INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

EARTH: AN EFFICIENT ARCHITECTURE FOR RUNNING THREADS

by

Kevin Bryan Theobald

School of Computer Science
McGill University, Montréal
Québec, Canada
May 1999

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH IN PARTIAL FULFILLMENT OF THE REQUIREMENTS OF THE DEGREE OF DOCTOR OF PHILOSOPHY

Abstract

The two current approaches to increasing computer speed are giving individual processors the ability to exploit instruction-level parallelism (ILP), and harnessing multiple processors to work together on a single problem. The first approach has worked well so far, but more and more investments in instruction scheduling hardware seem to be producing diminishing returns. Parallel processing, on the other hand, has tremendous potential, but has been disappointing in practice. One reason is that parallel computer designers cannot afford to put the same resources into building processors as the makers of commodity chips, who command a much larger market. Therefore, most parallel architects have turned to off-the-shelf microprocessors, which do not support parallel processing well.

Multithreaded systems based on dataflow principles promise a solution to the problems inherent in many of today's parallel machines. A multithreaded processor supporting a program execution model designed according to such principles would be a better building block for parallel machines than today's ILP processors, because multithreading and dataflow address the problems facing contemporary parallel machines, such as latency and synchronization. Acceptance of such systems in a world ruled by commodity systems requires taking an *evolutionary* approach, in which the multithreaded machine is initially emulated on an existing multiprocessor based on off-the-shelf microprocessors, while still achieving good performance, and then transformed into more powerful versions by gradually replacing the stock components with custom hardware.

This dissertation is about EARTH (Efficient Architecture for Running Threads), a multithreading model suitable for such an evolutionary approach. The thesis begins with a study of program parallelism which identifies important properties which affect the design of a multithreaded system. The EARTH model is defined at several layers of abstraction, and several implementations along the evolutionary

path are discussed. The Threaded-C language, used for expressing algorithms in the EARTH model, is presented, and several applications are written in this language to illustrate its use.

Experimental results show that EARTH lives up to its name, that it can run parallel programs efficiently, and that this efficiency improves as the machine moves along the evolutionary path and custom multithreading hardware replaces off-the-shelf hardware piece by piece.

Résumé

Les deux approches actuelles utilisées pour accroître la vitesse des ordinateurs consistent à donner à chaque processeur la capacité d'exploiter le parallélisme niveau-instructions (PNI) et à charger plusieurs processeurs de travailler ensemble sur un même problème. La première approche a bien fonctionné jusqu'ici, mais les investissements grandissants dans le matériel d' ordonnancement des instructions semblent semblent donner des résutats décroissants. Le traitement parallèle, d' autre part, offre des possibilités intéressantes, mais a été décevant dans la pratique. Ceci s'explique par le manque de ressources des créateurs d'ordinateurs parallèles par rapport aux fabriquants de puces de produits qui ont accès à un marché beaucoup plus grand. Par conséquent, la plupart des architectes parallèles se sont tournés vers les microprocesseurs déjà disponibles qui ne conviennent pas bien au traitement parallèle.

Les systèmes de *multithread* basés sur des principes de flux de données promettent une solution aux problèmes inhérents à plusieurs machines parallèles d'aujourd'hui. Un processeur *multithread* soutenant un modèle d'exécution des programmes conçu selon un tel principe serait un meilleur fondement pour les machines parallèles que les processeurs PNI d'aujourd'hui, parce que le *multithreading* et le flux de données adressent les problèmes auquels font face les machines parallèles contemporaines, tels le temps d'attente et la synchronisation. L'acceptation d'un tel système dans un monde où règnent les produits systèmes exige une approche évolutive; la machine *multithread* est initialelement émulée sur un multiprocesseur utilisant des microprocesseurs présentement disponibles, tout en réalisant une bonne performance d'exécution, et puis ensuite, elle est transformée en versions subséquantes de plus en plus puissantes par substitution progressive du matériel courant par des composantes matérielles faites sur mesure.

Cette dissertation porte sur EARTH (architecture efficace pour exécuter les

threads), un modèle *multithread* approprié à une telle approche évolutive. La thèse commence par une étude du parallélisme de programme qui identifie les propriétés importantes qui affectent la conception d'un système *multithread.* Le modèle EARTH est défini avec plusieurs couches d'abstraction, et plusieurs implantations sont discutées tout au long du parcours évolutif. Le langage Threaded-C, utilisé pour exprimer des algorithmes en modèle EARTH, est présenté, et plusieurs applications sont écrites dans ce langage pour illustrer son utilisation.

Les résultats expérimentaux prouvent que EARTH fait honneur à son nom; qu'il peut exécuter des programmes parallèles efficacement, et que cette efficacité s'améliore au fur et à mesure que la machine évolue en remplaçant progressivement le matériel existant par des pièces fabriquées sur mesure.

Contributions

The main original contributions of this research are summarized as follows:

1. The design and construction of a tool for analyzing the parallelism in programs, and a study, using this tool, of representative benchmarks, identifying fundamental properties that point to the need for multithreaded architectures;

2. A definition of the EARTH (Efficient Architecture for Running Threads) Program Execution Model, an abstract model describing a way to divide a parallel program into threads and the operations performed on these threads;

3. Definitions of two EARTH Virtual Machines, one based on global addresses and one based on frames, which present specifications of operation sets corresponding to the abstract operations of the Program Execution Model, at a level of detail sufficient for implementation of a real system;

4. Detailed specification and high-level design of a custom hardware Synchronization Unit providing efficient support for the EARTH Program Execution Model;

5. Development of a tool for accurate simulation of an existing off-the-shelf multiprocessor, and the use of this tool to:

   (a) Measure the performance of the multiprocessor with a greater number of processors than available with the current hardware;

   (b) Measure the performance of the multiprocessor augmented by the custom hardware Synchronization Unit, thereby demonstrating the efficiency of the EARTH model when there is hardware support for multithreading;

   (c) Measure the performance of the multiprocessor, with and without the custom SU, with different processor parameters to confirm the benefits

of the EARTH model on processors built after those used in the multiprocessor platform;

6. A study of possible extensions to EARTH and Threaded-C which could improve both runtime efficiency and programmability.

