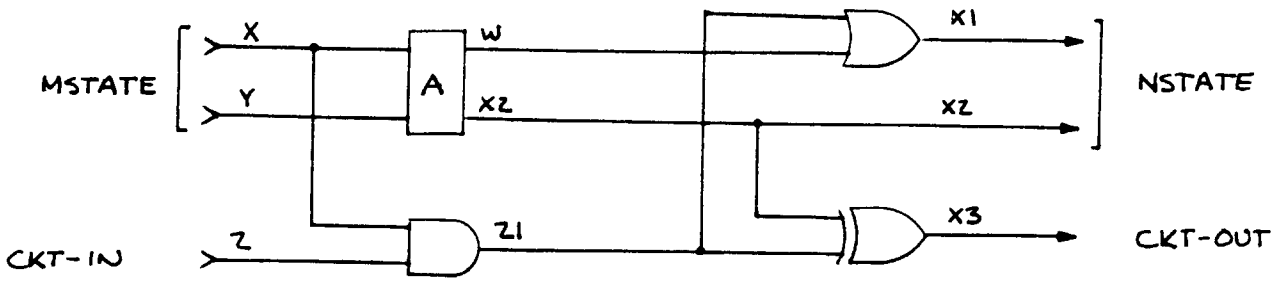
```
(and2 (m n) (w))
```

These code lines assume that the Multilisp function "and2" is either in the library or defined in the **defmod** section of this HDL file. As explained above, if the types of m, n, or w are not those given in the **defmod** call which defined "and2", or if "and2" is undefined, the compiler will return an error.

An Example: The complete HDL description of a circuit.

The schematic and HDL description of the circuit *example1* is shown in Figure 4.2. For this circuit, current state (mstate) and next state (nstate) are two-bit variables. There is one input z, one output x3, several standard gates, and a single non-library

[4] mstate, ckt-in, nstate, and ckt-out



Circuit

```
(circuit-name example1)
(defmod A f-A (a1 a2) (B B) (B B)
(list (and (not a1) (not a2)) (and (not a1) a2)))
(mstate (mstate) (x y))
(ckt-in (ckt-in) (z))
(A (x y) (w x2))
(and2 (x z) (z1))
(or2 (w z1) (x1))
(xor2 (x2 z1) (x3))
(nstate (x1 x2) (nstate))
(ckt-out (x3) (ckt-out))
```

HDL Description
Figure 4.2

block called A. Block A has two Boolean inputs a1 and a2 and two Boolean outputs (and (not a1) (not a2)) and (and (not a1) a2).

Note the interplay of internal and external variables: mstate, the 2 bit "current state" vector, fans out to the circuit variables x and y. Ckt-in, the single bit external input, connects to the circuit variable z. Nstate, the 2 bit "next state" vector, is composed of the circuit variables x1 and x2. The external single-bit output, ckt-out, is the internal variable x3. The external variables must be decomposed into internal variables before they are used and must be explicitly reassembled before being returned by the procedure. This is the general rule for the treatment of composite signals, and external signals are defined to be composite. Internal and external signal connections follow the same syntax, which is a little repetitive but allows generality. The interconnect lines are shown in data-dependent order for clarity, but their actual order in the HDL file is immaterial, since the compiler sorts them into data-dependent order before generating the single-cycle procedure.

Finally, note that the defmod statement defining the A block imposes a specific order on its inputs and outputs. The HDL statements (A (x y) (z w)) and (A (y x) (z w)) have different meanings. [5]

4.4 Code Structure and Future Placement

The various algorithms which the HDL compiler uses to translate an HDL circuit description into a Multilisp single-cycle procedure create different forms and amounts of parallelism in the code. Heuristics which govern how *futures* are used to exploit this parallelism also produce varying results. The trick is to find the best combination.

The simplest way to generate the single-cycle procedure from the HDL code is a one-to-one mapping. Each HDL interconnect line maps to a single call of the Multilisp function, created by the defmod statement, which simulates the input/output behavior of a block. Each intermediate signal in the circuit maps to a temporary local variable in the procedure. Code generated using this algorithm is highly vertical — a long string of simple statements.

```
(defun example1 (mstate ckt-in cyc-num)
        (let*  ((g1 mstate)
                (x (field 1 g1))
                (y (field 2 g1))
                (g2 ckt-in)
                (z (field 1 g2))
                (g3 (f-a x y))
                (w (field 1 g3))
                (x2 (field 2 g3))
                (z1 (f-and2 x z))
                (x1 (f-or2 w z1))
                (x3 (f-xor2 x2 z1))
                (nstate (list x1 x2))
                (ckt-out (list x3)))
               (list cyc-num nstate ckt-out)))
```

Figure 4.3
Single-Cycle Procedure
For Circuit Of Figure 4.2

The single-cycle procedure generated from the HDL description of Figure 4.2 via one-to-one mapping is shown in Figure 4.3. The top line of the procedure shows

[5] Just as swapping the data and address inputs to a RAM chip gives different results.

the inputs. `Cyc-num` is the cycle number which is used for book-keeping purposes. *Let** is the Multilisp way of defining local variables. *Let**, unlike the standard Lisp *let*, is sequential: variables computed in the body of the *let** may be used later on in that statement. *(let* ((x <expr1>) (y <expr2>)) (<expr3>))* binds the value of *<expr1>* to the local variable x, then binds the value of *<expr2>* to y in the resulting environment, and finally evaluates *<expr3>* in the environment that remains after the full succession of bindings. [6] X may be used in *<expr2>* to compute y. The last line, `(list cyc-num nstate ckt-out)`, is the returned value. Any intermediate result which is composite (i.e., a list containing several variables) is first assigned to a temporary variable (`g1`, `g2`, etc.) and then decomposed using the function *(field n alist)*, which picks out the n^{th} field of *alist*. All external variables are composite by definition.

The definition of a local variable uses computer resources in two stages. The expression whose result is to be assigned to the variable is evaluated, then the allocation of storage space and binding take place. Both steps require CPU time and memory. If an intermediate variable is used in several places in a process, it makes sense to explicitly define that variable — to compute its value and store it in a slot which can later be called up by name — when it is first used, rather than to recompute the value every time the variable is used. However, explicit definition of a variable that is only used once is superfluous. The time and memory spent in the assignment step are wasted, since the slot will never be accessed again. The CONSIM code shown in Figure 4.3 exhibits this kind of wastefulness: the variables `z`, `w`, `x1`, and x3 appear in the code only once after their definitions.

This waste can be eliminated by "condensing" the code. If a variable occurs only once later on in the code, that instance is replaced with the expression used to synthesize the variable. Condensed code is horizontal — a shorter sequence of more complex statements. Since the relative "costs" of computing the expression versus making the assignment differ from computer to computer, the algorithm's efficiency depends upon the implementation. On a computer where storage allocation was extremely slow and computation extremely fast, it might be more efficient to allow two, three, or even more later instances before an explicit definition is made.

The condensed version of the code for the circuit in Figure 4.2 is shown in

[6] The number of variables in a *let* is limited to 512 on the VAX and 256 on Concert. This effectively limits the size of the circuits tested in this particular implementation, but is not a general problem.

Figure 4.4. Note that the intermediate definitions of z, w, x1, and x3 have disappeared; those of g1 and g2 remain because they correspond to the composite external variables mstate and ckt-in. Although this example is too small to effectively demonstrate the full shift in aspect ratio, it is clear that the code is shorter and wider.

```
(defun example1 (mstate ckt-in cyc-num)
        (let*  ((g1 mstate)
                  (x (field 1 g1))
                  (y (field 2 g1))
                  (g2 ckt-in)
                  (g3 (f-a x y))
                  (x2 (field 2 g3))
                  (z1 (f-and2 x (field 1 g2)))
                  (nstate (list (f-or2 (field 1 g3) z1) x2))
                  (ckt-out (list (f-xor2 x2 z1))))
                (list cyc-num nstate ckt-out)))
```

Figure 4.4
Condensed Single-Cycle Procedure
For Circuit Of Figure 4.2

Futures must be placed in this code for the parallelism to be exploited. Recall that *(future <expr>)* sends the computation *<expr>* off in parallel and returns a token that can be used as if it were the result of that computation. A naive heuristic would suggest that every expression be enclosed within a *future*. This "optimistic" approach trusts the implementation to find and exploit any parallelism and to simply sidestep extra operations if a spurious *future* is encountered. Applying this heuristic to the condensed single-cycle code for circuit *example1* yields the version in Figure 4.5.

This approach has one very large problem: it falsely assumes that *futures* incur no overhead. Extra operations cannot simply be sidestepped when a useless *future* is executed. For each expression, the algorithm must weigh the time gained by parallel evaluation against the overhead associated with the *future* operation. Only if the former outweighs the latter should a *future* be used around that expression. This comparison varies from computer to computer as well, so a particular grouping of *futures* which work well on Concert may not (and does not) work well on the Symbolics 3600 Lisp Machine's sequential version of Multilisp. On Concert, a *future* takes 4 times as long as a procedure call; on a 3600, it is computationally much more expensive and takes about 40 times as long as a procedure call. Thus, simulations on a 3600 of Concert's behavior reflect this ratio. The code is tailored to Concert

```
(defun example1 (mstate ckt-in cyc-num)
        (let*  ((g1 mstate)
                (x (future (field 1 g1)))
                (y (future (field 2 g1)))
                (g2 ckt-in)
                (g3 (future (f-a x y)))
                (x2 (future (field 2 g3)))
                (z1 (future (f-and2 x (future (field 1 g2)))))
                (nstate (future (list (future (f-or2 (future (field 1 g3)) z1)) x2)))
                (ckt-out (future (list (future (f-xor2 x2 z1))))))
               (list cyc-num nstate ckt-out)))
```

Figure 4.5
Condensed Single-Cycle Procedure With Naive Futures
For Circuit Of Figure 4.2

and its efficiency is predicated upon Concert's balances.

The efficiency of a placement algorithm depends upon how well it predicts the effects upon run time of dispatching a particular task in parallel. Some predictions are obvious: *(future 3)* is useless, since no computation is required to evaluate the atom 3 and thus no task exists to run in parallel. The statement *(+ (future (* 3 4)) 7)* is also useless, since the "+" operator requires the values of all operands immediately, and the operation *(* 3 4)* is too small to warrant the overhead of being sent off in parallel. A statement like *(+ (future (f 100)) (future (g 100)))*, where the functions *f* and *g* are complex, *would* be useful, since the two arguments to the + can be efficiently evaluated in parallel. [7] Other predictions are much less obvious — see [14] or [11] for further discussion. The overall trend can be informally summarized: the more complex <*expr*> is, the more efficient *(future* <*expr*>*)* becomes. A *future* must have enough to chew on before it becomes useful.

This informal definition can be formalized and extended to the point where it can be used as a mathematical rather than a subjective criterion for the efficiency of a particular *future*. The automatic *future* insertion program *Savant*, written by Sharon Gray [11], automates this criterion and achieves impressive results. Two properties, *quickness* and *strictness*, are defined in terms explicit enough to be used by *Savant's* compiler. *Quickness* is roughly analogous to the informal "enough to chew on" definition: "An expression is considered *quick* if it can be evaluated in less time than the time it would take to return a *future* for that expression." [11] Any combination of applications of Multilisp's primitive functions is defined as *quick* for

[7] Note that none of these "useless" *futures* cause a wrong answer — they just delay the correct answer's arrival.

Savant's purposes. *Quickness* is also affected by the context of evaluation. The function *(car x)* is *quick* if x has been touched already (i.e., if the value of x has been used) but not if x may be bound to an undetermined *future*. *Savant* maintains context information in order to make these decisions. *Strictness* formalizes the precedences between a function and its arguments. A function that is *strict* with respect to a particular argument needs its actual value and not a token. If a function is not *immediately strict* with respect to a particular argument, execution can proceed, with a token in place of that argument, up to the point where the actual value is required. The further along that point lies in the execution of the function, the more efficient a *future* around that argument becomes. Applying *Savant* to the condensed code for circuit *example1* yields the code in Figure 4.6.

```
(defun example1 (mstate ckt-in cyc-num)
       (let*  ((g1 mstate)
               (x (future (field 1 g1)))
               (y (future (field 2 g1)))
               (g2 ckt-in)
               (g3 (future (f-a x y)))
               (x2 (future (field 2 g3)))
               (z1 (future (f-and2 x (future (field 1 g2)))))
               (nstate (list (future (f-or2 (future (field 1 g3)) z1)) x2))
               (ckt-out (list (future (f-xor2 x2 z1)))))
              (list cyc-num nstate ckt-out)))
```

Figure 4.6
Condensed Single-Cycle Procedure With Savant's Futures
For Circuit Of Figure 4.2

Savant is very conservative in this example and thus uses almost as many *futures* as does the optimistic heuristic. It makes no assumptions about the determinedness and thus the *quickness* of the procedures's inputs, so it puts *futures* around the *field* statements which decompose those inputs (i.e., `(future (field 1 g1))`). The function *field* is not a Multilisp primitive, so its instances are enclosed in *futures*. If the primitive functions *car*, *cadr* and *caddr* had been used to take the lists apart, and mstate was known not to be a *future*, the operations would be *quick* and no *futures* would have been used. The functions `f-or2`, `f-xor2`, `f-a`, and `f-and2` are, like *field*, assumed not to be *quick*. *List*, however, is known to be *quick* and not *strict*, so its invocation receives no *future*.

The vertical Multilisp code consists of a long succession of small and probably *quick* expressions. The expressions in the horizontal code are more substantial and thus less *quick*. The naive *future* placement heuristic ignores the very real problem

of overhead, while *Savant* takes careful account of it. From this analysis, the best combination of code generation and *future* placement algorithms would appear to be *Savant* applied to horizontal code. This hypothesis is verified by the results in chapter 5, although not as resoundingly as one might expect.[8] The verification and the reasons for this apparent inconsistency are discussed in great detail in section 5.3.

4.5 Using the Simulator

Once the compiler has produced the single-cycle procedure and *Savant* or some other entity has judiciously placed *futures* in that procedure, the simulator is brought into action. This is by far the simplest of CONSIM's tasks, although because of its repetitive nature it may be the most time-consuming.

The simulator performs the calls to the single-cycle procedure and communicates with the outside world. Simulator inputs are a compiled and "futured" circuit (the single-cycle procedure), the initial machine state, a list of circuit inputs which contains one vector of variables (ckt-in and mstate) for each cycle to be simulated, and commands from the user. The initial machine state and the elements of the circuit input list must match the HDL templates in number, order and type, or the simulator returns an error message. A command from the user tells the simulator how many cycles to run. If the input list doesn't contain enough elements, the simulator goes as far as possible, then stops and reports the error.

It may become apparent from the first round of output that the design is flawed. The user would then make a change in the HDL file, recompile the circuit, run the single-cycle procedure through *Savant*, and try the run again. To help speed up this painful process, the compiler itself has been accelerated with *futures*. For a small circuit, where the *Savant* run takes longer than the performance gained by the few *futures* it can place in the code, it would be efficient to skip that step during the design cycle. Once the circuit works, the simulation runs presumably become longer and *futures* save time.

A useful and interesting extension of parallel computing is to enclose entire cycles in *futures*. This is accomplished by enclosing the call to the single-cycle procedure in a *future*. This might not seem advisable at first glance, because of the data dependencies: each cycle depends upon the "next state" computed in the

[8] Code condensation turns out to be not as important as it initially appears.

last cycle. However, at least *some* computation can proceed without *every* result from the previous cycle; requiring each cycle to terminate completely is akin to the extra requirements imposed by a *pcall*, as discussed in section 3.2. The parallelism within the cycle has a hardware analog but the "macroparallelism" of calling cycles in parallel is purely a software phenomenon. Its effects are nonetheless extremely powerful — in fact, it actually multiplies the effects of the cycle-internal parallelism. The mechanics and results of this technique are discussed in section 5.3.

Since *print* does not require the explicit values of its arguments, a *future* left unevaluated at the end of a simulation run, such as the final cycle's outputs, is printed out as is — in the form of Multilisp's internal undetermined *future* representation. Not only is this information incomplete, but the time required to return it also does not reflect the full computation. Background processes, assigned to fill in the values of those *futures*, continue to run after the computation "returns". These orphan processes can persist into the next process invoked, possibly damaging its results [9] and certainly altering the statistics of its run time. To return actual values and account fully for computation time, all elements must be touched during the final return process. The touching operation adds a certain amount of unavoidable overhead; the significance of this amount is explored in section 5.3.