As in any project developing a computer system, many people were involved in EARTH's design, implementation and experimentation. The following contributions are not the exclusive work of the author, but the author played a major role in their execution:

1. A definition of the Earth Architecture Model, describing an architecture appropriate for executing programs under the EARTH Program Execution Model;
2. A definition of the Threaded-C language, an explicitly threaded language extending standard C with EARTH operators;
3. Coding of various benchmarks in Threaded-C so that they may be tested on EARTH platforms;
4. Implementations of EARTH on several off-the-shelf platforms, and experiments showing the performance achieved by Threaded-C benchmarks on these platforms.

Acknowledgements

My advisor, Prof. Guang R. Gao, was the inspiration for this thesis and the EARTH project. He has the vision to conceive a great project, the wisdom to manage it well, and the drive and perseverance to carry it through. He encourages and inspires everyone in his group to put out their best. He has taught me numerous valuable lessons, and it was a privilege to work with him.

EARTH was a team effort, and would not have been possible without the efforts of many other people. Prof. Herbert Hum was one of the co-originators of the EARTH concept and the leader of the implementation effort. Dr. Olivier Maquelin did a tremendous job coding the main parts of the EARTH system and Threaded-C compiler almost singlehandedly. Andres Marquez taught me many things about the MANNA memory system, and wrote the first version of the memory module for the SEMi simulator. He also performed the initial feasibility study of SU hardware design. Shashank Nemawarkar and Dr. Xinmin Tian made important analyses of the performance of EARTH. Dr. Maquelin, Dr. Tian, Xinan Tang, Nasser Elmasri and Yingchun Zhu wrote some of the application benchmarks used in this study. Phillip Mueller and Pascal Davoust made contributions to the SEMi simulator. Haiying Cai and Prasad Kakulavarapu made major improvements to the load balancers. On the compiler side, Prof. Laurie Hendren was one of the principal designers of the EARTH-C language, as well as the founder and leader of the McCAT compiler on which the EARTH-C-to-Threaded-C translator is based. Xun Xue, Pierre Ouellet, Xinan Tang, and Yingchun Zhu did most of the work on this translator. Prof. Gao and Hendren also participated in the SITA studies of program parallelism which preceded the start of the EARTH project.

The CAPSL group at the University of Delaware continues to make great strides in their continuation of the EARTH project. Cheng Li did a major porting effort to get EARTH working on Beowulf. I have had many insightful conversations about

language design and applications with Dr. Gerd Heber, Parimala Thulasiraman, and Dr. Jose Nelson Amaral, and with Thomas Geiger about SU design. Dr. Amaral also wrote the Threaded-C code for the mutex example used in this thesis.

Thomas Geiger, Xinan Tang, Dr. Heber and Dr. Amaral reviewed portions of my thesis. Etienne Gagnon (McGill) translated my abstract into French.

GMD (Gesellschaft für Mathematik und Datenverarbeitung) generously provided the ACAPS group with a MANNA parallel machine, without which this study would have been impossible, and provided excellent technical support. The National Sciences and Engineering Research Council (NSERC) of Canada provided the bulk of the remaining funding for the EARTH project. Several American research agencies, including DARPA, NASA, NSF and NSA, funded my research while I was working in Delaware.

On a personal level, I have had the support of many friends and colleagues. Among those not already listed are the ACAPS members Russ and Yoshiko Olsen, Bob Yates, Chandrika Mukerji, V. C. Sreedhar, and Erik Altman, with whom I had many stimulating conversations. Ping Gao was a wonderful advisor in many areas outside of the lab. Outside of ACAPS, Honglang Li, Carol Novitsky, Rishan Tan, and Cameron Wakefield were always there for me, even if I couldn't always be there for them. And Bing Wang gave me all of her love and understanding during the difficult final stages of research.

Finally, my family, especially my parents, were tremendously supportive and infinitely patient through these long years. They gave me far more than I could ever give back.

In memory of my sister *Kathie*

Contents

List of Figures

Chapter 1

Introduction

The 1980s saw the popularization and commercialization of parallel processing. What was mostly confined in the 70s to university project laboratories moved to industry as computer architects tried to bring the benefits of parallelism to the marketplace. Many companies were formed during the 80s, and their competition led to a diverse set of approaches to the problem of dividing a task among many processors. However, commercial parallel processing has failed to live up to its promise. Most of the high-flying parallel computer companies have gone bankrupt or have re-oriented themselves toward specialized markets and applications.

There are several reasons for this. First, today's microprocessors are highly complex and require a huge investment, which can only be recovered through high sales volumes. Therefore, most parallel computer makers have had to turn to the "killer micros" [16], off-the-shelf processors built for the workstation and personal computer mass markets. These chips generally don't offer good support for multiprocessing. Second, parallel computers have been notoriously difficult to program, so that their market has been limited to a small number of institutions with both the computing needs and the available programming resources to justify buying one.

Nevertheless, there is still a need for parallel processing. Predictions that a certain level of performance would be "enough" have time and time again been proven wrong. Even though today's microprocessors have performances rivaling last decade's supercomputers, applications expand to consume the available power, and increases in power lay the ground for new applications to be created. Furthermore, some useful applications are NP-complete, creating a potentially unbounded demand

for computing power. An application which is barely feasible on a given state-of-the-art computer becomes much more viable if it can be run a hundred times faster. It is only a question of whether that can be done sufficiently cheaply [104].