Several additions are required to make CONSIM user-friendly enough to serve as more than just a testbed for exploring parallelism in simulation code. It now reports cycle number, machine state, and outputs for every cycle simulated. Users who don't want to wade through all this information might wish to instruct the simulator to omit parts of this report. Someone who is uncertain that his circuit functions properly would find breakpoints and "variable-watch" functions useful. The simulator runs until a "watched" variable changes or assumes some defined value, then stops and notifies the user. Implementing this may be tricky: the condensation algorithm obliterates some of the intermediate variables. Loops in the logic are, by the FSM model definition, not permitted. If present, they throw the compiler into a tailspin. A more graceful response would be nice. A single-step mode would be helpful too, although it completely bypasses the performance gains made by calling cycles in parallel. Another unrelated but effective modification is the interface between CONSIM and the expert development system SCHEMA [35]. SCHEMA provides extensive graphics support for schematic circuit entry and its internal circuit representation is not vastly different from the CONSIM HDL. A simple

[9] if side effects, which can occur in Multilisp, but are not used in CONSIM, were to collide.

translation program would obviate human entry of the HDL description — a step fraught with errors — for circuits designed on SCHEMA. One would simply display a circuit on a SCHEMA workstation and type "simulate."

Chapter 5

Results and Discussion

5.1 Procedures

Four test circuits were simulated with CONSIM. Examples of different sizes were chosen to represent a variety of common circuit types. Two small examples were HDL-coded and compiled by hand before the compiler was automated. Experiments with these examples and the main simulation loop (which was the first CONSIM section implemented) guided the development of the code structure and *future* placement algorithms. After the compiler was automated, HDL descriptions for two larger circuits were compiled and the resulting single-cycle procedures were tested with combinations of code structure and *future* placement algorithms. Example 1 is purely illustrative and has no useful function. It contains 10 gates. Example 2 is a controller for a 1960 vintage Coke machine (which sells Coke for 25 cents) and it contains 25 gates. The combinational logic block of the FSM model in Example 3 contains an ALU with 58 gates, equivalent to the TTL 181 chip. This machine can, for example, be used as a 4-bit counter if the next state is connected to the current state plus one. Example 4 is the same as example 3, but the ALU block is 16 bits wide and contains four times as many gates. Schematics and state transition diagrams for these examples appear in appendix 1.

The next-state and output logic for each example was derived using standard digital design techniques (Karnaugh maps, etc.) A description of this logic, given in the HDL format specified in section 4.3, was entered into a file. This description was then compiled — by hand at first, then by the automated HDL compiler — into a Multilisp single-cycle procedure following one of the code structure algorithms described in section 4.4. *Futures* were then placed in this code using one of the placement algorithms discussed in the same section, and the finished procedure was stored in another file. A purely sequential version, without any *futures*, was also used for comparison. The HDL description and Multilisp code resulting from application of various combinations of code structure and *future* placement algorithms

to example 1 appear in appendix 2.

The various versions of the Multilisp code were tested on two computer systems: a VAX 11/780 and Concert. Until it was certain that the compiler produced syntactically and functionally correct code, and until the HDL description of the circuit was known to match the desired function, the VAX's sequentialization of the code made debugging easier.[1] Multilisp is implemented by a program called XML, itself written in Multilisp.

The histogram facility in XML, used to simulate Concert on the VAX, computes how much actual parallelism would be used by a program given 50 Concert processors. These data, together with the amount of actual VAX CPU time that was used, predict how fast the program will run on the Concert hardware itself. XML predictions were gathered on the VAX for the four examples, and then the Multilisp function *time* was used to gather statistics on Concert itself for simulation runs of various lengths.

Single-cycle procedures were tested both in cycle-serial and in cycle-parallel simulation runs. If a *future* has not been "filled in" by the time the program terminates (e.g., if the program returns a result, like a list, that can contain tokens), the Multilisp token itself will be returned; program termination does not force a global join. In CONSIM, this occurs most frequently when cycles are called in parallel because more *futures* are present. Each element in the returned list is touched using a tree-walk algorithm to force all *futures* to be filled in. Examples with no *futures* were tested with and without the tree walk to determine the scope of this overhead.

5.2 The Issues

This research investigates a number of issues, some of which interact strongly with several of the others. A simulator is a software entity which mimics a hardware entity; which of these realms' laws dominates the program's behavior? Is program speed affected more strongly by hardware topology or by programming algorithms and techniques? [2] Which programming techniques are important? Should *futures* or *pcalls* be used and, if so, what heuristics should govern their placement? What code structure contains the most exploitable parallelism? What are the effects upon

[1] Debugging code which runs in no apparent deterministic order can be difficult.

[2] This issue is of particular interest to the author — a hardware designer misplaced into the software arena!

efficiency of circuit size and simulation run length , and how does this bode for real-world applications and trends? Finally, how reliable, duplicatable, applicable, and explainable *is* all this anyway?

In the very simplest conceptual model of concurrent computation, simulator performance improves from that in the sequential case by a factor proportional to the parallelism in the circuit under test — a hardware effect. This model assumes that no software overhead is required to control the concurrency, and that software complexity and hardware complexity are somehow linked: a complex logic block will require lots of CPU time to simulate. The former is clearly an idealization. Given levels of abstraction, the latter assumption may also be flawed. The simplified model also assumes that passing values between blocks — simulating the wiring, as it were — takes no time. When the computations within the blocks are complex, this approximation is valid. However, when the insides of the blocks are simple, "simulating the wiring" dominates — a software effect. A mixture of hardware and software effects governs software efficiency.

The following brief summary of chapters 3 and 4 is presented for those impetuous readers who skipped directly to "Results." Multilisp provides two tools, *pcall* and *future*, which allow a programmer to exploit parallelism. One or both may be useful in the CONSIM application. These two constructs differ in how they use the parallelism of the underlying architecture and also in how they are used in a program. Ideally, a compiler should insert them and the programmer shouldn't have to learn the intricacies of their use. The efficiency of an algorithm which inserts *futures* in a program depends upon its intelligence and the structure of the code to which it is applied. Code generated by the HDL compiler takes two forms: vertical and horizontal, so-called because of the aspect ratio of its printed form. The former is produced by a one-to-one mapping between HDL and code. The latter is condensed to improve software resource usage and efficiency. Each exposes a different flavor and amount of parallelism. This affects the success of the *future* placement algorithm used to transform that code.

A physical analogy provides a framework from which to reason about the actions of a multiprocessor. Consider a tank of liquid with a number of outlet pipes, where the liquid represents the pool of tasks to be executed and each pipe corresponds to a processor. A more powerful (faster) processor maps to a pipe with a wider mouth. The performance of an actual multiprocessor also depends on the execution time after a task has entered a pipeline. This is a function of the size of the task, which is in turn affected by the code generation algorithm. Given a particular

distribution of tasks, the procedure which empties the reservoir the fastest gives the shortest execution time. If a system contained pipes of different diameters, the routing of tasks to pipes would be critical — the larger tasks would be directed to the larger pipes to make the execution times of all tasks comparable. Concert consists of identical processors, so its pipes have identical diameters and task routing is immaterial. The Multilisp implementation fits this pattern: the task at the head of the queue is grabbed by the *first* free processor. The order in which tasks are placed on the queue is nondeterministic because it depends upon the order in which *futures* were spawned and in which they were completed. Each *future* may hang up any number of times as it encounters tokens among its own arguments. A final addition to the model concerns *futures'* overhead, which may be modeled as liquid added in proportion to the amount which enters each processor.

One problem with this model is its lack of a graceful analogy for a quantized task. Another model, which does not suffer from this problem, is a typing pool of n secretaries. Each secretary corresponds to a processor and each job — letter, paper, envelope — corresponds to a parallel task. The reservoir in the last model maps to the "in box", where jobs to be typed are stacked. Hanging up has a plausible analogy in this model. If a table of data has not been filled in, the secretary can continue to type until that table is reached, but then must put that paper aside until the table is filled in by a "background process". In this case, this process is the author of the paper, who can be working, in parallel, on the data for that table, which represents the *future*. Overhead is represented by travel time between in box and typist.

A simple throughput argument [14] shows that parallel computation is only really efficient when the task-pipelines are kept full. The *future* placement algorithms determine the optimum task "size" that fills the pipe width-wise; the circuit size and simulation run length create the tasks which fill the reservoir. Increasing circuit size within the same hardware family or lengthening a simulation run have similar effects: the *number* of tasks increases without changing their *statistical distribution* (i.e., average run time.) This assumption is based on the placement algorithm's habit of dicing any circuit into chunks of roughly the same size. When the circuit and run length are too small to fill all the pipes, the speedup, defined as

$$\frac{\text{execution time on one processor}}{\text{execution time on } n \text{ processors}}$$

continues to rise, since the numerator rises with the number of tasks and the de-

nominator doesn't, but the system is underutilized. Processors are idle. The level in the reservoir is so low that all liquid immediately finds a vacant pipeline and exits. Once there are enough tasks to fill all the pipes (n tasks for an n-processor system), the entire system comes into action and the parallelism is maximum. Adding more tasks above this level causes a backup; parallelism is still maximum, but tasks have to wait in the reservoir for pipelines and the speedup is reduced. The mathematics of speedup and runtime evolve from these considerations.

5.3 Results

The execution time for the vertical single-cycle procedure with no *futures*, invoked in cycle-serial mode, is the baseline measurement for these experiments. This version is generated by the simplest of the code generation algorithms. Since it contains no *futures*, it does not utilize Concert's parallelism. Speedup factors for versions with improved code structure and/or *future* placement will be measured against these results. All data are for simulation runs of 20 cycles. [3]

Example	Total Ops	% Parallelism	Real-Time Ops
1	9189	100	9189
2	25424	100	25424
3	74802	100	74802
4	356566	100	356566

Table 5.1
Baseline XML Results — All Examples
Vertical Code With No *Futures*

The results in Table 5.1 were produced by a histogram facility within XML that simulates Concert's performance on the VAX. The "total ops" column shows the total number of operations performed by all processors to complete the computation. The "% parallelism" column shows the speedup factor which would be obtained by running the program on a 50-processor Concert machine. 100% is a factor of *one* speedup — no parallelism gains. The "real time ops" column contains the result of dividing "total ops" by "% parallelism." With no *futures*, one certainly expects no speedup; the 100% factors reflect this and "total ops" and "real time ops" are equal.

[3] The effects of run length are discussed later in this section.

"Operation" is ambiguous; XML counts both a simple add and a complex procedure call as one operation, but their actual Concert execution times are very different. The average operation time in a program is a function of the program's instruction mix.[4] XML results are difficult to interpret in light of this operation time uncertainty. The "% parallelism" is the statistic that relates most closely to Concert for this purely sequential case. Unfortunately, the "real time ops" is the statistic which is of the greatest interest, since it mirrors the cumulative speedup, including parallelism overhead.

The execution times of the same programs on Concert itself are shown in Table 5.2. Concert results, like XML results, are for 20 cycle runs unless otherwise noted. Results vary with the number of processors in the system, as discussed at the end of this section. A 14-node system was used in these trials simply because that was the operational limit as of this writing. Each value shown was averaged from two time trials to filter out background system effects like process scheduling and garbage collection. No actual garbage collection occurred during these particular trials [5] but the two-sample format is followed throughout this paper for consistency.

Example	Time in Seconds
1	3.000
2	7.400
3	16.717
4	78.867

Table 5.2
Baseline Concert Results — All Examples
Vertical Code With No *Futures*
14 Processors

The "Total Ops" in the XML data and the time in the Concert data are very closely correlated. This correlation does not persist as the code structure and placement algorithms change [6] for a variety of reasons, including the operation time un-

[4] One way to get a handle on this average operation time is to factor in the total VAX CPU time required to execute the simulation.

[5] *Futures* are most allocation-intensive operations in CONSIM and thus cause the most garbage collection. Since none appear in this series of examples, very little garbage collection occurred.

[6] i.e., the Concert data do not fit the XML prediction. The data in the appendices

Example	Time (sec) Vertical Code	Time (sec) Horizontal Code
1	3.000	3.067
2	7.400	8.533
3	16.717	16.075
4	78.867	74.600

Table 5.4
Concert Results — All Examples
Vertical Code vs. Horizontal Code
14 Processors

certainty discussed above. XML's statistics do not reflect garbage collection, which is a distinct advantage, since the execution times of garbage collection algorithms vary widely. However, the uncertainty arising from its approximations — typically ±10% — overshadows its value in this application (beyond preliminary tests and debugging). Hereafter, XML data will only be used for comparison, except in preliminary comparisons of the various *future* placement algorithms, where the XML statistics were the only data available. XML histogram data for all versions of the single-cycle procedure and main simulation loop for all four examples are shown in appendix 3. All Concert time trial results for the same cases appear in appendix 4.

Condensing the code has interesting effects. Both XML and Concert data are shown, in Tables 5.3 and 5.4 respectively, to illustrate the disparity between the simulation data gathered on the VAX and the real data gathered on Concert. Data points for vertical code are shown for comparison in all cases. The "total ops" and "% parallelism" data are not shown in the first table; since this code is sequential, parallelism is universally 100% and the other two columns are equal.

Example	Real Time Ops Vertical Code	Real Time Ops Horizontal Code
1	9189	8259
2	25424	24476

Table 5.3
XML Results — Examples 1 and 2
Vertical Code vs. Horizontal Code

demonstrate this variance.

The XML data predict that the horizontal code will execute 1.113 times ($\frac{9189}{8259}$) faster than the vertical code in example 1 and 1.039 times ($\frac{25424}{24476}$) faster in example 2. Concert, however, has a different opinion. For both of these examples, the horizontal code actually runs slower (0.978 and 0.867 times as fast as the vertical code, for examples 1 and 2 respectively.) This lack of a correct prediction may be a consequence of XML's ±20% error, but Concert's results are somewhat puzzling. The horizontal version of a single-cycle procedure is a *subset* of the vertical code: no operations are added during condensation and thus it should run as fast as or slightly faster than vertical code. This may be a result of the way Multilisp is compiled into MCODE.

The Concert results hold no other great surprises. Where the circuit and the procedure are small (examples 1 and 2) the condensation from vertical to horizontal doesn't seem to eliminate enough computation to override the effect described above. Where the circuit is large and the procedure is long (examples 3 and 4) the effects are felt and the run time shrinks. The decrease is not drastic — 4% in example 3 and 6% in example 4. The amount of shrinkage the condensation algorithm is able to perform, as measured by the reduction in program execution time, does not appear to be a constant function of program size — the percentage grows with increasing size. Overall, the effects of condensing the code, when they *are* helpful, aren't very striking when the code executes sequentially.

Several *pcall* and *future* placement heuristics were tested during the preliminary stages of this project, beginning with "judicious hand placement" and ending with *Savant*, the automated compiler described in section 4.4.

Pcalls are constrained by the call/return structure of their target program and thus are difficult to use: a given program contains many fewer potential *pcall* sites than *future* sites. The best possible combination (as determined by exhaustive process-of-elimination testing of combinations) of *pcalls* and code structure yielded 121% parallelism in example 1 and 126% in example 2, while *futures* far outstripped those levels. *Pcalls* cause treacherous artificial parallelism gains, not reflected in a lowered "real time ops" value. Both branches of a *pcall* must be long enough to require several quanta of CPU time for the construct to be efficient. Thus "padding" one or both branches yields more parallelism. This effect is illusory. The *pcalls* added in combination were themselves acting as padding for the outer *pcalls* and thus made the overall speedup look better than it really was. *Pcalls* force the pattern of the computation to follow the hardware topology; from the results, this would appear to be inefficient. Topology-independent *futures* are more general

and powerful. Indeed, the *pcall* command in Multilisp is simply syntactic sugar — implemented with *futures*. For these reasons, *pcalls* were omitted from the CONSIM picture very early on.

The XML results for "judicious hand placement" of *futures* in the horizontal and vertical code versions of the first two examples are shown in Table 5.5. Sequential results, without *futures*, are shown for comparison.

Example	Version	Futures	Total Ops	% Parallelism	Real Time Ops
1	vertical	no	9189	100	9189
1	vertical	yes	9369	101	9270
1	horizontal	no	8259	100	8259
1	horizontal	yes	8687	152	5717
2	vertical	no	25424	100	25424
2	vertical	yes	25484	104	24454
2	horizontal	no	24476	100	24476
2	horizontal	yes	24776	158	15724

Table 5.5
XML Results — Examples 1 and 2
Hand Placement of *Futures*

Each *future* adds to the "total ops" column via its overhead and to the "% parallelism" column if it facilitates parallel branching in the program flow. If the *future* is useful, these two effects combine to yield a lowered "real time ops" value. In all but one of the versions above, this is indeed the case. The exception, where the code doesn't contain enough exploitable parallelism to make *futures* worth trying, is the vertical coding of the small example. In example 2, code the same structure doesn't exhibit this effect. Since the vertical procedures for these two examples differ only in length — the instructions in the stream are almost identical because the circuits are composed of similar gates — the lack of parallelism in example 1 must be a result of its small size. The data dependency graph is not complex enough to make relaxing its constraints an advantage; there aren't enough tasks to keep the "in box" or the pipelines full, to extend the comparison to the two models which were introduced in section 5.2. Example 2's size allows it to escape this problem. The horizontalized versions are saved by the granularity of their available parallelism, which was precisely the intent of the condensation algorithm.

The "judiciousness" of hand-placement hinges upon the programmer's knowledge. In the process of finding the best combination, several empirical heuristics were discovered and followed. The most important one was later formalized as *"quickness"* — that *futures* enclosing simple tasks are inefficient. The interactions between *futures* can be perplexing. Two *futures* may allow 124% and 119% individually but 151% together. Synergy is obviously at work, both in positive and negative directions: two otherwise useful *futures* can work badly in combination. These interactions are too intricate to be summarized in a limited set of general heuristics. See [14] or [11] for further elaboration.

The optimistic *future* placement algorithm is at the opposite end of the spectrum. It exercises no judiciousness whatsoever about its choices, adding as many *futures* as possible. The results of applying this algorithm to the code for examples 1 and 2 are in Table 5.6. Although the plethora of *futures* certainly keeps the task pipelines full, those tasks are too small to be efficient in the vertical versions. Note that the "total ops" column is greatly increased over that of the sequential code — much more so than was the case when *futures* were placed, more sparingly, by hand. The amount won back by the parallelism is sufficient, in all cases but the first, to at least counterbalance the added overhead. The size issues that caused the similar effect with *futures* hand-placed in the code are at the root of this. A comparison of the "real time ops" columns of hand and optimistic versions shows that the latter loses in all cases except in the vertical coding of example 2, where the difference is too small to be significant.

Example	Version	Futures	Total Ops	% Parallelism	Real Time Ops
1	vert	no	9189	100	9189
1	vert	yes	10473	101	10417
1	horiz	no	8259	100	8259
1	horiz	yes	8829	138	6377
2	vert	no	25424	100	25424
2	vert	yes	27164	112	24338
2	horiz	no	24474	100	24476
2	horiz	yes	25496	156	16379

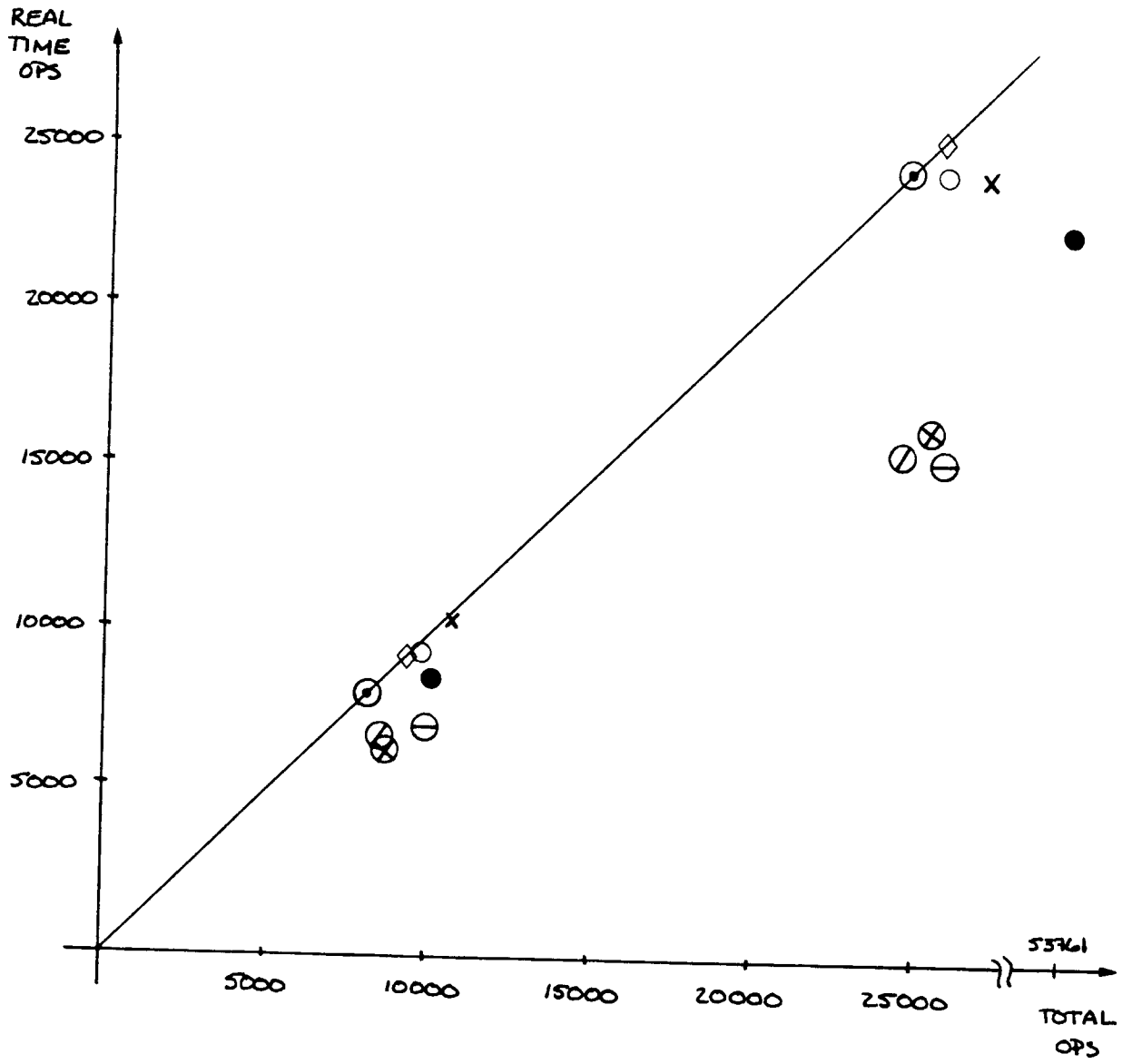
Table 5.6
XML Results — Examples 1 and 2
Optimistic Placement of *Futures*

Savant, exercising its objective mathematical judgment, succeeds best of all. Data are shown in Table 5.7. Speedup factors occur where expected, and in *every* case the "real time ops" value for the *Savant*-compiled code is lower than the original. The data from the other placement heuristics both showed exceptions to this trend, where the overhead killed the performance gain. The speedup factors are larger in the second example because of the circuit's size, which keeps the pipelines full of tasks and allows the condensation algorithm to perform more efficiently. The vertical code appears to have more potential (2.36 best-case speedup versus 1.68 best-case for the horizontal code.) As will be demonstrated later, this is not true for the XML data from the larger examples, nor for actual Concert data for *any* example. It is assumed to be related both to size and to XML approximations.

Example	Version	Futures	Total Ops	% Parallelism	Real Time Ops
1	vert	no	9189	100	9189
1	vert	yes	9772	114	8571
1	horiz	no	8259	100	8259
1	horiz	yes	8740	125	6992
2	vert	no	25424	100	25424
2	vert	yes	53761	236	22780
2	horiz	no	24474	100	24476
2	horiz	yes	25813	168	15365

Table 5.7
XML Results — Examples 1 and 2
Savant's Placement of *Futures*

The performances of the various *future* placement algorithms are shown graphically in Figure 5.1. The vertical axis shows "real time ops" and the horizontal axis shows "total ops." The line with a slope of 1.00 represents 100% parallelism or a factor of 1.00 speedup. The overhead added by its *futures* pushes the data point for a *future*-doctored procedure to the right and any parallelism gains push the point down. A useful *future* pushes harder *down* than *right* so that the overall trend is down — lowered overall real-live simulation time. Eight points are plotted for each of the two examples: four placement algorithms acting upon two code structures. The upper cluster of points is from example 2 and the lower cluster from example 1. The different placement algorithms and coding schemes are distinguished by the shape of the symbol used to plot the data point, as shown in the Figure's key.



◇ = none	⊙ = none
○ = hand	⊘ = hand
× = optimistic	⊗ = optimistic
• = *Savant*	⊖ = *Savant*
Vertical	Horizontal

Figure 5.1
XML Results — Examples 1 and 2
Comparison of All *Future* Placement Methods

With no *futures*, there is no parallelism and thus the "no *future*" ($\diamond$ and $\odot$) data points all lie on the 1.00 line. The other points all fall somewhere below it. The best of all the points, given the intent to improve overall time, is the lowest one. In all cases but one, the lower cluster of red points, this point is *Savant's* $\bullet$ or $\ominus$. The reason for the exception was discussed at the end of section 4.4 — *Savant* is overcautious about procedure inputs, treating them as if they were all undetermined *futures* and thus wrapping all references to them in another layer of *futures*. The extra overhead kills performance in cases where the program contains so few internal sites that the input-wrapping *futures* form the majority. This caution is vindicated when cycles are called in parallel and inputs are indeed undetermined, in which case the input-wrapping *futures* greatly enhance the parallelism.

Since *Savant's* results are consistently the best and the only exception arises from a mechanism which will actually be beneficial when cycles are called in parallel, hand and optimistic placement were not pursued any farther. This choice was based solely on the early XML data, so the next step in the experiments was to verify the results on Concert.

For each example, four versions of the single-cycle procedure exist, comprising all possible combinations of horizontal/vertical code with or without *Savant's futures*. The results of running these four versions on Concert, both in cycle-serial and in cycle-parallel mode, are shown in Table 5.8.

Garbage collections clutter the data for example 4, obscuring the actual results. Since allocation activity increases with program size and *future* usage and example 4 is by far the largest circuit this is unavoidable. Garbage collection frequency is a function of the amount of system free storage and the time lost during the cleanup is a function of the algorithm used. Each computer system has a unique combination of these two parameters. Incorporating Concert's personal combination as a part of these results would make them not only less general, but also more difficult to interpret. Neither function — the time used by an algorithm or the necessary collection frequency at a particular amount of free storage — is simple or even well-defined, and thus "backing out" the unadulterated time value is impossible. Example 4 is only presented to illustrate CONSIM's behavior within a class of problems relevant in the real world, and this garbage-collection-produced interference is thus a realistic problem. However, it complicates drawing conclusions about trends in experimental results. These trends must be identified if the process is later to circumvent them. In the experimental world, data which masks the trends under study are worse than useless. Because it is the largest example not crippled by garbage collection,

Example	Version	Time (sec) Cycle-Serial	Time (sec) Cycle-Parallel	Improvement Ratio Serial ÷ Parallel
1	vert	3.000	1.908	1.57
1	vert/fut	3.200	1.692	1.89
1	horiz	3.067	1.984	1.55
1	horiz/fut	2.167	1.750	1.24
2	vert	7.400	6.317	1.17
2	vert/fut	4.134	3.234	1.28
2	horiz	8.533	7.725	1.10
2	horiz/fut	3.959	3.067	1.29
3	vert	16.717	16.259	1.03
3	vert/fut	12.342	4.025	3.07
3	horiz	16.075	15.509	1.04
3	horiz/fut	7.217	3.783	1.91
4	vert	78.867	85.600*	0.92
4	vert/fut	24.775*	23.050*	1.07
4	horiz	74.600	83.559*	0.89
4	horiz/fut	23.792*	24.942*	0.95

Table 5.8
Concert Results — All Examples
14 Processors; * = garbage collection
Cycle-Serial vs. Cycle-Parallel Code

example 3 is considered the single "most representative" example of the four.

The promised effects of cycle-parallel invocation make a striking appearance. In *all* cases, except where obscured by garbage collection (and probably there too, underneath the noise — recall that the extra *futures* of cycle-parallel execution actually cause more garbage collection), cycle-parallel code runs faster. This holds regardless of size, regardless of code structure, and whether the single-cycle procedure is purely sequential or contains internal *futures*. The magnitude of the speedup ratio *is*, however, a function of the first and last of these three parameters. With one exception (the horizontal coding of example 1), the speedup ratio is larger for single-cycle procedures with internal parallelism, suggesting that "macroparallelism" multiplies the effects of "microparallelism." The mechanism at work in this case is that the cycle-wrapping *futures* prevent the single-cycle procedures from

immediately hanging up because mstate is a *future*, as discussed in a following paragraph and Figures 5.2 and 5.3. The improvements are greater, in cycle-parallel mode, in example 3 than in example 2, which again suggests that size plays a role in keeping pipelines full. There is no identifiable connection between code structure and improvement ratio, but since the times for the horizontal code start out lower than those for the vertical code, the same factor of improvement yields a lower final value.[7]

The "best combination" of code structure and *future* placement — the consistently lowest of any example's cluster of eight time values — is the horizontal code, with *Savant's futures*, run in cycle-parallel mode.

The synergy between microparallelism and macroparallelism is illustrated in Figures 5.2 and 5.3. The main execution sequence of the program is shown by the line of squares linked by double arrows. Parallel tasks, symbolized by circles, are spawned where indicated by wavy arrows. Single arrows represent precedences imposed by data dependencies within the program. Figure 5.2 shows cycles without internal parallelism, called in parallel. Here the circles — the tasks — represent entire cycles. Since mstate of cycle n, needed at the very *beginning* of that cycle's computation, is nstate of cycle $n-1$, assembled at the very *end* of that cycle's computation, the data dependencies all but sequentialize those tasks. Figure 5.3 shows cycles with internal parallelism, called in parallel. The internal parallelism causes each cycle to spawn pieces of itself off as separate tasks. Now the data dependency arrows aren't constrained to lead from the end of one cycle to the beginning of the next, but from the point of production to the point of need of the datum in question — which may well be from midcycle to midcycle, as shown. At least *some* work can proceed before hangup, which keeps processors busy and useful. The depths within the cycle at which those arrows enter and exit, which ultimately govern how much parallelism can be squeezed out of the program, depends upon the application.

In these cases, the amount of exploitable parallelism is primarily a function of the code structure and *future* placement, rather than of the circuit topology. Creating efficient *future* sites dominates all other considerations and requires the code to depart from the topology given in the HDL. (The topological analog of code

[7] The data from examples 1 and 4 are given less weight than the others in most of these comparisons, since example 1 has previously demonstrated odd size effects and the latter are obscured by garbage collection.

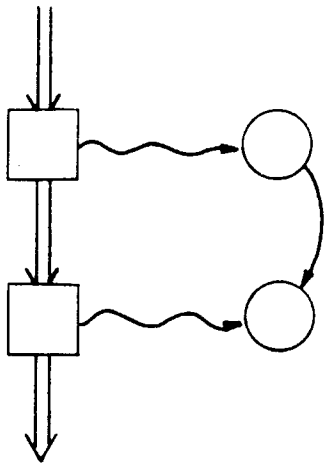


Figure 5.2
Cycle-Parallel Invocation of Internally Serial Code
Program Flow and Precedences

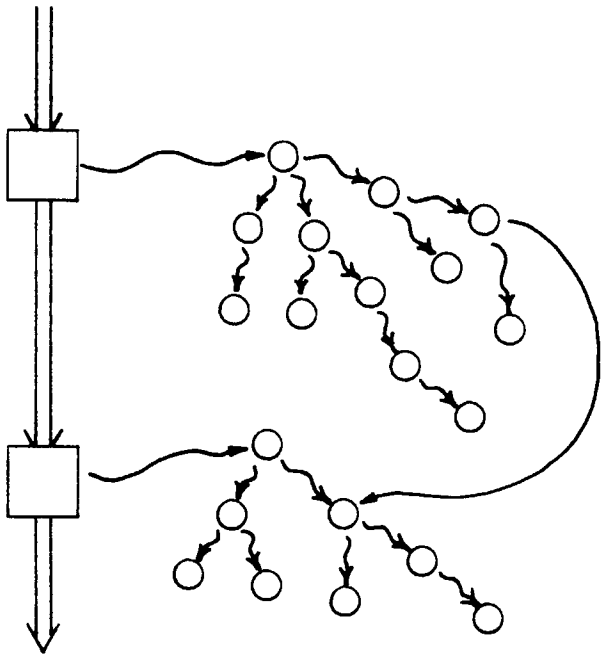


Figure 5.3
Cycle-Parallel Invocation of Internally Parallel Code
Program Flow and Precedences

condensation is lumping circuit blocks together.) Passing values between blocks (simulating the wiring) is clearly not "free" — it actually dominates the behavior. The simple circuit model is partially responsible for this. ANDs and ORs are simple things compared to scheduling of processes! If the blocks contained complex models of huge linear networks, the balance would be different and hardware topology might play a bigger role in simulation speed. Topology issues are discussed, for extreme cases, later in this section. As it is, software issues dominate CONSIM's performance.