Therefore, where should architecture go from here? Will the present trends in processor evolution lead to any significant improvements? If not, what path should be taken instead? Will parallel machines always remain at the mercy of the "killer micro" market, or is it possible to serve both domains at the same time? It is the goal of this research to find answers to these questions.

1.1 ILP and Multithreading

A traditional von Neumann processor has a single program counter which always points to the next instruction to be executed. A processor built strictly according to this principle can execute at most one instruction per cycle (IPC). Modern microprocessors go beyond this by exploiting *Instruction-Level Parallelism* (ILP), parallelism through the simultaneous execution of individual instructions that are near each other in the instruction stream. Both *VLIW* (Very Long Instruction Word) and *superscalar* processors allow multiple instructions to execute simultaneously, while still presenting the programmer with the *appearance* of a sequential program counter.

A superscalar processor is programmed by the user as if it were a single-instruction-per-cycle processor with the same instruction set. Correct functional behavior of the code is described by the semantics of a sequential, instruction-by-instruction execution of the same code. However, the processor can dynamically choose multiple instructions to execute at the same time, and may even issue instructions out of order, provided the sequential semantics are preserved. When instructions execute simultaneously or out of order, the hardware checks to ensure that only *independent* instructions are executed concurrently. The number of instructions executed in a given cycle can vary from cycle to cycle, and depends on the number of instructions available for execution and the dependences between them.

In a VLIW processor, instructions representing basic operations are combined at compile time into "very long instruction words" of fixed length. These long words are scheduled at run time using an ordinary program counter, which reads and executes them sequentially unless there is a branch. Typically, each field of a VLIW

instruction, corresponding to one basic operation, is sent to a different functional unit, depending on its location in the instruction. There generally are restrictions on how operations can fill instructions (e.g., each VLIW instruction can consist of a floating-point add, a floating-point multiply, two integer operations, and two load-store operations). If the code requires a different balance of operations, or if not enough independent operations can be found to fill the long instructions, then some fields are filled with NOPs.

VLIW and superscalar have succeeded in breaking the 1-IPC barrier, but not by much. In spite of all the extra functional units, and all the extra hardware to allow simultaneous execution, most processor designers have not been able to achieve even 2 IPC on representative benchmarks, despite having potential issue rates of 4 or even 8 IPC. There are several reasons for this. As will be shown in the next chapter, the number of independent instructions within a single *basic block* (a sequence of instructions that is always entered at the beginning and exited at the end, with no branches to or from the middle) is limited. This problem is exacerbated by the fact that all the additional hardware needed to support out-of-order execution adds so much extra complexity to the processor that the number of stages in the fetch and execution pipeline is usually much higher than in a simple RISC processor.

Getting parallelism beyond that available in a basic block requires looking beyond conditional branches when scheduling instructions. A hardward-based approach, for instance, would be to use *branch prediction*, so that the processor can *speculatively* execute instructions beyond the branch and move the results of those instructions into the permanent CPU state once the branch outcome is known. Unfortunately, branch prediction is not always correct, and given the large number of pipeline stages between instruction fetch and resolving a conditional branch, a misprediction exacts a large penalty.

Therefore, it is not likely that the current path of processor development, which attempts to extract more and more parallelism from a single thread of code, will yield many more *architectural* improvements (those not due merely to faster clocks). There are ambitious designs on the drawing boards for processors capable of executing larger numbers of instructions simultaneously, as many as 16 or more, but experience with present processors is not promising, and suggests that additional hardware is likely to produce diminishing returns.

Several architecture research projects have shown that *multithreaded* processors [41, 65] can find and exploit more parallelism than processors that look only for ILP. A multithreaded processor differs from a single-thread processor in that the machine model allows the concurrent execution of instructions from different locations in the code. An advanced single-thread processor may allow different instructions to execute simultaneously, but these all come from a single thread of instructions. A multithreaded processor, in broadest terms, has "hardware support for multiple program counters" [114]. By allowing execution from different sections of code, a multithreaded processor has more places from which it can extract ILP. Multithreaded machines are surveyed in Chapters 3 and 9.

1.2 Fundamental Issues in Multiprocessing

From the discussion in the previous section, it can be concluded that getting more parallelism than the small-scale parallelism offered by the current-generation superscalar and VLIW processors will require exploiting multiple threads of control simultaneously. Some recent proposals (included in the surveyed in Chapter 9) seek to boost uniprocessor performance by using multithreading (with some speculation) to expose more parallelism than just ILP, but they still try to extract this parallelism from sequential code. This will make their immediate acceptance more likely, but it will also limit how much parallelism they can ultimately get.

Going beyond that will require running threads on multiple processors, which will require a paradigm shift on the part of mainstream programmers. Unfortunately, while multiprocessors have been proposed for decades, there are some fundamental problems with multiprocessing which most designs have not adequately addressed. Arvind and Iannucci [9] identified two "fundamental issues" of multiprocessing: latency and synchronization. Several other important issues are bandwidth, programmability and manufacturability. These five issues are elaborated below.

1.2.1 Latency

One fundamental problem is based on a simple fact about today's technology: For non-trivial applications, it is physically impossible to keep all data required by the computation close enough to the processor to be instantly accessible at all times.

For the purposes of this discussion, "instantly accessible" can be taken to mean "without causing delay," which for RISC processors means a very small number of cycles, preferably one.

The traditional solution for uniprocessors has been to form a *memory hierarchy,* consisting of most of the following: registers, on-chip caches (single-level, two-level, or hybrid-access [121]), off-chip caches, main memory, and secondary storage (disks and tapes). Elements higher in the hierarchy are placed physically closer to the CPU than those below. This proximity means both that the storage locations closer to the CPU can be accessed more quickly, and that they are less numerous than storage locations further away. The hierarchy is generally effective because most programs exhibit *temporal locality* [110]. In most cases, objects which have been accessed recently are more likely to be referenced again in the near future than objects which have not been accessed recently. Therefore, it is useful to keep these objects closer to the CPU.