As the run length and thus the number of parallel tasks increase, the parallelism also increases at first, then reaches an asymptotic value. The XML "% parallelism" statistic shows this more clearly than the Concert time. Table 5.9 illustrates this effect. These results are from a slightly larger version of example 2. The total number of operations increases linearly (to within 1%) with the number of cycles simulated. The parallelism increases, as the number of tasks grows to the capacity of the pipelines, from 100% to some maximum determined by the code. Further increases simply raise the backlog — the level of liquid in the reservoir — and not the parallelism. It is at or above this capacity that Concert's efficiency is maximal and CONSIM's results indicate its true power. Note that for cycle-serial code, this level is already reached in a 5 cycle run and no increase in the fourth column is seen for longer runs. For cycle-parallel code, where different granularities of parallelism are at work, this number doesn't level off until around 10 cycles. A simulation of a smaller circuit requires more cycles to reach this point. In view of this, a uniform 20 cycle run length was chosen for all tests in this paper. The scenario of full pipelines, with orderly lines of homogeneous tasks waiting in line at their entrances, represents a steady-state that dominates long-term behavior. Since long simulations of large circuits are the rule rather than the exception in simulator applications, simulations that fit this scenario reflect real-world use.

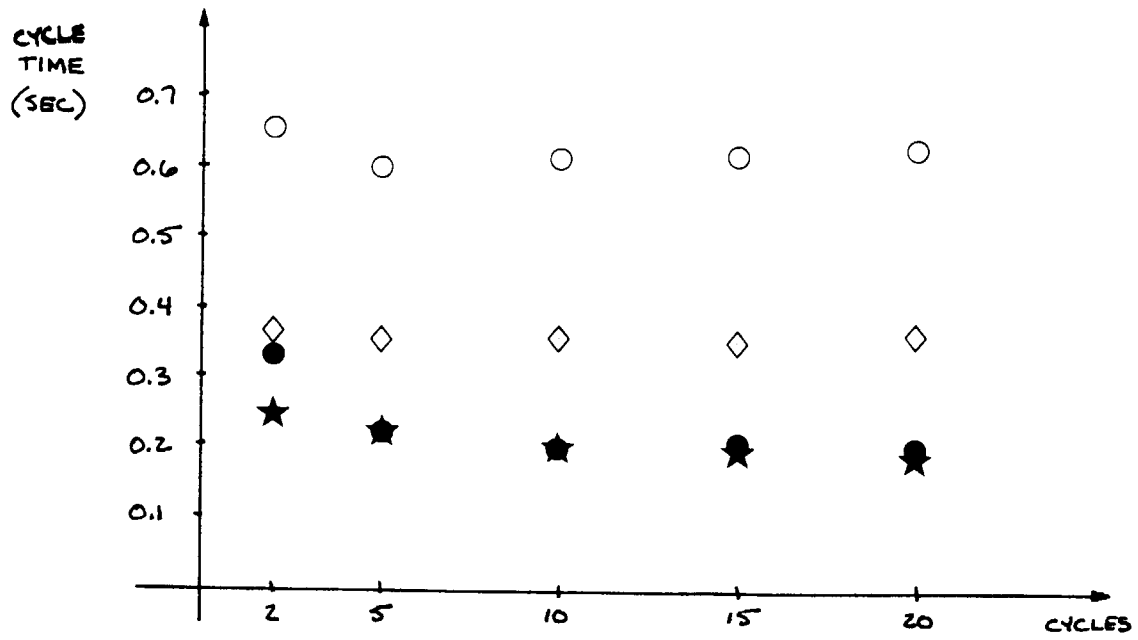
These effects can be indirectly observed on Concert by keeping track of the time per cycle for various length runs. The graphs in Figure 5.4 show these data for all parallel versions of example 3. This example is large enough to require only a few cycles to fill the pipelines. Following a slight initial downslope, the cycle time levels off, indicating that steady-state has been reached. This corresponds to the point in the XML data above which the maximum parallelism is attained. An identical experiment on an 8 processor Concert system did not show this initial glitch; since there were fewer pipelines, fewer tasks were required to fill them up. A smaller circuit would have the opposite effect: the glitch would be more pronounced and more cycles would be required to reach steady-state.

Version	Cycles	Total Ops	% Parallelism	Real Time Ops
vert	5	8176	100	8176
vert	20	25424	100	25424
vert/fut	5	17214	240	7160
vert/fut	20	53761	236	22780
horiz	5	9346	100	9346
horiz	20	24474	100	24474
horiz/fut	5	9856	170	5782
horiz/fut	20	25813	168	15365
		Cycle-Serial Code		
vert	5	8320	137	6012
vert	20	32819	146	22506
vert/fut	5	17268	844	2046
vert/fut	20	69138	1115	6199
horiz	5	9400	132	7145
horiz	20	37499	137	27316
horiz/fut	5	9910	504	1967
horiz/fut	20	39539	612	6461
		Cycle-Parallel Code		

Table 5.9
XML Results — Example 2(a)
Effects of Run Length on Parallelism

Note from Table 5.10 that simulation time grows with the number of gates in the circuit, but that the time per gate remains about the same, barring the interference of garbage collection, for both horizontal and vertical codings when no *futures* are present. This is consistent with the condensation algorithm, where the code is simply reorganized so as to create expressions that are more complex. Adding *futures*, either around or within the cycles, removes this correlation and the time per gate drops drastically, since many gates are being simulated at the same time.

Circuit size is a double-edged sword. Small circuits do not usually contain enough tasks to fill the parallel pipelines, and when they do, the tasks are too *quick*



○ = vertical/cycle-serial
● = vertical/cycle-parallel
◇ = horizontal/cycle-serial
★ = horizontal/cycle-parallel

Figure 5.4
Run Time Per Cycle For Various Run Lengths on Concert

to aid performance. The condensation algorithm, a possible solution to this, works poorly on small programs. A toy circuit is not a "useful" simulator application; optimizing a simulator for such a circuit would be naive and artificial. Simulators come into play when people are faced with circuits that are too big or too complex for them to analyze by hand. CAD tools should be optimized for the applications in which they will be used, and not for some academic example which happens to give intriguing results (although exploring those results may give insight into the simulator's behavior and perhaps even suggest something about the type of circuit it should be used on.) XML data from the longer simulations of larger circuits are more realistic for statistical reasons, since the histogram facility uses methods that require large sample spaces to produce valid results. On the other hand, large

Example	Number of Gates	Time per Gate per Cycle (sec) Vertical No *Futures*	Vertical *Futures*	Horizontal No *Futures*	Horizontal *Futures*
1	10	0.015	0.016	0.015	0.011
2	25	0.015	0.008	0.017	0.008
3	58	0.014	0.011	0.014	0.006
4	232	0.017	0.005*	0.016*	0.005*
		Cycle-Serial Code			
1	10	0.010	0.008	0.010	0.009
2	25	0.013	0.006	0.015	0.006
3	58	0.014	0.003	0.013	0.003
4	232	0.018*	0.005*	0.018*	0.005*
		Cycle-Parallel Code			

Table 5.10
Circuit Size and Concert Execution Time
14 Processors; * = garbage collection

circuits use more storage space and need more garbage collection, which obscures true results, but reflects a real-world problem. Experiments must steer a careful path between these two extremes to avoid misrepresented data.

If t tasks, each requiring execution time Δt, were running on n processors, and each task were assigned to the first vacant processor and ran to completion with no overhead, execution time as a function of number of tasks can be predicted as follows. For $1 \leq t < n$, there are idle processors. The runtime is Δt and the speedup is t. For $n < t \leq 2n$, the runtime is $2\Delta t$ and the speedup is $t/2$. In general, the runtime is $\lceil t/n \rceil \Delta t$ and the speedup is $t/\lceil t/n \rceil$. This relation is plotted in Figure 5.5.

For CONSIM, the worst of the approximations in this hypothesis is that all tasks have the same execution time. Not only are the sizes of the circuit blocks different, implying differences in the granularity of the potential "microparallelism," but *entire cycles* can also be invoked as tasks. The *future* placement algorithm adds another variable, because it decides which of these available tasks to exploit. The approximation that all tasks run to completion is more applicable to *pcalls* than to *futures*, since processes which require an explicit argument hang up when they encounter an undetermined *future*. The real Concert data will thus comprise

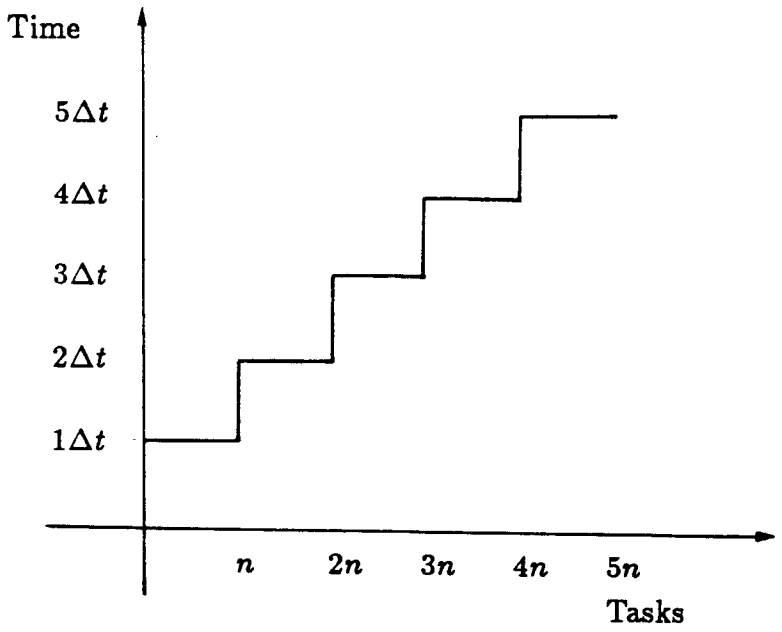


Figure 5.5
Execution Time vs. Number of Tasks
Idealized Execution Model

some complex statistical distribution of tasks, governed by data dependencies which further complicate how these tasks run.

Some information about the statistical distribution of the tasks created by the various code structures and *future* placements is needed to determine how closely CONSIM fits this model. It is unfortunate that the detailed breakdown provided by the XML histogram is invalid. XML reports the speedup factor that would result on a system with one to 50 processors. Given *how* the speedup grows with number of processors, one can determine the nature of the available parallelism. The speedup might grow linearly as processors are added, or it might asymptotically approach a limiting value after only a few processors are included in XML's calculations. The latter, which is apparently the case with CONSIM, indicates that the application program does not contain enough parallelism to keep more than a few processors busy. This effect is discussed later in this section in conjunction with the discussion of Figure 5.7. This breakdown of information is not available *explicitly* from any other measurement, but can be gleaned indirectly from simulation runs on different numbers of processors.

Two user-level parameters affect the number of tasks: circuit size and simulation run length. Four points (from four examples) are not enough to predict

a trend, so the easily-altered run length was chosen as the variable parameter in investigations of time versus number of tasks. Number of tasks and run length are linearly related. The relationship between circuit size and number of tasks is not so clean, since data dependencies also affect run time. Two n-gate circuits with different topologies may have vastly different execution times if one's topology creates data dependencies which cause many processes to hang up (e.g., a ring oscillator, where dependencies actually force sequentialization.) Execution times of all parallel versions of example 3 for various run lengths are graphed in Figure 5.6.

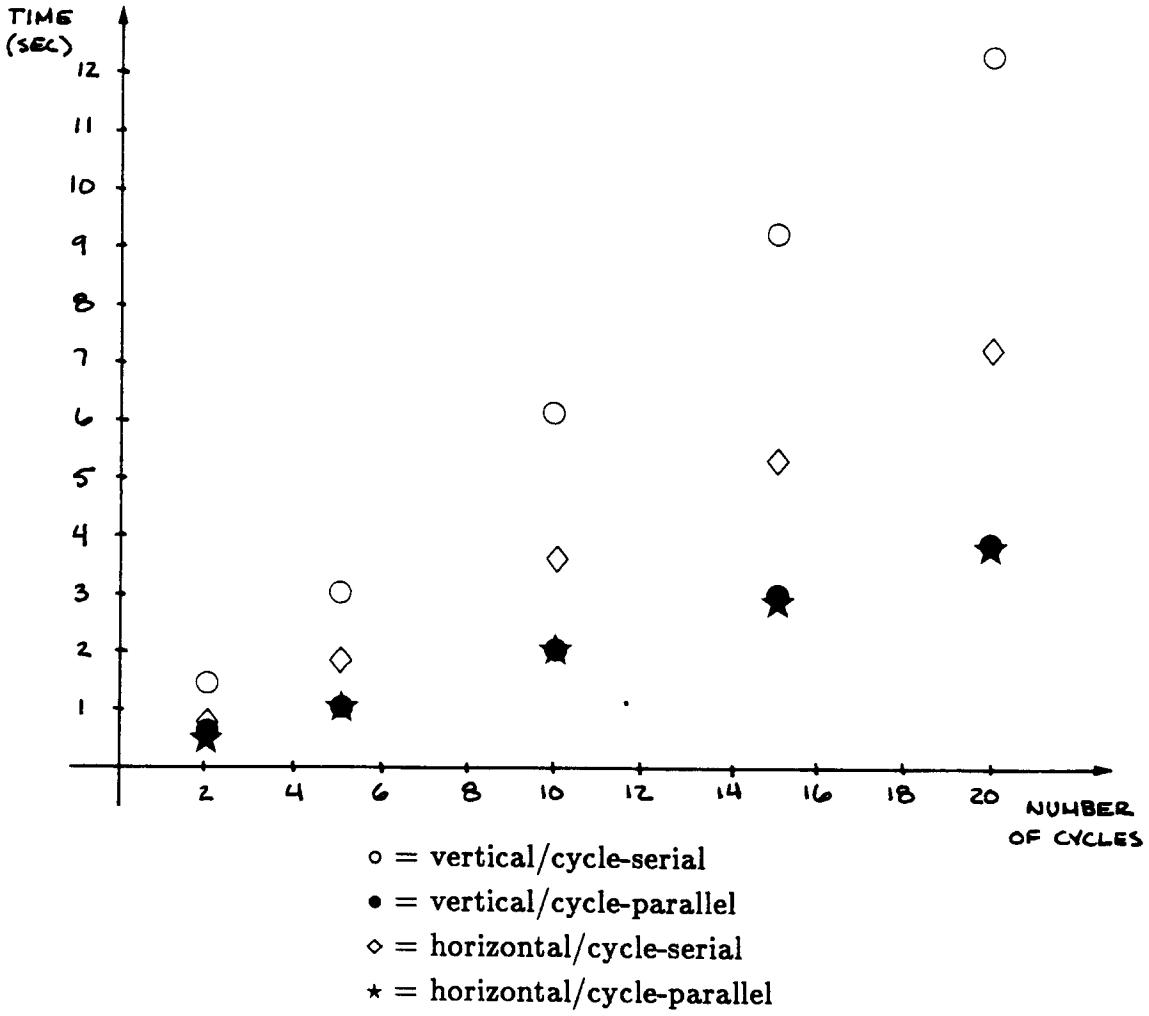


Figure 5.6
Overall Run Time For Various Run Lengths on Concert

The upward trend is obvious, but the "staircase" effect of Figure 5.5 is absent. The "frequency" of the staircase wave is $\frac{1}{n}$, so the number of tasks would have

to be increased in increments of $\frac{n}{2}$ ("sampling" at twice the highest "frequency"), according to the Nyquist criterion, for its fine-grain nature to be apparent. A one-cycle increase in run length, for this example, comprises many more than 14 (n) tasks. Overall, the "mean" behavior seems to follow the global pattern predicted by the model. Even if there were a way to gather more detailed data, it would be unlikely that the required complex statistical analysis would yield a closed-form solution or provide any insight into the causes and predictions of the simulator's performance.