This technique breaks down when there is more than one processor, because the memory hierarchy can no longer be viewed as a simple pyramid with the CPU at the apex. Instead, the "hierarchy" becomes a set of pyramids, whose bases must merge at some level in order for sharing of data to occur. For all but so-called "embarassingly parallel" applications, whose computations can be divided into independent sections running concurrently without any need for communication or cooperation, data is likely to be needed by different processors at different times. Due to the dispersal of processors, data cannot be kept physically close to all processors at the same time. If a processor requires a datum not close to it, it will have to wait for that datum to be fetched from somewhere else in the system, such as another processor.

How the processor handles this *latency* affects performance significantly. Multiprocessors based on traditional processor designs typically stall while waiting for data to return from the remote fetch. The penalty, in terms of cycles lost, of this stalling is generally more severe than the cost of a cache miss on a uniprocessor, because the data will have to come from further away. The problem increases as more processors are added, first, because the average distances, and hence latencies, between processors increase, and second, because if the application is spread among a greater number of processors, then more of the data needed by each processor will be located on remote processors.

1.2.2 Bandwidth

One cannot successfully treat the latency problem without dealing with another problem: *bandwidth*. Since it takes longer to send a signal over an IC pin than along a wire in the interior of the chip, it follows that the throughput on that pin will be lower than the throughput along the wire. The fact that IC pins must be much larger than internal wires restricts their numbers, which causes external throughput to drop even further relative to internal throughput.

A simple calculation will illustrate why latency can't be separated from bandwidth considerations. Suppose that in a particular application, 30% of all operations are loads and stores (which is typical for numerical applications), and that 10% of these require going off-chip (e.g., they miss the cache, write through the cache to external memory, or perform explicit communications with remote processors). A 10% miss rate is a reasonable assumption for applications with large working sets; it can even be seen on uniprocessors, and is more likely to be the case on large-scale parallel processors, for the reason given at the end of Section 1.2.1. Suppose that an average of 8 CPU cycles are required in the actual transfer of data to or from the CPU (a reasonable assumption, given today's aggressive clock rates). Then 3% of all instructions executed will occupy the CPU's external interface for 8 cycles. A processor running under these conditions will not be able to execute, on average, much more than 4 IPC, no matter what latency-handling mechanisms are installed!

1.2.3 Synchronization

If processors are working together on a non-embarassingly-parallel application, there are times when one processor will require data created by another. They will have to coordinate their activities so that the producer of data knows where and when to send the data, and the consumer knows when the data has arrived.

In the simplest class of multiprocessors, called SIMD (Single Instruction/Multiple Data) machines [34], instructions are fetched from a single global instruction stream and broadcast simultaneously to all processors. Thus, every processor performs the same operation simultaneously, but on different data. These instructions generally include communications operations which allow processors to transmit data to one another. Since all processors are controlled by the same instruction, they all must transmit data, or receive data, at the same time. This makes

synchronization an easy task on such machines, because the programmer (or compiler) is forced to control every communication event, and every communication has a uniform, predictable latency. It is mainly the extreme difficulty in programming SIMD machines for most applications that has limited their commercial acceptance.

In the more general MIMD (Multiple Instruction/Multiple Data) class of machines [34], each processor is free to execute its own instruction stream. Since the producer and consumer of data are no longer in lock-step, as in the SIMD machines, they may be out of sync when the communication is to occur, e.g., either the consumer will not be ready when the producer sends the data, or the producer will not be ready when the consumer wants it. The simplest solution is to have one wait for the other, e.g., to have the producer wait for an acknowledgement from the consumer before continuing, but this can waste a lot of processor cycles, particularly if the communication events occur at different times on different processors.

A better solution is to have the processor do some other work while waiting for the response, much as an operating system will swap to another process while waiting for a page to be read from disk. But this imposes its own costs. The largest overhead, and the focus of the paper by Arvind and Iannucci, is the time to perform the *context switch*, i.e., to save the state of the current computation (as contained in the registers) and load the state of the new computation into the registers.

1.2.4 Programmability

As previously mentioned, SIMD machines did not gain wide popularity because they were difficult to program. They are hard to program because most problems can't be expressed as a uniform homogeneous computation over an array of elements, which is what SIMD machines are designed to do. But SIMD machines are only the worst case of a problem facing all parallel machines: ease of programming. Simply put, for general applications it is difficult to program parallel machines and get performance anywhere near the theoretical performance level of the machine (generally the performance of a single processor multiplied by the number of processors).

In some cases, poor performance on parallel machines may be due to an inherent lack of parallelism in the application itself. These cases are discussed at length in the next section. Sometimes, these can be improved by modifying the algorithm or restating the problem, but other times the problem simply cannot be parallelized.

Usually, however, applications are difficult to parallelize because there is no universal model for parallel machines. For decades, sequential machines have all been based on the von Neumann model, with its program counter, arithmetic units, register(s), and addressable memory, augmented by indirect addressing and the use of a stack. The basic model has remained unchanged through all the architectural enhancements over the years (virtual memory, caches, pipelining, even superscalar). This has made it possible to write programs that are essentially portable from one machine to the next, and for programmers to become used to a common paradigm of programming. On the other hand, there are many different models for parallel machines; for instance, the memory hierarchies looks quite different (to the programmer) on shared-memory and message-passing machines.

Since there is no uniform model for parallel computing, designing an efficient portable parallel programming language is practically impossible. Thus, programmers are forced to concern themselves with specific details of the target machine, and must extensively modify code written for other parallel machines. It is as if the machine had no compiler, and programmers were forced to write in a machine-specific assembly language. Attempts to develop languages that "abstract away" all details of the parallel machine from the programmer have not solved the problem, because the great differences among the models require that so much be abstracted from the user that efficient code cannot be generated. (Languages like Fortran and C have been successful *because* they do not abstract too much away; the differences in sequential models are sufficiently insignificant that it is possible to abstract away enough low-level details to make the languages useful while giving the programmer access to enough details to generate efficient code.)

Another reason for the difficulty in programming parallel machines is that most do not adequately address the problems of latency, bandwidth and synchronization already discussed. Programmers must spend extra time trying to tune their programs to compensate for the weaknesses of the architecture. If machines did not have these deficiencies, programmers could spend more time concentrating on the high-level details of the program.

This idea is illustrated in Figure 1.1. In this graph, there are three architectures (A, B, and C) with identical peak performance levels, represented by the top dashed line. The horizontal axis represents the amount of programmer effort required to achieve a particular level of performance (plotted on the vertical axis) for a given

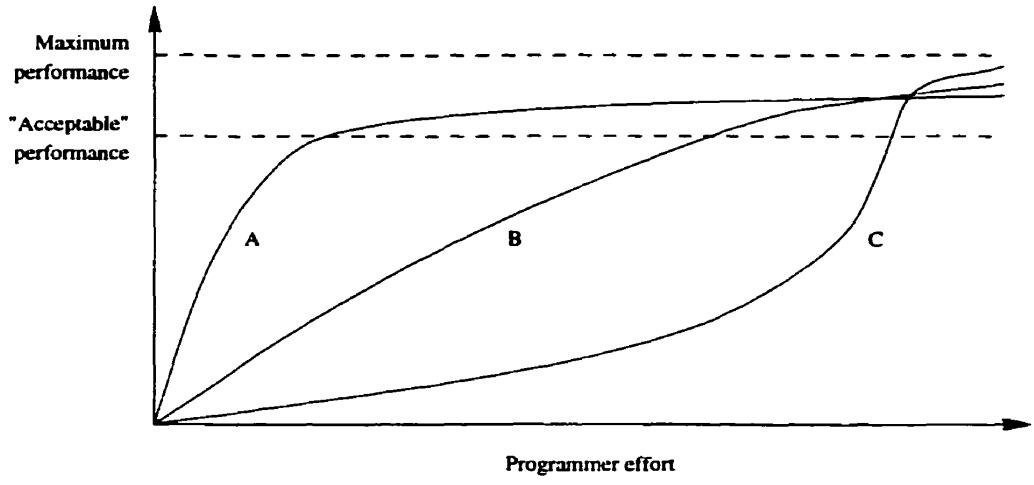


Figure 1.1: Performance Payoffs for Different Architectures

application, which we assume has enough parallelism to make this effort worthwhile. Architecture C exposes most of its details to the programmer, requiring a lot of effort to fine-tune the program to get the most speed. Ultimately, given enough programming, machine C can run the application the fastest. However, if users decide that a given fraction of peak performance is acceptable (represented by the bottom dashed line), then programmers will be able to reach "acceptable" performance much more quickly with architecture A.

1.2.5 Manufacturability

Even if one can overcome the fundamental problems just discussed, and design a parallel architecture that can deliver good performance, one also has to be able to build this machine in today's competitive market. Manufacturers of single-thread processors have squeezed everything they can out of RISC technology by pouring enormous resources into design efforts. Even with advanced CAD tools, a state-of-the-art processor takes hundreds of person-*years* to develop. This investment can only be recouped through high-volume sales, mostly in the workstation and high-end PC uniprocessor markets.

If the processors in parallel computers don't match the high-end microprocessors in performance, or at least come close, the benefits of combining processors in parallel will be erased by their inferior speed. This situation is what Eugene Brooks [16]

called "the attack of the killer micros." If parallel computer makers design processors that only work in large parallel machines, they won't be able to make enough sales to justify the design costs. They must either settle for inferior processor speed and hope that massive parallelism will make up the difference, or else, following the old maxim "if you can't beat 'em, join 'em," use off-the-shelf microprocessors.

Mainstream microprocessor manufacturers have responded by putting in some basic multiprocessing support in their chips, such as support for local cache consistency. However, these features only begin to address the problems outlined in the preceding subsections. If parallel machines are to become commercially viable, they will need to be based on processors with more substantial support for parallel computing. If microprocessor manufacturers are to be persuaded to add these features to their chips, then the features will have to give some benefit to the mainstream uniprocessor market. If they don't, they should at least not *interfere* with uniprocessor performance, and should present as little cost (in extra chip area) as possible so that the costs of their addition can be recovered through increased sales to the parallel computer market.

1.3 An Evolutionary Approach to Viable Parallel Processing

If a large quantum leap to a full-featured microprocessor supporting parallel programming is not commercially viable in today's marketplace, then efficient support for parallel processing will have to be introduced gradually. Such steps as can be made must be small, relatively risk-free, and produce tangible benefits. Therefore, it is most likely that multiprocessing systems will take an *evolutionary* path. Each step along the path should represent a small enough cost to allow a prototype to be built, and should improve upon the performance of the preceding step to establish the merits of the features added in that step. The following are the likely phases in this evolution:

1. Use of an existing parallel system, based on off-the-shelf microprocessors, to emulate a multiprocessing model well enough to demonstrate its viability.

2. Construction of a hybrid system, using off-the-shelf microprocessors to perform the regular computations, and custom auxiliary hardware to support

the instructions unique to the multiprocessing model. The custom hardware should improve the performance of the machine compared to the first machine (the emulated system).