Concert execution time data on 1, 2, 4, 8, and 14 processors for the various versions of each example appear in appendix 4. The data for examples 2 and 3 are graphed on log-log scales in Figure 5.7. The most striking trend in these graphs is the sharp downward slope of the internally-parallel versions' data lines. When only one processor is active, the code is forcibly sequentialized and the parallel versions lose by virtue of their uncancelled overhead. The overhead added by the *futures* appears as the vertical spacing between the one-processor data points. This spacing is greater for the vertical versions in example 2 independent of macroparallelism. It seems to be roughly the same in example 3. This difference is probably an artifact of program overhead, such as counting cycles, returning the list, etc., which is a larger percentage of the execution time for a smaller circuit, such as example 2. As processors are added, the overhead is amortized and the parallelism is exploited. Eventually the parallelism gains overtake and pass the overhead losses and the lines cross. Beyond the crossing point, the lines for the procedures with internal parallelism continue downwards more or less in parallel and then the horizontal code line seems to tail off, followed by the vertical line. This effect is most apparent in example 2; it appears to some extent in example 3, but is absent in the cycle-parallel version. The number of tasks is responsible for this behavior. Where adding a pipe reduces the flow in the other pipes, improvement continues, but not at the same efficiency per added processor: a "diminishing returns" effect. Presumably some upper limit is reached, where adding a processor not only gives no improvement, but may actually cause a degradation in performance because of bus contention. The number of available tasks mitigates this effect — if the reservoir is really chock-full, adding a pipe won't decrease the flow in the other pipes, and the new pipe will work at full capacity. Parallel cycles and size both increase the number of available parallel tasks, and the graphs reflect it. The macroparallel versions level off later, as do the lines for the larger example. Example 3 shows the breakpoint clearly: the cycle-serial lines level off and the cycle-parallel lines don't. The latter case obviously

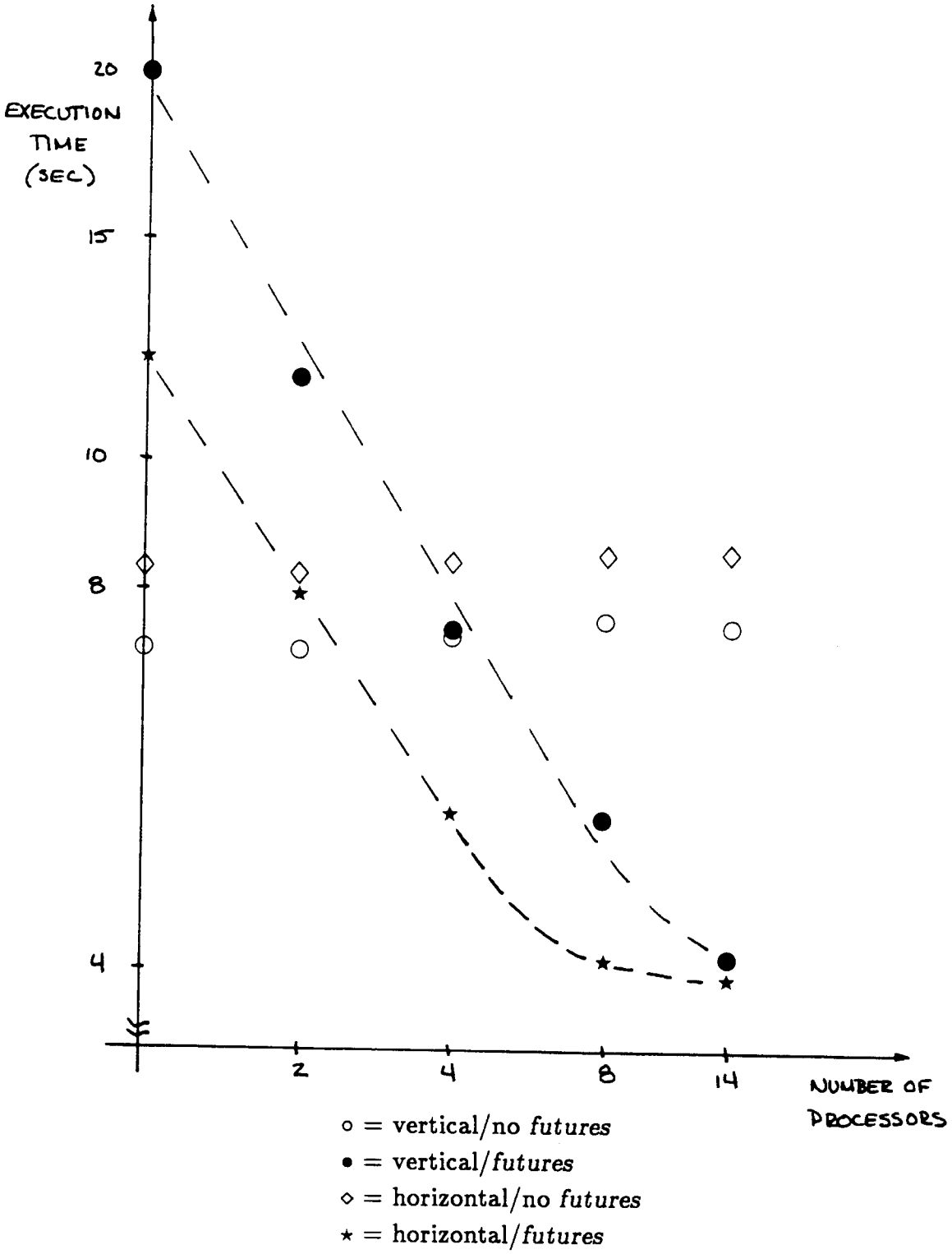


Figure 5.7 (a)
Concert Run Time vs. Number of Processors
Example 2 — Cycle-Serial Invocation

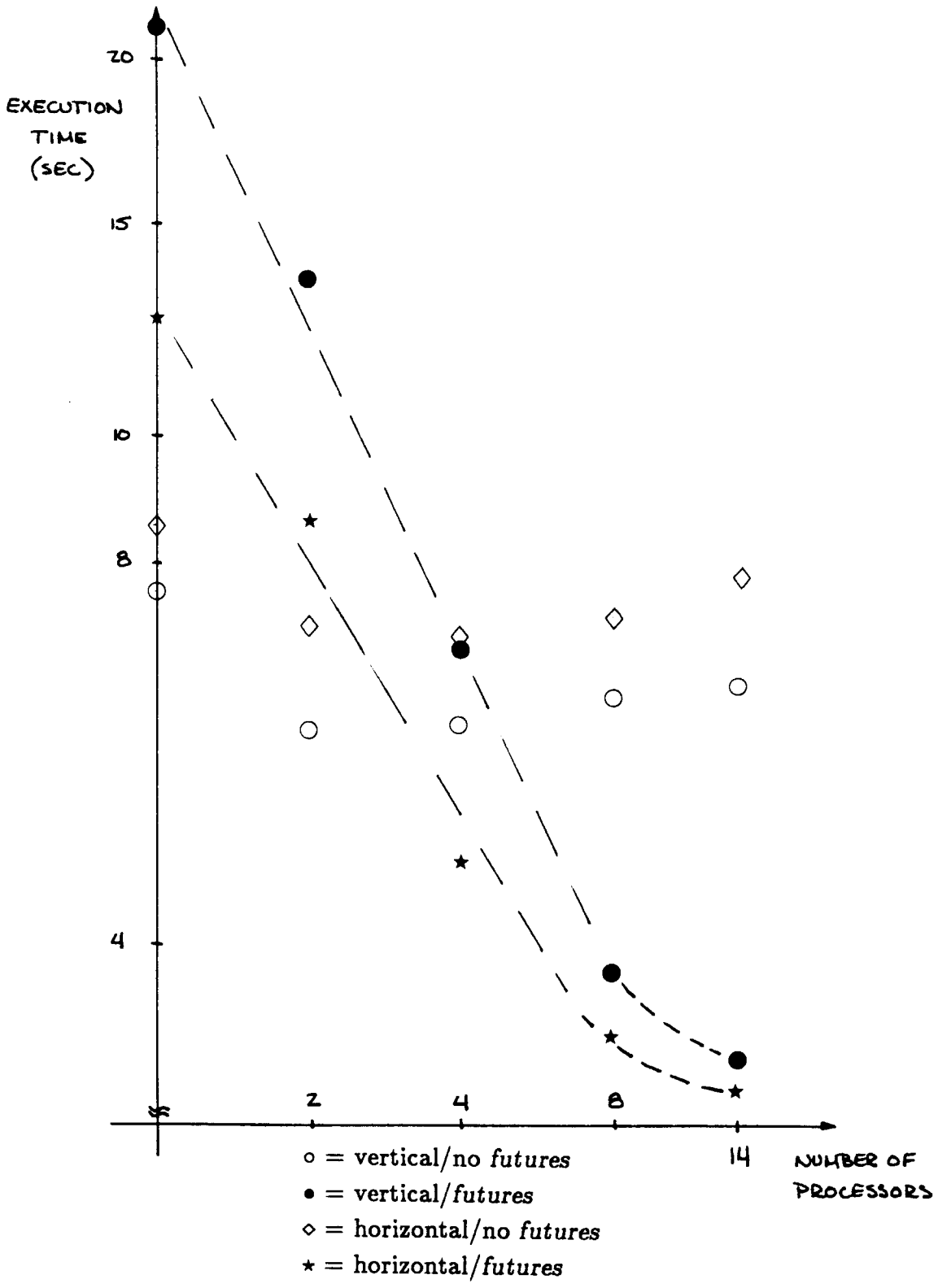


Figure 5.7 (b)
Concert Run Time vs. Number of Processors
Example 2 — Cycle-Parallel Invocation

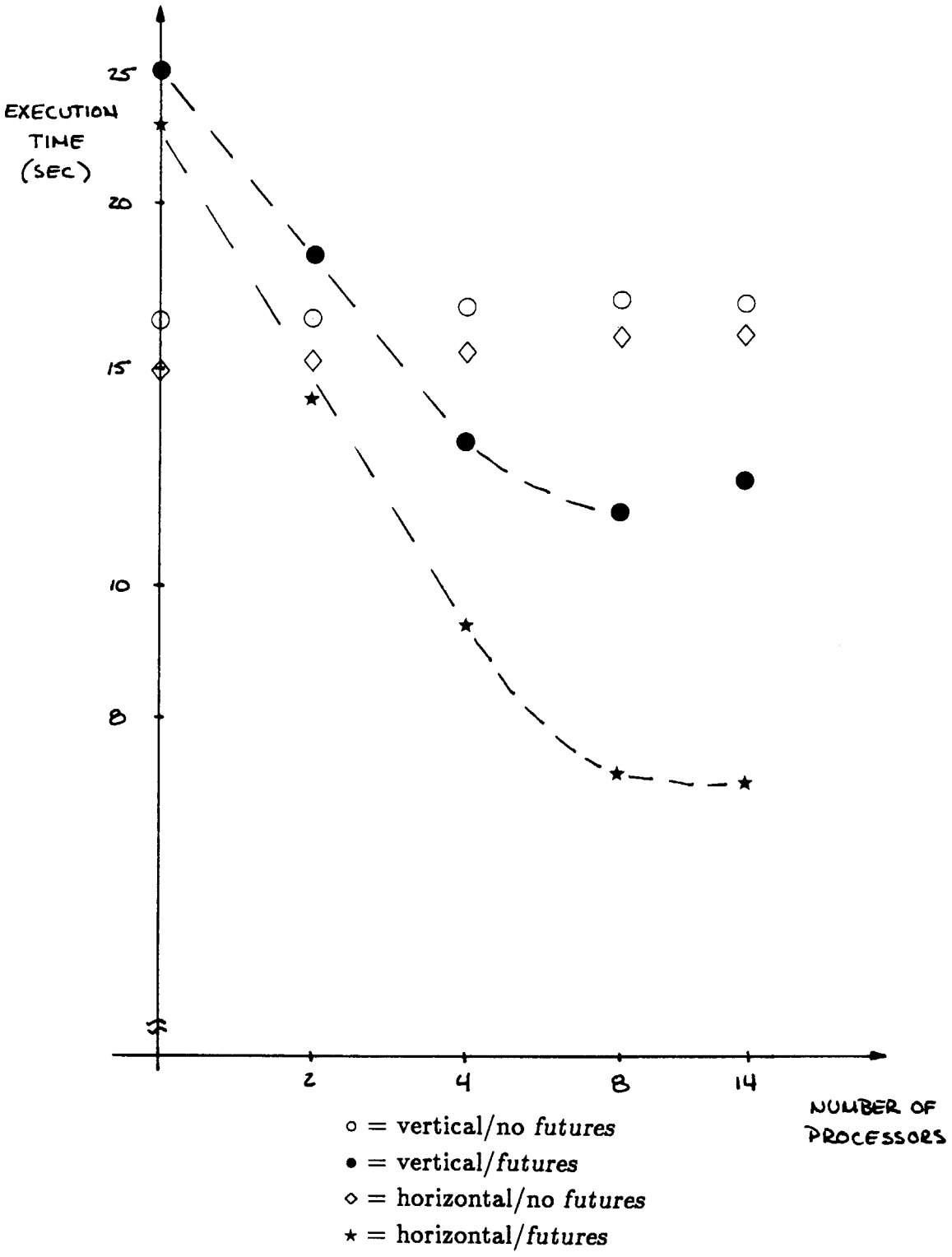


Figure 5.7 (c)
Concert Run Time vs. Number of Processors
Example 3 — Cycle-Serial Invocation

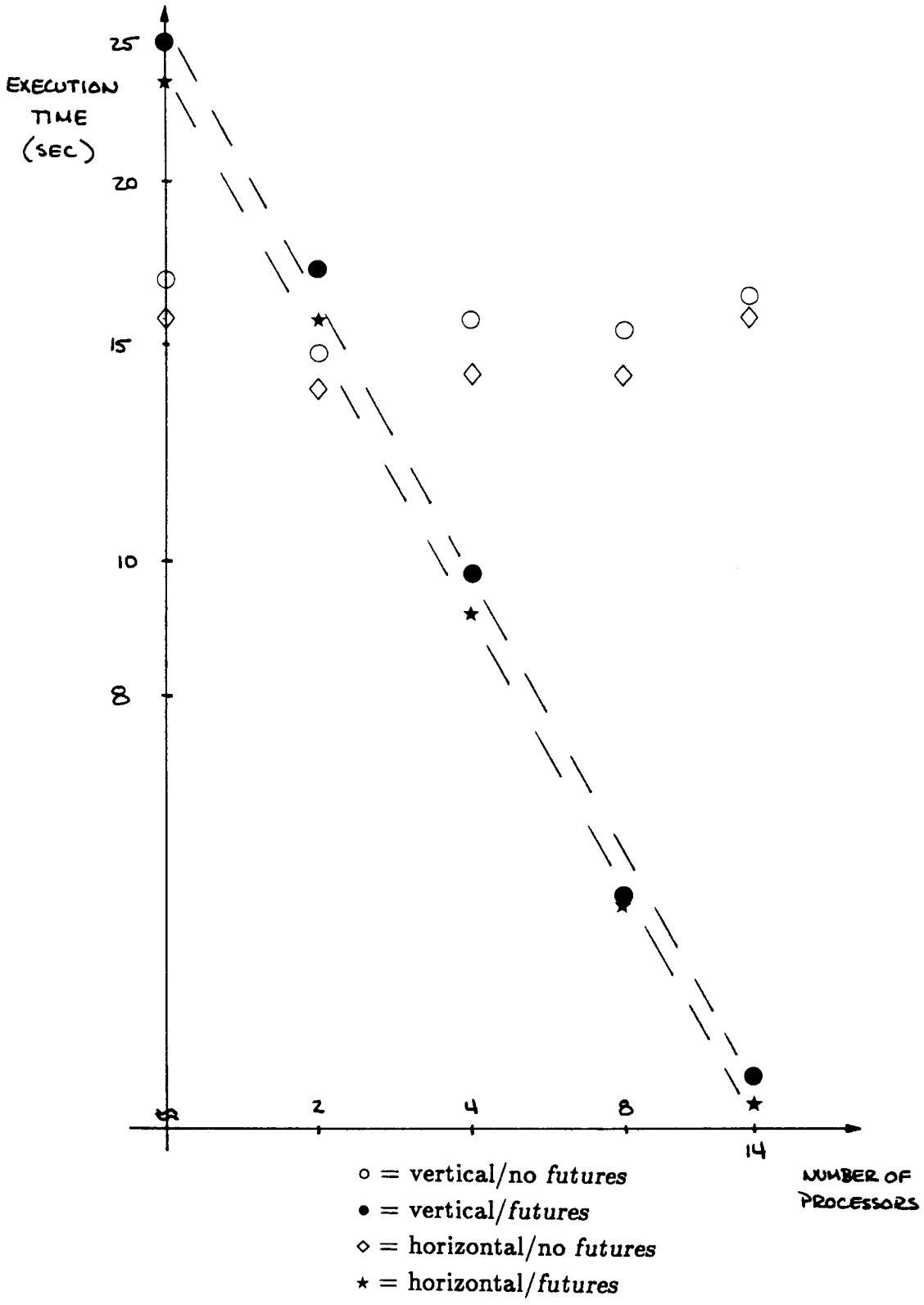


Figure 5.7 (d)
Concert Run Time vs. Number of Processors
Example 3 — Cycle-Parallel Invocation

fits the proposed steady-state scenario for long-term simulation.

Since the axes of these graphs are scaled logarithmically, a line of constant slope is a *power* function: the function $y = 1/x^n$ would be the line of slope $-n$. On an ideal concurrent computation system, where any program would run n times as fast on n processors as on one, all data line slopes would be -1. The actual CONSIM slopes are shown in Table 5.11.

Example	Cycle-Parallel		Cycle-Serial	
	Vertical	Horizontal	Vertical	Horizontal
2	0.82	0.80	0.74	0.61
3	0.74	0.74	0.44	0.67

Table 5.11
Asymptotic Line Slopes — Versions With *Futures*
Improvement Per Processor
Concert Results

A slope of -0.8 means that the program runs $n^{0.8}$ times faster on n processors $(1/n^{0.8})$. The lines on the graphs only appear to be straight for low numbers of processors, as explained above. The slopes in this table were measured in the linear sections of each line, which comprise a greater and greater portion of the entire line as parallelism increases. Cycle-parallel invocation may create larger line slopes than cycle-serial invocation because it creates more tasks, or because it relaxes more of the artificial precedence constraints to which cycle-serial invocation adheres, thus causing a wider average parallelism profile. Horizontal and vertical code structures generally result in roughly similar slopes. The reason that the data apparently follow a power-law speedup other than n^{-1} is not obvious. It may be a result of the average amount of available parallelism, which limits processor activity. It also may be illusory; five data points do not comprise an exhaustive proof that a line is straight.

The lines for the versions with no *futures* inside the single-cycle procedures don't slope down. Any "parallelism" these programs contain is exploited by the first processor. They don't take advantage of Concert's power at all, regardless of whether or not cycles are called in parallel. [8] In fact, all lines for these versions

[8] The synergy between internal parallelism and cycle-parallel invocation was discussed earlier in this section — see Figures 5.2 and 5.3.

show a slight *rise* in runtime as the number of processors increases. This trend may be simply an artifact of measurement approximation, since its magnitude is not really far enough out of the error band to even prove its existence beyond a doubt, although the same effect *has* recently been observed when sorting routines are executed on Concert [16]. If it *does* exist, it may be due to bus contention and other side effects incurred because of the the idle processors' searches for tasks to perform. Horizontal versions with no internal *futures* execute more slowly than similar vertical versions in example 2; this is *reversed* in example 3 and is due to the condensation algorithm's increased efficiency upon larger programs.

The behavior of the tailoff of the microparallel data lines is encouraging. It suggests that execution time will probably continue to improve as processors are added to the system beyond the number actually available in these experiments. As circuit size increases, the tailoff points move further to the right (on the axes for the graphs in Figure 5.7). The improvement in run time gained by adding another processor to an n-processor system is given by the slope of the tangent to the curve where the number of processors is n. It is more efficient to add processors in the linear region, since the slope is higher before tailoff. A longer linear region means that a larger number of processors can be used efficiently. Extrapolating the lines for example 3 to the 50 processor level is unwise — the tailoff point probably lies just off the graph and the extrapolated run time value would be far too optimistic. Farther-reaching extrapolations could be made for larger circuits, such as example 4, were it not for garbage collection.

Garbage collection is a function of the amount of free storage: the more free storage a system contains, the less often program activity will fill it up and the less often it will require cleanup. At the time of these experiments, the fixed amount of Concert memory was divided up among the active processors. When only 8 processors are active, each processor owns $\frac{14}{8}$ times as much memory as does each member of a 14 node system, and thus garbage collection is less frequent. Where memory is limited, this has severe implications for CONSIM's performance on larger numbers of processors. If the garbage collection time negates the $1/n^{0.8}$ gain, CONSIM will actually run faster on the smaller multiprocessor system than on the larger one.

Garbage collection pads execution time and drives the lines in Figures 5.4, 5.6 and 5.7 upwards. In the long term steady state scenario, this padding is more or less constant, since all pipelines are always working (and creating garbage) at full capacity. Increasing the simulation time with larger circuits or longer runs will

not alter this percentage, although it *will* increase the total time used. Since the experiments described in this paper attain the steady state only in fits and starts, garbage collection is a nuisance because it randomly scatters otherwise-useful data points. However, it must be accounted for in extrapolations to larger circuits.

The immediate return of a token for a *future*'s value, although vital in other parts of CONSIM, presents a problem in the return of the final value. The immediate return of an undetermined *future* can hide a huge percentage (upwards of 90% in some cases) of the computation work for that future, so the "execution time", measured up to the actual return, is meaningless. The solution is to touch every *future* at the end of the CONSIM run. Since *futures* can appear unpredictably in any part of the output list, the entire structure must be touched at all levels by a tree walk. This tree walk adds a constant amount of overhead for every version of an example (since all flavors of code structure and *future* placement had better return the same logical results!) Table 5.12 shows full simulation times for sequential versions of example 3 with and without the tree walk. The time taken by tree walks of fully-determined result lists for all examples are shown in Table 5.13. The differences between columns in Table 5.12 correlate well with the times in Table 5.13. These times remain constant, for a given length run on a fixed number of processors, for all code structure/*future* placement versions.

Version	5 Cycles		20 Cycles	
	Walk	No Walk	Walk	No Walk
vertical	4.217	3.800	17.050	14.950
horizontal	3.917	3.392	15.500	13.683

Table 5.12
Concert Results — Example 3
20 Cycle Simulation With and Without Tree Walk

Example	5 Cycles	20 Cycles
1	0.250	0.992
2	0.333	1.267
3	0.516	2.050
4	1.492	6.050

Table 5.13
Concert Tree Walk Times — All Examples

These times represent significant chunks (ranging from 10% to 13% in Table 5.12) of the *total* simulation times used earlier in this chapter and reported in appendix 4. This is not "experimental error" — it simply reflects operations required to get the data into a specific form. If the user of CONSIM's output needs determined values (as a human designer would), this time must indeed be included in its performance measure. However, if the output is piped into another Multilisp program which can capitalize upon undetermined *futures*, no tree walk would be required and including its time in the speed estimate is pessimistic. Performance is a function of environment in many ways: the implementation affects the relative costs of *futures* versus those of other operations, the garbage collection time varies with algorithm and amount of free storage, and the way the output is cascaded affects the amount of work that CONSIM's back end must perform.

Chapter 6

Summary, Conclusions and Future Work

This thesis documents the investigation of a new approach to circuit simulation: the use of a multiprocessor via the parallel language Multilisp. Finding the strengths and weaknesses of this *particular* language is part of a bigger goal — to determine what features of a multiprocessor system are useful in such an application and how the simulator's performance reacts to system and application parameters. This approach was also motivated by the parallelism inherent in circuits and by the Von Neumann bottleneck, which ultimately limits the speed of sequential computers. The novelty of using Multilisp in such an application lies in the way it exploits parallelism. The Multilisp *future* allows the programmer to exploit the parallelism in the code without adding artificial precedence constraints, as do sequentialization or fork-join constructs. Learning how to model and exploit the parallelism in a logic circuit under Multilisp's facilities and constraints led to the implementation of the logic simulator CONSIM.

The results of this project take on several forms. CONSIM is a functional CAD tool which can be used to verify designs. More important are the results of using CONSIM, in several modes of operation, as an *experimental* tool on various sorts of circuits. Many issues, among them circuit size, code structure and *future* placement, were investigated using repeated CONSIM runs. The execution time data from these runs exhibit coherent, understandable trends which suggest not only a model for multiprocessor behavior in this application, but also the direction along which future simulators should develop.

Modeling the behavior of Multilisp running on the Concert multiprocessor is a very complex problem. Many layers of interpretation and other automatic (and sometimes unpredictable) intervention lie between the Multilisp user and the CPU. Thus, data take on a much more empirical flavor than is usual for experiments on a man-made system, and many simplifications are required to reach any sort of agreement at all between system and model. Modeling is, however, an essential

step, or the trends and observations in this thesis will apply only to *this* particular system. If the mechanisms which underlie the trends are understood, this body of knowledge can be applied to other systems and applications as well. This is particularly important where circuit size and number of processors are involved; both of these parameters are being pushed higher and higher by advances in VLSI technology.

The most important concept in the Multilisp performance model is that all processors must be kept busy for the machine to attain its maximum efficiency. The pool of tasks waiting to be executed must remain as full as possible. Two things affect the level in this pool: the number of tasks and their data dependencies. A task does not enter the *active* pool until its essential arguments have been evaluated. [1] Some kinds of arguments are not immediately essential; evaluation of those can be deferred until absolutely necessary, raising the number of tasks that can enter the active pool. Finally, parallel invocation adds some overhead to each task, which must be counterbalanced by the performance gain for the task to be efficient in parallel.

The various parts of this model were verified in a series of experiments. The first group of the series concentrated on how the sizes and dependencies affect the performance (via the level in the pool). The second group investigated the effects upon performance of this average level of available parallelism.

Three different *future* placement algorithms were applied to horizontal and vertical versions of the code which simulates a cycle of the FSM under test. Success, measured as execution time speedup, was limited where the algorithm neglected all overhead, slightly better where overhead was taken into account, and best where data dependencies were also factored in. Horizontal code, where the potential tasks are larger, proved more fruitful.

Extending the test one step further, entire cycles were called in parallel. Not only does this technique exploit bigger tasks, but it also relaxes some artificial precedence constraints which were introduced by executing the cycles in sequence. These constraints are analogous to the sequencing imposed in hardware by the clock signal, but are unnecessary in simulation. For both reasons, cycle-parallel code ran much faster. Synergy between internal- and cycle-parallelism allows the combination to work better than the sum of its parts — again, as a result of relaxed

[1] Here Multilisp differs from some other parallel languages, where essential and nonessential arguments are not distinguished.

precedences.

This technique could be applied recursively: a task could be spawned whose only function is to spawn two more tasks, which each spawn two more, etc., until finally the cycles are called as the "leaf nodes" of a tree, as shown in Figure 6.1. The example given here is a binary tree, but the spawning process could follow tertiary or higher protocol as well.

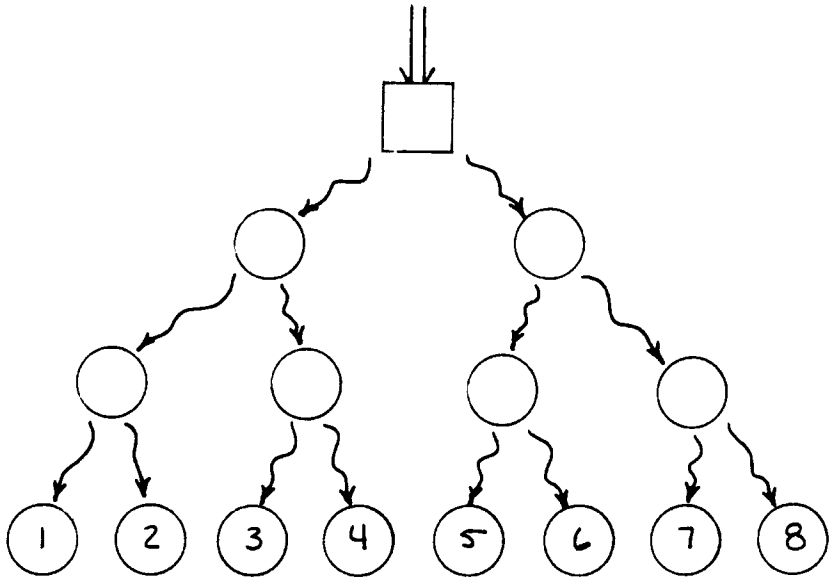


Figure 6.1
Alternate Spawning Structure

The notation in Figure 6.1 is the same as in Figures 5.2 and 5.3. The main execution sequence of the program is shown by a line of squares linked by double arrows. Here the main execution sequence consists of a single spawn. Parallel tasks, symbolized by circles, are spawned where indicated by wavy arrows. Single arrows represent precedences imposed by data dependencies within the program. The bottom row of tasks — the leaf nodes — are the cycles. If each leaf-task (cycle) can make truly significant progress *and* the tree structure facilitates task spawning (better distribution of work), this technique could be truly efficient.

When the level in the task pool was raised via increased run length, the run

time grew linearly as expected. When the number of processors was raised, the other extreme of this effect was observed: the improvement in run time gained by adding a processor to an n-node system shrank markedly with increasing n. A tailoff was observed where the added processors outdistanced the level in the pool. Both trends verify the hypothesis that processors must be kept busy and indicate what happens when they are not.

Run length and circuit size both fill the task pool, but changing the circuit also changes the *precedences* within the pool. Execution time is strongly affected by circuit topology as well as size. Thus, the amount of parallelism that can be squeezed out of the simulation of a circuit is related to its inherent *hardware* concurrency. An inherently sequential circuit, such as a ring oscillator, cannot really benefit from multiprocessor simulation under the assumptions made in CONSIM. [2] This would not be the case in a different kind of simulator, such as one employing waveform relaxation, where solution requires several iterations on each of the bilateral blocks.

The exact shape of the time vs. number of processors curve contains a lot of information about the actions of the system, but the available diagnostic tools do not allow analysis detailed enough to unearth it. Speedup appears to initially follow a power law other than $t = n^{-1}$ (where t is execution time and n is the number of processors), which may be a result of limited available parallelism — the level in the pool. A more detailed breakdown of *what* task executed *when* and for *how long* would lead to an explanation, but all Concert provides is raw run time. The reasons for the *existence* of the tailoff are understandable from the models, but its *exact shape* suffers from the same lack of illumination. Another explainable trend is that tailoff occurs at a higher number of processors for a larger circuit. This is a clear implication that, in this type of simulator, size is not an all-around evil, as it is in other CAD tools. Large circuits will efficiently exploit the large multiprocessor systems of the future.

CONSIM demonstrates several things. First and foremost, it shows that multiprocessor logic simulation is viable and efficient, potentially even more so than simulation on a classical Von Neumann machine, where circuit size is an inescapable yoke. There is no such thing as a free lunch, of course — although the *real time* is lower, the actual *number of CPU cycles* is the same or slightly higher, since many processors are working in tandem on the job. The price paid is in hardware: n times

[2] Some of these assumptions: unilateral blocks, a solution which is reached in one cycle, and tasks that only know about variables at their own terminals.

as much hardware (n processors) is required to make the simulation run approximately $n^{0.8}$ times faster. Secondly, Multilisp is a good language in which to write simulators. Its constructs fit freely into a structure which efficiently simulates logic circuits. The third result is somewhat amorphous. The body of knowledge gained from modeling and investigating CONSIM's response to changes in the various parameters, like code structure, *future* placement, circuit size and topology, provides a base from which we can make judgments and predictions about as-yet unexplored systems and applications.

Bibliography

[1] H. Abelson and G. Sussman, *Structure and Interpretation of Computer Programs*, M.I.T. Press, Cambridge, 1985.

[2] M. Abramovici *et al*, "A Logic Simulation Machine," *Proceedings of the 19th Design Automation Conference*, June 1982.

[3] T. Anderson, *The Design of a Multiprocessor Development System*, M.I.T. Laboratory for Computer Science TR-279, September 1982.

[4] J. Applegate *et al*, "A Digital Orrery," *IEEE Transactions on Computers*, September 1985.

[5] J. Arnold, *Parallel Simulation of Digital LSI Circuits*, M.I.T. Laboratory for Computer Science TR-333, February 1985.

[6] M. Breuer and A. Parker, "Digital System Simulation: Current Status and Future Trends," *Proceedings of the 18th Design Automation Conference*, June 1981.

[7] R. Bryant, *A Switch-Level Simulation Model for Integrated Logic Circuits*, M.I.T. Laboratory for Computer Science TR-259, March 1981.

[8] S. Chappell *et al*, "Logic Circuit Simulator," *Bell System Technical Journal*, 53:8 1978.

[9] B. Chawla *et al*, "MOTIS — An MOS Timing Simulator," *IEEE Transactions on Circuits and Systems*, December 1975.