3. Design of a hybrid chip containing the original core of the stock microprocessor and the extra custom hardware. The combination of the two components on one chip should reduce communication delays between the two and allow better sharing of common resources, such as caches.

4. Creation of a fully-integrated processor for a parallel system, one which also performs well in a uniprocessor environment.

The key to the success of such an approach is choosing a good programming model which allows programmers to express parallelism without much difficulty and without sacrificing efficiency, yet is fully portable along the evolutionary path. The latter requirement is crucial, for if programmers, having spent the effort to parallelize an application, have to redo this effort to take the next evolutionary step, few will bother to take that step. Users should be able to write an application once, according to the given programming model; as improved versions of the parallel machine are introduced, the application should run on the new machines with no more modification than recompiling. The model should be simple and efficient enough to yield reasonable performance even in the early evolutionary stages, yet flexible enough to produce even better results as the new machines are created.

To summarize, the main question driving this research is:

> What should be the architecture of a parallel computing system which can effectively handle the problems of latency, bandwidth, and synchronization, can provide a sufficiently general programming model, and can provide a viable evolutionary path from mainstream architectures?

We believe that *multithreaded* computers based on the *dataflow* model of communication and synchronization [24] have the potential to satisfy these requirements. Multithreading by itself addresses the latency problem, because the processor can execute instructions from another section of code when one instruction is blocked by a long-latency operation. On the other hand, dataflow scheduling provides an efficient form of synchronization. It states that any instruction is eligible for execution as soon as the operands it requires are ready. The two paradigms are combined

to get both the latency-tolerating capabilities of multithreading and the flexible synchronization of dataflow.

1.4 The EARTH Project

The *Efficient Architecture for Running THreads* (EARTH) [60, 61, 59, 83] project at McGill University is intended to demonstrate that a programming model meeting the requirements in Section 1.3 can be designed and implemented. Table 1.1 lists the major milestones in this project. EARTH is a large team effort; four professors, two post-doctoral fellows, and more than a dozen graduate students have been involved.

EARTH began in the Fall of 1993, when the author, Prof. Guang Gao (McGill) and Prof. Herbert Hum (Concordia University) developed a model for the efficient implementation of multithreading on off-the-shelf microprocessors with minimal additional hardware support for multithreading. Each of the three had previously designed multiprocessors based on dataflow principles [37, 62, 122] (see Section 3) and knew the tradeoffs associated with such machines. The ACAPS (Advanced Compilers, Architectures and Parallel Systems) group at McGill and Concordia formed a partnership with GMD (Gesellschaft für Mathematik und Datenverarbeitung) in Berlin, which loaned a multiprocessor called MANNA [17], developed at GMD, to ACAPS.

Development of an emulator on an off-the-shelf multiprocessor, the first stage in the evolutionary path, began in earnest in Spring, 1994, after the machine was delivered to McGill.[1] At that time, the Threaded-C language was defined as the target language for application and compiler writers. Most of the design and implementation of the runtime system and back-end compilation environment (described in Chapter 7) was done by Dr. Olivier Maquelin, who joined ACAPS in 1994 after previous research in dataflow [82]. Some benchmarks were running by July of that year, and the MTA-MANNA system was shown at Supercomputing '94.

Subsequent work has focused on porting the EARTH model to various machines,

[1] Prior to 1995, this project was called the Multi-Threaded Architecture (MTA). However, this name is currently used by Tera Corporation for their multithreaded machine (see section 9.1). Private discussions with Burton Smith revealed that both groups independently started using the acronym "MTA" at about the same time circa 1992. We have changed the name of our machine to avoid confusion (and because we like the new name better).

Time	Event or design	Participants
Sept. 1993	First MTA position paper	Hum, Theobald, Gao
Sept.	Partnership with GMD established	ACAPS, GMD
Mar. 1994	2-node Mini-MANNA installed at McGill	GMD, ACAPS
Jul.	First programs on EARTH-MANNA-D	Maquelin, ACAPS
Nov.	MTA-MANNA demo at SC'94	ACAPS
Dec.	First i860 simulator (SEMi)	Theobald, Mueller
Feb. 1995	Project formally renamed EARTH	ACAPS
Feb.	First "portable" version of EARTH	Maquelin
Feb.	First EARTH-C-to-Threaded-C translator	Xue, Tang, Ouellet, Hendren
Apr.	SEMi extended to MANNA simulation	Theobald, Marquez
Aug.	20-node MANNA installed at McGill	GMD, ACAPS
Jul.	EARTH-MANNA-S implementation	Maquelin
Aug.	Polling watchdog on EARTH-MANNA-S	Maquelin, Theobald
Sept.	Polling watchdog added to SEMi	Theobald
Mar. 1996	EARTH on SP-2	Maquelin, Cai
Nov. 1997	EARTH on 4-node Beowulf	Cheng
Jan. 1998	EARTH on 60-node Beowulf	Theobald, Cheng
Apr.	EARTH on Sparc SMP Cluster	Cheng
Jul.	Functional design of SU complete	Theobald
Aug.	Portable EARTH on PowerMANNA	Heber, Theobald
Oct.	SEMi extended to SU simulation	Theobald
Nov.	EARTH-PowerMANNA demo at SC'98	CAPSL, GMD

Table 1.1: Milestones in the EARTH Project

tuning the implementations for better performance, and experimenting with architectural enhancements. A separate language development effort is running in parallel, with the goal of creating a higher-level parallel language without the need for explicit threads, and writing a compiler to translate this language to Threaded-C. Major participants in the language effort include Prof. Laurie Hendren, Haiying Cai, Pierre Ouellet, Xinan Tang, Xun Xue, and Yingchun Zhu. Other institutions that have been involved in the EARTH project include the University of Delaware, the University of Southern California, and GMD.

1.5 Contributions

As in any project developing a computer system, many people were involved. The following original contributions are solely or primarily the work of the author:

1. The design and construction of a tool for analyzing the parallelism in programs, and a study, using this tool, of representative benchmarks, identifying fundamental properties that point to the need for multithreaded architectures (this study actually preceded the start of the EARTH project);

2. A definition of the EARTH (Efficient Architecture for Running Threads) Program Execution Model, an abstract model describing a way to divide a parallel program into threads and the operations performed on these threads (while the EARTH model was co-developed by Gao, Hum and the author, the author provided the formal definition);

3. Definitions of two EARTH Virtual Machines, one based on global addresses and one based on frames, which present specifications of operation sets corresponding to the abstract operations of the Program Execution Model, at a level of detail sufficient for implementation of a real system;

4. Specification and high-level design of a custom hardware Synchronization Unit providing efficient support for the EARTH Program Execution Model;

5. Development of a tool for accurate simulation of an existing off-the-shelf multiprocessor, and the use of this tool to measure the performance of the multiprocessor with more processors than available with the current hardware (to measure scalability), a custom hardware Synchronization Unit (to test the efficiency of the EARTH model when there is hardware support for multithreading), and with different processor parameters (to confirm the benefits of the EARTH model on processors built after those used in the multiprocessor platform);

6. A study of possible extensions to EARTH and Threaded-C which could improve both runtime efficiency and programmability.

The following contributions are not the exclusive work of the author, but the author played a major role in their execution:

1. A definition of the Earth Architecture Model, describing an architecture appropriate for executing programs under the EARTH Program Execution Model;

2. A definition of the Threaded-C language, an explicitly threaded language extending standard C with EARTH operators;

3. Coding of various benchmarks in Threaded-C so that they may be tested on EARTH platforms;

4. Implementations of EARTH on several off-the-shelf platforms, and experiments showing the performance achieved by Threaded-C benchmarks on these platforms.

1.6 Synopsis

This dissertation is organized as follows:

Before building a parallel processor to run a class of applications, one should analyze the applications to know both how much parallelism is intrinsic to each application and what architectural properties are necessary to achieve this parallelism. Chapter 2 reviews a study performed by the author for this purpose, based on real applications. (A survey of previous studies is provided in Appendix A.) The results show that most applications have enough inherent parallelism to keep at least a moderately-sized parallel computer usefully busy, but very little parallelism exists at the instruction level, meaning that the ILP techniques discussed in Section 1.1 will not produce the desired performance.

EARTH has its roots in the dataflow model of computation. Chapter 3 explores the history and evolution of dataflow. It begins with a review of basic dataflow concepts, and surveys the machines designed according to dataflow principles, focusing on those machines most related to EARTH. The chapter then discusses multithreaded machines based on dataflow, particularly the multithreading work at McGill which preceded EARTH, and includes a survey of other dataflow-based multithreaded machines in the literature.

Chapter 4 defines the most fundamental properties of EARTH, the *Program Execution Model* (PXM) and *Architecture Model*. The first model describes how a program is divided into threads, how these threads are created, coordinated and synchronized, and how they share data. The second model describes, in general

terms, the structure of a multiprocessor appropriate for this model. The chapter concludes by discussing possible future extensions to the models.

The PXM defines, in general terms, the types of operations on threads that need to be performed for effective support of parallel programs on EARTH. More information is needed so that designers can have a target for implementation. Chapter 5 fills in important details about these operations, including how data and threads are addressed globally, and provides a complete specification of EARTH operations. This is called an *EARTH Virtual Machine* because it is still more abstract than a complete instruction set. (A complete instruction set for EARTH would be inappropriate, as that would tie EARTH to a particular processor.) An alternate virtual machine (addressing scheme and set of operations) is proposed; both virtual machines are consistent with the specifications of the PXM.

Chapter 6 defines the Threaded-C language. Threaded-C extends ANSI-standard C with functions corresponding to the EARTH operations defined in Chapter 5. This allows programmers to write applications for EARTH in a high-level language, though one which is explicitly threaded by the programmer. The language is illustrated through programming examples.

The next two chapters present several implementations representing various points along the evolutionary path listed in Section 1.3. Chapter 7 represents the first step, the emulation of the EARTH model on an off-the-shelf multiprocessor without specialized hardware support for EARTH. Two implementations based on the GMD MANNA machine are described, one in which each node has two processors, and one in which each node has a single processor. The chapter describes the runtime system (software which supports the EARTH operations) and the compiler (which compiles Threaded-C code with the help of an off-the-shelf compiler for the MANNA's processor).

Experiments on a 20-node MANNA show that the EARTH operations operate very quickly, and latency and bandwidth measurements are far better than commercial parallel machines using comparable technology. Experiments with eleven benchmark applications show that the multithreading support provided by EARTH imposes only moderate overhead costs on the code, as measured by comparing multithreaded code running on one node with sequential code. Furthermore, most of these programs have good (nearly linear) speedups up to 20 nodes, showing that the EARTH implementation on MANNA is good for many applications, even though it

is only the first evolutionary step.

SEMi, a simulator for the MANNA with high timing accuracy, is also presented in this chapter. SEMi is used to extend the speedup curves of selected benchmarks to 120 nodes, six times what is available on real machines. The accuracy of SEMi gives confidence that these results are a reasonable estimate of the performance of these benchmarks on a large MANNA, if one were constructed.

Chapter 8 considers the remaining evolutionary steps of Section 1.3. The second and third steps involve developing special hardware to support EARTH operations more efficiently, either as a separate module or as a component added to a microprocessor core at the chip level. Complete interface specifications for this component are given, along with a high-level block design. The SEMi simulator is augmented to simulate this hardware, and the selected benchmarks are run on the new system and compared to the original (software-only) configuration. The results show that each evolutionary step leads to substantial speed improvements in individual EARTH operations, reductions in latencies between threads and in multithreading overheads, and improvements in speedups for all applications. With hardware support, many of the applications tested have high speedups even on 120 nodes. The end of the chapter discusses the possibilities for the final evolutionary step (to full-custom hardware).