[10] E. Cohen, *Performance Limits of Integrated Circuit Simulation on a Dedicated Minicomputer System*, ERL Memo No. ERL-M81/29, University of California at Berkeley, May 1975.

[11] S. Gray, "Using Futures to Exploit Parallelism in Lisp," S.M. dissertation, M.I.T., February 1986.

[12] G. Hachtel *et al*, "The Sparse Tableau Approach to Network Analysis and Design," *IEEE Transactions on Circuit Theory*, January 1981.

[13] G. Hachtel and A. Sangiovanni-Vincentelli, "A Survey of Third-Generation Simulation Techniques," *Proceedings of the IEEE*, October 1981.

[14] R. Halstead, "Multilisp: A Language for Concurrent Symbolic Computation," *ACM Transactions on Programming Languages and Systems*, October 1985.

[15] R. Halstead, "Architecture of a Myriaprocessor," in *Advanced Computer Con-*

cepts, J. Solinsky, ed., La Jolla Institute, La Jolla, California, 1981.

[16] R. Halstead, "Parallel Symbolic Computing," to appear in *IEEE Computer*, August 1986.

[17] R. Halstead *et al*, "Concert: Design of a Multiprocessor System," 13^{th} International Symposium on Computer Architecture, Tokyo, June 1986.

[18] B. Kernighan and D. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, 1978.

[19] M. Matson, private communication.

[20] I. Michaels, "New Circuit Simulators Offer Speed, Accuracy and Convergence," *Computer Design*, 15 November 1985.

[21] L. Nagel, *SPICE2: A Computer Program to Simulate Semiconductor Circuits*, ERL Memo No. ERL-M520, University of California at Berkeley, May 1975.

[22] L. Nagel and R. Rohrer, "Computer Analysis of Nonlinear Circuits, Excluding Radiation (CANCER)," *IEEE Journal of Solid State Circuits*, August, 1971.

[23] A. Newton and A. Sangiovanni-Vincentelli, "Relaxation Based Electrical Simulation," *IEEE Transactions on Computer Aided Design*, October 1984.

[24] A. Newton, "The Simulation of Large-Scale Integrated Circuits," ERL Memo No. M78/52, University of California, Berkeley, July 1978.

[25] J. Ousterhout, "CRYSTAL: A Timing Analyzer For NMOS Circuits," *3rd Cal. Tech. Conference on VLSI*, 1983.

[26] D. Pederson, "A Historical Review of Circuit Simulation," *IEEE Trans on Circuits and Systems*, January 1984.

[27] D. Pederson, "Computer Aids in Integrated Circuit Design", in *Computer Design Aids for VLSI Circuits*, P. Antognetti *et al*, eds., Martinus Nijhoff Publishers, The Hague, 1984.

[28] G. Pfister, "The Yorktown Simulation Engine: Introduction," *Proceedings of the 19th Design Automation Conference*, June 1982.

[29] T. Sasaki *et al*, "MIXS: A Mixed Level Simulator for Large Digital System Logic Verification," *Proceedings of the 17th Design Automation Conference*, June 1980.

[30] S. Szygenda, "TEGAS2 — Anatomy of a General Purpose Test Generation and

Simulation System for Digital Logic," *Proceedings of 9th ACM Design Automation Workshop*, June 1982.

[31] C. Terman, *Simulation Tools for Digital LSI Design*, M.I.T. Laboratory for Computer Science TR-304, 1983.

[32] J. Vlach and K. Singhal, *Computer Methods for Circuit Analysis and Design*, Van Nostrand, New York, 1983.

[33] W. Weeks *et al*, "Algorithms for ASTAP — A Network Analysis Program," *IEEE Transactions on Circuit Theory*, November 1973.

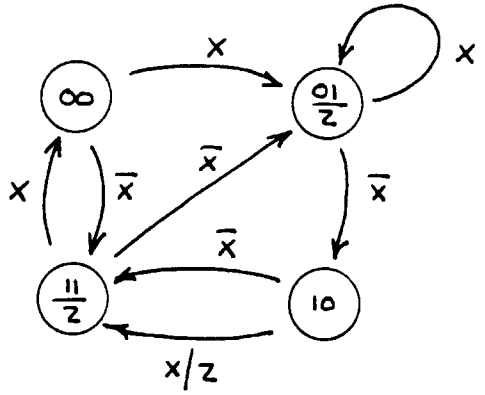
[34] C. Zukowski *et al*, "Bounding Techniques and Applications for VLSI Circuit Simulation," *1985 IEEE International Symposium of Circuits and Systems.*

[35] R. Zippel and G. Clark, "SCHEMA: An Architecture for Knowledge Based CAD," *Proceedings of the International Conference on Computer Aided Design*, November 1985.

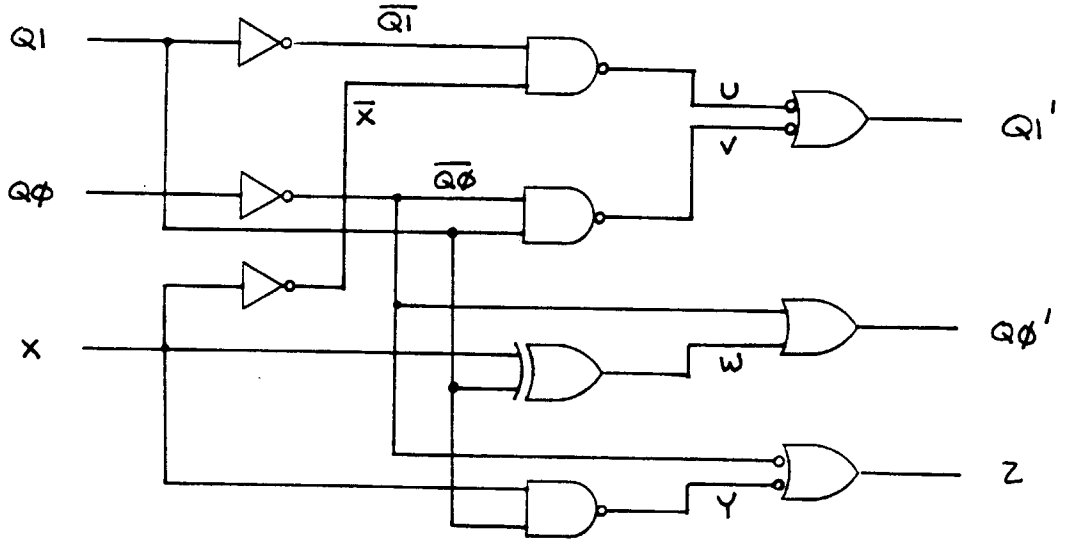
Appendix 1

Schematics and State Transition Diagrams For All Examples

Example 1:

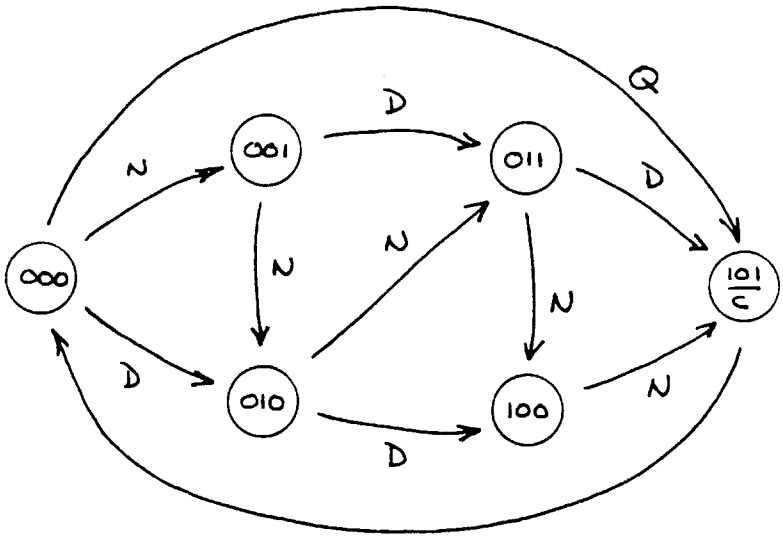


State Transition Diagram

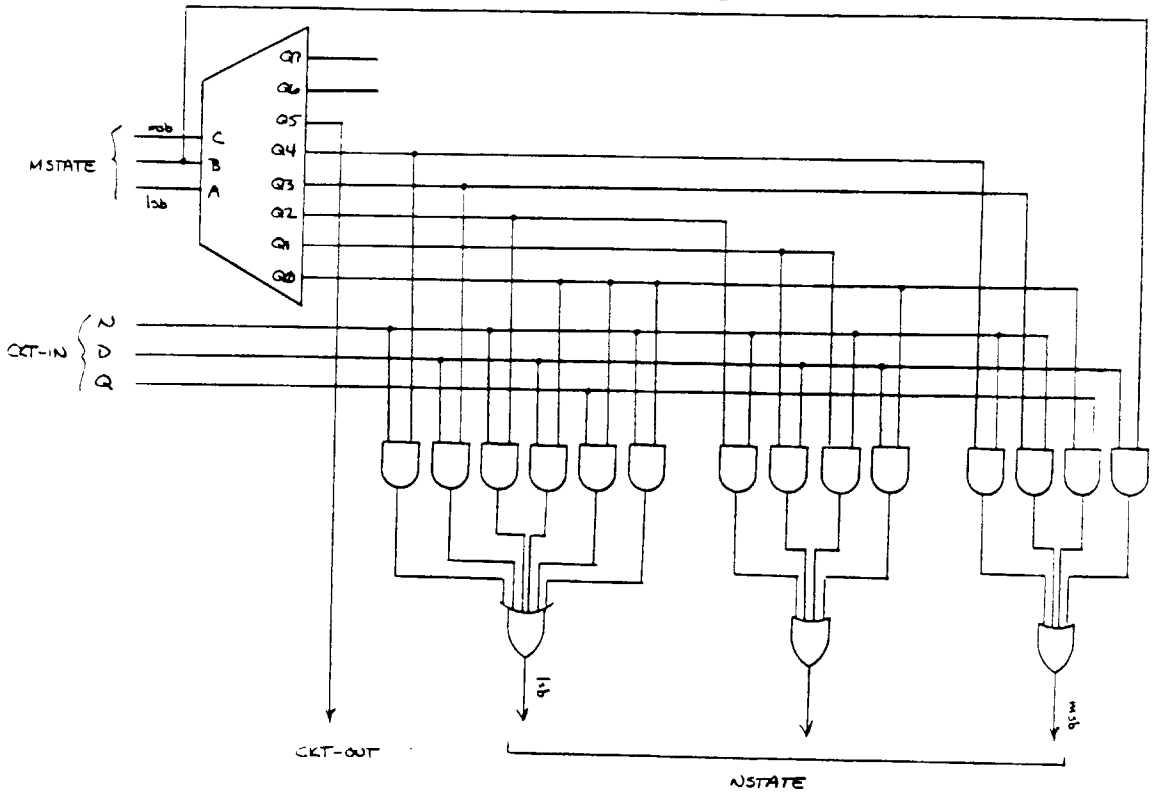


Schematic

Example 2:

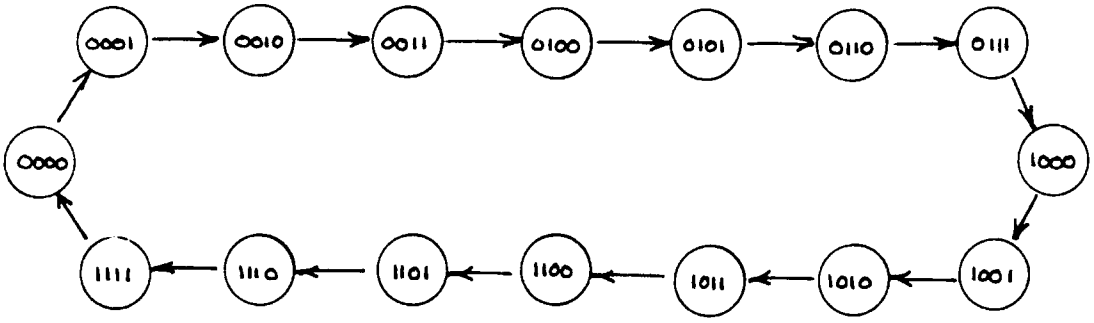


State Transition Diagram

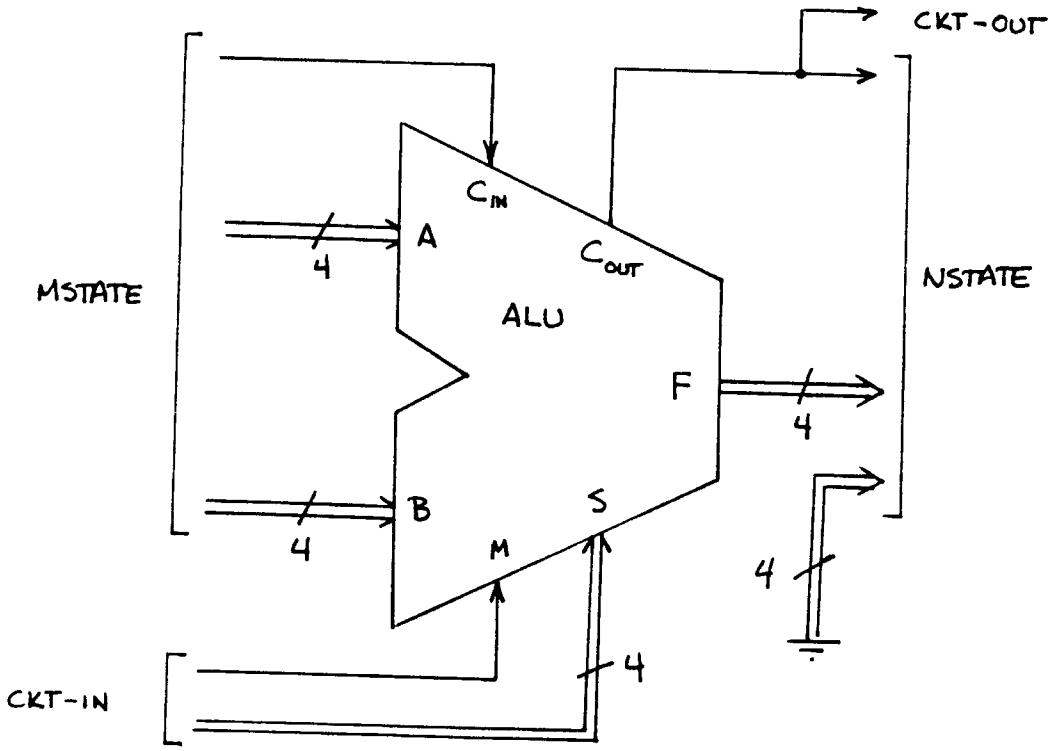


Schematic

Example 3:

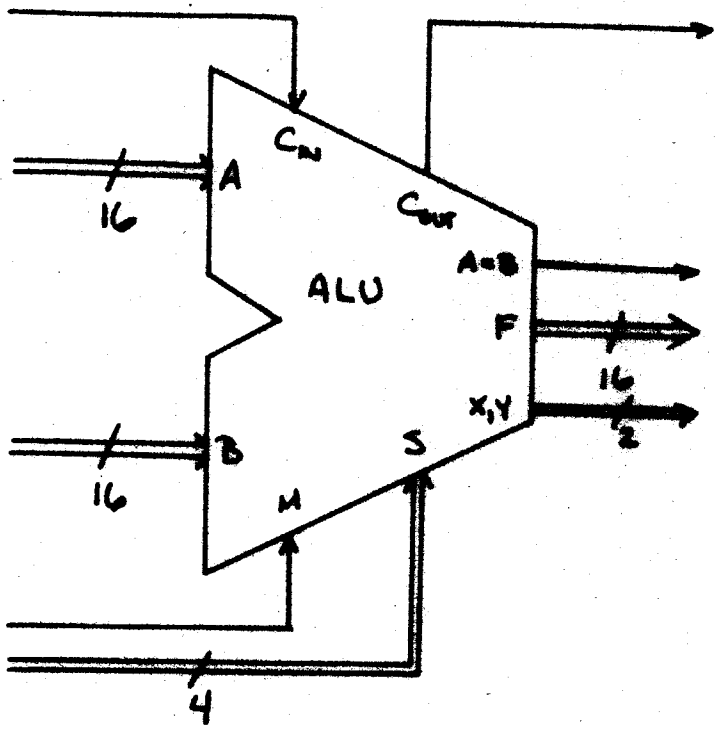


State Transition Diagram — Counter Connection
For M = 0, C_n = 1, S = 0000, F = A + 1.
(Many other modes of operation are possible.)



Schematic

Example 4:



Schematic

Similar to example 3, but data paths are 16 bits wide.

Appendix 2

HDL Descriptions and Single-Cycle Procedures For Example 1

1. HDL Description:

```
(circuit-name fsm)

(mstate (mstate) (q1 q0))
(ckt-in (ckt-in) (x))

((nota) not (x) (x̄))
((notb) not (q1) (q̄1))
((notc) not (q0) (q̄0))
((nanda) nand2 (x̄ q̄1) (u))
((nandb) nand2 (q1 q̄0) (v))
((nandc) nand2 (u v) (q1p))
(xor2 (x q1) (w))
(or2 (q̄0 w) (q0p))
((nandd) nand2 (x q1) (y))
((nande) nand2 (q̄0 y) (z))

(nstate (q1p q0p) (nstate))
(ckt-out (z) (ckt-out))
```

2. Vertical Single-Cycle Procedure:

```
(defun fsm (mstate ckt-in cyc-num)
  (let*
    ((g0000 ckt-in)
     (x (field 1 g0000))
     (g0001 mstate)
     (q1 (field 1 g0001))
     (q0 (field 2 g0001))
     (x̄ (f-not x))
     (q̄1 (f-not q1))
     (q̄0 (f-not q0))
     (u (f-nand2 x̄ q̄1))
     (v (f-nand2 q1 q̄0))
     (q1p (f-nand2 u v))
     (w (f-xor2 x q1))
     (q0p (f-or2 q̄0 w))
     (y (f-nand2 x q1))
     (z (f-nand2 q̄0 y))
     (nstate (list q1p q0p))
     (ckt-out (list z)))
    (list cyc-num nstate ckt-out)))
```

3. Vertical Single-Cycle Procedure With *Futures*:

```
(defun fsmgf (mstate ckt-in cyc-num)
  (let*
    ((g0000 ckt-in)
     (x (future (field 1 g0000)))
     (g0001 mstate)
     (q1 (future (field 1 g0001)))
     (q0 (future (field 2 g0001)))
     (x̄ (future (f-not x)))
     (q̄1 (future (f-not q1)))
     (q̄0 (future (f-not q0)))
     (u (future (f-nand2 x̄ q̄1)))
     (v (future (f-nand2 q1 q̄0)))
     (q1p (future (f-nand2 u v)))
     (w (future (f-xor2 x q1)))
     (q0p (future (f-or2 q̄0 w)))
     (y (future (f-nand2 x q1)))
     (z (future (f-nand2 q̄0 y)))
     (nstate (list q1p q0p))
     (ckt-out (list z)))
    (list cyc-num nstate ckt-out)))
```

4. Horizontal Single-Cycle Procedure:

```
(defun fsms (mstate ckt-in cyc-num)
  (let*
    ((g0618 mstate)
     (q1 (field 1 g0618))
     (g0619 ckt-in)
     (x (field 1 g0619))
     (q̄0 (f-not (field 2 g0618)))
     (nstate (list
         (f-nand2
             (f-nand2 (f-not x) (f-not q1))
             (f-nand2 q1 q̄0))
         (f-or2 q̄0 (f-xor2 x q1))))
     (ckt-out (list
         (f-nand2 q̄0 (f-nand2 x q1)))))
    (list cyc-num nstate ckt-out)))
```

5. Horizontal Single-Cycle Procedure With *Futures*:

```
(defun fsmsgf (mstate ckt-in cyc-num)
  (let*
    ((g0618 mstate)
     (q1 (future (field 1 g0618)))
     (g0619 ckt-in)
     (x (future (field 1 g0619)))
     (q0 (future (f-not (future (field 2 g0618)))))
     (mstate (list
        (future (f-nand2
           (future (f-nand2 (future (f-not x)) (future (f-not q1))))
           (future (f-nand2 q1 q0))))
        (future (f-or2 q0 (future (f-xor2 x q1))))))
     (ckt-out (list
        (future (f-nand2 q0 (future (f-nand2 x q1)))))))
    (list cyc-num mstate ckt-out)))
```

Appendix 3

XML Results

20 Cycle CONSIM Runs

The four versions of the single-cycle procedure are identified by the following abbreviations: "vert" = vertical; "hor" = horizontal; "fut" = *futures* added to code by *Savant*. In cycle-serial invocation, these procedures are called serially; in cycle-parallel mode each cycle is enclosed in a *future*. "Total Ops" is the total number of operations (adds, conses, procedure calls, etc) executed in the simulation. "% Parallelism" is the speedup factor gained on a hypothetical 50-processor machine. "Real-Time Ops" is "Total Ops" divided by "% Parallelism".

Example 1:

Version	Total Ops	% Parallelism	Real Time Ops
1. cycle-serial:			
vert	9189	100	9189
vert/fut	9772	114	8571
hor	8259	100	8259
hor/fut	8740	125	6992
2. cycle-parallel:			
vert	9351	183	5110
vert/fut	9934	257	3865
hor	8403	183	4591
hor/fut	8884	224	3966

Example 2:

Version	Total Ops	% Parallelism	Real Time Ops
1. cycle-serial:			
vert	25424	100	25424
vert/fut	53761	236	22780
hor	24476	100	24476
hor/fut	25813	168	15365
2. cycle-parallel:			
vert	25606	146	17538
vert/fut	53943	1115	4838
hor	24629	137	17978
hor/fut	25967	612	4243

Example 3:

Version	Total Ops	% Parallelism	Real Time Ops
1. cycle-serial:			
vert	74802	100	74802
vert/fut	79542	100	79542
hor	69755	100	69755
hor/fut	74375	174	42672
2. cycle-parallel:			
vert	75036	123	60975
vert/fut	79776	1475	5409
hor	69989	125	55881
hor/fut	74609	1652	4515

Example 4:

Version	Total Ops	% Parallelism	Real Time Ops
1. cycle-serial:			
vert	356566	100	356566
vert/fut	373726	118	316277
hor	337179	100	337179
hor/fut	353859	210	168391
2. cycle-parallel:			
vert	356800	111	321905
vert/fut	187010	1054	17739 (10 cycles)
hor	337413	112	302518
hor/fut	354093	1504	23544

Appendix 4

Concert Time Results

14 Processors

20 Cycle CONSIM Runs

The four versions of the single-cycle procedure are identified by the following abbreviations: "vert" = vertical; "hor" = horizontal; "fut" = *futures* added to code by *Savant*. In cycle-serial invocation, these procedures are called serially; in cycle-parallel mode each cycle is enclosed in a *future*.

Averages of the first two data points are given for all versions. These averages filter the effects of the scheduling, garbage collection, and other time-variant processes which contribute processing time. For example 4, more than two data points were taken to study the pattern of the garbage collection frequency, but only the first two are used in the average.

Example 1:

Version	Time (sec) Number of Processors in Concert System				
	1	2	4	8	14
1. cycle-serial:					
vert	2.750	2.883	2.900	2.883	2.983
vert	2.750	2.883	2.917	2.817	3.016
average	2.750	2.883	2.909	2.850	3.000
vert/fut	4.367	3.717	3.150	3.017	3.200
vert/fut	4.383	3.650	3.117	3.017	3.200
average	4.375	3.684	3.134	3.017	3.200
hor	2.767	2.867	2.950	3.000	3.050
hor	2.783	2.867	2.933	3.000	3.083
average	2.775	2.867	2.942	3.000	3.067
hor/fut	4.167	3.083	2.467	2.083	2.200
hor/fut	4.167	3.050	2.417	2.117	2.133
average	4.167	3.067	2.442	2.100	2.167
2. cycle-parallel:					
vert	3.183	2.000	1.883	1.850	1.933
vert	3.167	1.967	1.867	1.867	1.883
average	3.175	1.984	1.875	1.859	1.908
vert/fut	4.667	3.467	2.417	1.967	1.667
vert/fut	4.667	3.450	2.567	1.900	1.716
average	4.667	3.459	2.492	1.934	1.692
hor	3.200	1.950	1.900	1.917	2.000
hor	3.200	1.933	1.933	1.867	1.967
average	3.200	1.942	1.917	1.892	1.984
hor/fut	4.567	3.367	2.467	1.850	1.733
hor/fut	4.567	3.167	2.417	1.800	1.767
average	4.567	3.267	2.442	1.825	1.750

Example 2:

Version	Time (sec) Number of Processors in Concert System				
	1	2	4	8	14
1. cycle-serial:					
vert	7.167	7.100	7.283	7.483	7.333
vert	7.150	7.117	7.250	7.500	7.467
average	7.159	7.109	7.267	7.492	7.400
vert/fut	20.267	11.667	7.433	5.217	4.250
vert/fut	20.250	11.617	7.450	5.267	4.017
average	20.259	11.639	7.442	5.242	4.134
hor	8.217	8.150	8.317	8.500	8.633
hor	8.217	8.167	8.300	8.467	8.433
average	8.217	8.159	8.309	8.484	8.533
hor/fut	12.033	7.967	5.350	3.967	3.917
hor/fut	12.100	7.633	5.283	4.117	4.000
average	12.067	7.800	5.317	4.042	3.959
2. cycle-parallel:					
vert	7.483	5.700	5.850	6.033	6.300
vert	7.483	5.783	5.917	5.933	6.333
average	7.483	5.792	5.884	6.183	6.317
vert/fut	20.783	12.800	6.800	3.650	3.217
vert/fut	20.417	13.500	6.683	3.850	3.250
average	20.600	13.150	6.742	3.750	3.234
hor	8.533	7.200	6.917	7.250	7.783
hor	8.333	6.900	6.967	7.100	7.667
average	8.433	7.050	6.942	7.175	7.725
hor/fut	12.317	8.550	4.717	3.483	3.083
hor/fut	12.400	8.667	4.567	3.267	3.050
average	12.359	8.609	4.642	3.375	3.067

Example 3:

Version	Time (sec) Number of Processors in Concert System				
	1	2	4	8	14
1. cycle-serial:					
vert	16.667	16.733	16.783	16.917	16.767
vert	16.167	16.183	16.700	17.083	16.667
average	16.417	16.458	16.742	17.000	16.717
vert/fut	25.667	18.550	13.333	11.533	12.350
vert/fut	25.733	18.333	12.983	11.917	12.333
average	25.700	18.442	13.158	11.725	12.342
hor	15.033	15.217	15.417	15.917	16.067
hor	14.950	15.100	15.367	15.917	16.083
average	14.992	15.159	15.392	15.917	16.075
hor/fut	23.567	14.150	9.550	7.367	7.317
hor/fut	23.617	14.650	9.483	7.383	7.117
average	23.592	14.400	9.517	7.375	7.217
2. cycle-parallel:					
vert	16.717	14.817	15.617	15.367	16.050
vert	16.700	14.967	15.467	15.150	16.467
average	16.709	14.692	15.542	15.259	16.259
vert/fut	25.317	17.083	10.217	5.183	4.183
vert/fut	25.333	16.983	9.433	5.967	3.867
average	25.325	17.033	9.825	5.575	4.025
hor	15.500	13.850	14.083	14.017	15.467
hor	15.517	13.600	14.067	14.067	15.550
average	15.509	13.725	14.075	14.042	15.509
hor/fut	23.733	15.533	9.083	5.800	3.833
hor/fut	23.733	15.733	9.300	5.100	3.733
average	23.733	15.633	9.192	5.450	3.783

Example 4:

Version	Time (sec) Number of Processors in Concert System				
	1	2	4	8	14
1. cycle-serial:					
vert	78.183	76.950	79.533	78.100	78.383
vert	75.967	77.833	79.250	78.250	79.350
vert	93.450*	89.467*		89.133*	89.400*
vert	93.516*	89.683*		89.750*	88.750*
average	77.075	77.392	79.392	78.175	78.867
vert/fut	145.38*	93.033*	56.483*	46.650*	26.917
vert/fut	127.95*	84.883*	55.833*	48.683*	22.633*
vert/fut	145.21*				
average	136.66	88.958	56.158	47.667	24.775
hor	71.383	84.967*	74.883	75.200	75.000
hor	88.917*	85.533*	84.017*	84.983*	74.200*
hor	89.083*				
average	80.150	85.250	79.450	80.092	74.600
hor/fut	120.07*	73.567*	41.300*	29.900*	24.050*
hor/fut	119.87*	83.267*	41.933*	28.767*	23.533*
average	119.97	78.417	41.617	29.334	23.792

Note: Averages are based on first two points only.
* = garbage collection

Example 4: (cont)

Version	Time (sec) Number of Processors in Concert System				
	1	2	4	8	14
2. cycle-parallel:					
vert	76.650	87.283*	87.500*	87.850*	88.300*
vert	93.900*	73.617	77.733*	74.300	82.900
vert	94.050*				
average	85.275	80.450	82.617	82.792	85.600
vert/fut	125.32*	80.317*	47.500*	33.783*	21.250*
vert/fut	125.38*	91.983*	55.133*	32.583*	24.850*
vert/fut		92.200*	54.483*		
average	125.35	86.150	51.317	33.183	23.050
hor	89.650*	85.350*	70.067	83.667*	81.800*
hor	89.700*	87.167*	82.333*	71.700	85.317*
hor			81.550*		
average	89.675	86.259	76.200	77.684	83.559
hor/fut	119.27*	75.717*	44.600*	28.583*	25.517*
hor/fut	119.20*	75.100*	48.167*	32.667*	24.367*
hor/fut		75.100*	48.283*		
average	119.24	75.409	46.384	30.625	24.942

Note: Averages are based on first two points only.
* = garbage collection

This blank page was inserted to preserve pagination.

CS-TR Scanning Project
Document Control Form

Date : 4/11/95

Report # LCS-TR-380

Each of the following should be identified by a checkmark:
Originating Department:

☐ Artificial Intellegence Laboratory (AI)
☒ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) ☐ Technical Memo (TM)
☐ Other:_____

Document Information

Number of pages: 93 (102-IMAGES)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:
☒ Single-sided or
☐ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☐ Typewriter ☐ Offset Press ☐ Laser Print
☐ InkJet Printer ☒ Unknown ☐ Other:_____

Check each if included with document:

☒ DOD Form (2) ☐ Funding Agent Form ☒ Cover Page
☒ Spine ☒ Printers Notes ☐ Photo negatives
☐ Other: _____

Page Data:

Blank Pages (by page number): _____

Photographs/Tonal Material (by page number): _____

Other (note description/page number):

Description :	Page Number:
IMAGE MAP (1-6)	PAGES #'ED i - vi (INCLUDING TITLE PAGE)
(7-93)	PAGES #'ED 1-87
(94-99)	SCANCONTROL, COVER, SPINE, PRINTER'S NOTES, DOD(2)
(100-102)	TRGTS (3)

Scanning Agent Signoff:
Date Received: 4/11/95 Date Scanned: 4/13/95 Date Returned: 4/20/95

Scanning Agent Signature: Michael W. Cook

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: MIT/LCS/380
2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle): Logic Simulation on a Multiprocessor
5. TYPE OF REPORT & PERIOD COVERED
6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s): Elizabeth Bradley
8. CONTRACT OR GRANT NUMBER(s): Office of Naval Research · Contract N00014-83-K-0125

9. PERFORMING ORGANIZATION NAME AND ADDRESS:
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139
10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS:
DARPA/DOD
1400 Wilson Blvd.
Arlington, VA 22217
12. REPORT DATE: November, 1986
13. NUMBER OF PAGES: 93

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)
15. SECURITY CLASS. (of this report): Unclassified
15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report):
Approved for Public Release; distribution is unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report):
unlimited

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number):
simulation, logic simulation, Lisp, multiprocessing, parallel processing

20. ABSTRACT (Continue on reverse side if necessary and identify by block number):
The performance of circuit simulators running on SISD computers is fundamentally limited by the Von Neumann bottleneck. Multiprocessors do not share this limitation. The task of solving the equations for the many parallel signal paths found in most circuits lends itself readily to concurrent computation. For both of these reasons, parallel processing is a highly promising approach to circuit simulation. This thesis explores several facets of this problem.

The logic simulator CONSIM was implemented in the parallel language Multilisp, which contains special constructs for dispatching tasks in parallel. A model for the simulator's behavior was developed using a series of experiments. The analysis explains the effects upon CONSIMS's performance of several parameters, including: the number of nodes in the multiprocessor, circuit size and topology, and the algorithms for generating the simulation code and for taking advantage of its inherent parallelism. The final generation of these algorithms exposed and exploited significant parallelism, but did not attain linear speedup.

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**

Scanned

Date: 4/13/1995

M.I.T. Libraries
Document Services