The dissertation then reviews other multithreading systems (excluding those already covered in Chapter 3), including both architectures and software-based systems (Chapter 9), and presents our final conclusions (Chapter 10).

There are several appendices. Appendix A surveys other studies of parallelism related to Chapter 2. Appendix B is a complete list of the data types and operators of the Threaded-C language. Appendix C recapitulates some of the experimental results of Chapters 7 and 8 to allow a direct comparison of different implementations of EARTH on MANNA. Appendix D presents the results of an auxiliary study which uses the SEMi simulator to modify the performance parameters of the MANNA hardware to be more consistent with current off-the-shelf systems. The experiments from Chapters 7 and 8 are repeated; the results show that the efficiency of the EARTH model and the benefits of hardware support are applicable to contemporary systems as well as the older MANNA hardware.

Chapter 2

Parallelism in Computer Programs

In the introduction, it was shown that exploiting parallelism is a worthwhile pursuit because there is a need for computers of greater and greater power. However, before attempting to build a parallel processor to run a class of applications, one should analyze the applications to know both how much parallelism is intrinsic to each application and what architectural properties are necessary to achieve this parallelism. This section reviews a study performed by the author for this purpose, based on real applications. (A survey of previous studies is provided in Appendix A.) The results show that most applications have enough inherent parallelism to keep at least a moderately-sized parallel computer usefully busy.

This section is divided into four parts. The first part describes the SITA tool, developed by the author to study program parallelism. An abstract model of execution on parallel machines, called the Dynamic Control Dependence Tree, is also presented. The next two parts present the experimental results obtained with this tool. The section closes with a discussion of the conclusions that can be drawn from the SITA studies.

2.1 The SITA Tool

The current study is based on an analysis tool developed at McGill, called SITA (Sequential Instruction Trace Analyzer) [116, 119, 120]. Like some of the previous studies, the SITA tool analyzes traces of machine-language instructions generated from executed code, rather than analyzing the high-level source code. Figure 2.1 shows conceptually how SITA works.

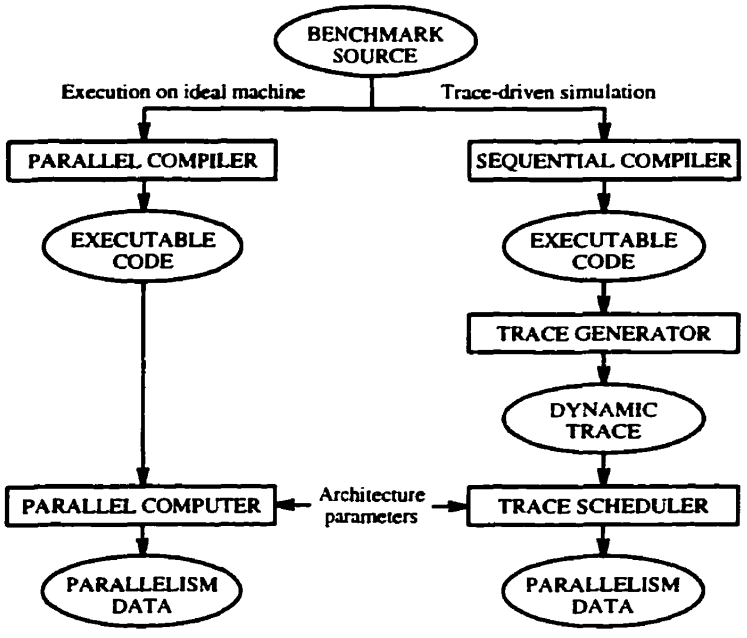


Figure 2.1: Trace Simulation Methodology

In the flowchart, the left fork represents the compilation and execution of a program on an ideal machine with a given set of architectural characteristics. Lacking such a machine, one can model it using trace simulation, as shown in the right fork. First, a benchmark program is compiled into an executable file using a conventional sequential compiler. Then the executable code is run through a *trace generator*, which produces a program trace. This trace consists of a stream of operations representing the actual sequence of instructions executed (*not* the static object code). Finally, the *trace scheduler*, modeling a machine with the same architectural features as the "ideal machine" in the left path, determines the time at which each individual instruction appearing in the trace would be executed on that ideal machine. The trace scheduler produces statistics on parallelism which should predict the behavior of the code on the ideal machine.

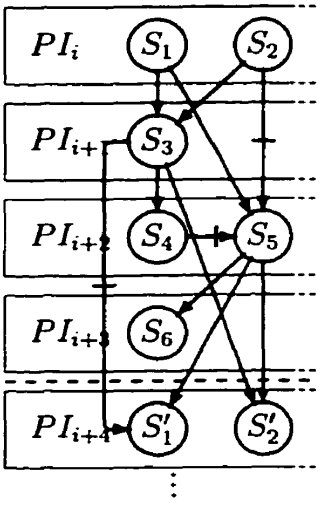
Figure 2.2 illustrates the operation of the trace generator and scheduler. Part (a) shows a fragment of Sparc executable code, a simple loop. The trace generator produces a dynamic stream of instructions as shown in part (b). For each executed instruction, the trace gives the opcode, PC address, the load or store address (if any), and the outcome of the conditional branch (if any). The trace scheduler reads

```
S1:  ld [%r2+%r1],%f0
S2:  ld [%r3+%r1],%f1
S3:  fmuls %f0,%f1,%f2
S4:  st %f2,[%r4+%r1]
S5:  subcc %r1,4,%r1
S6:  bg S1
S7:  nop
```

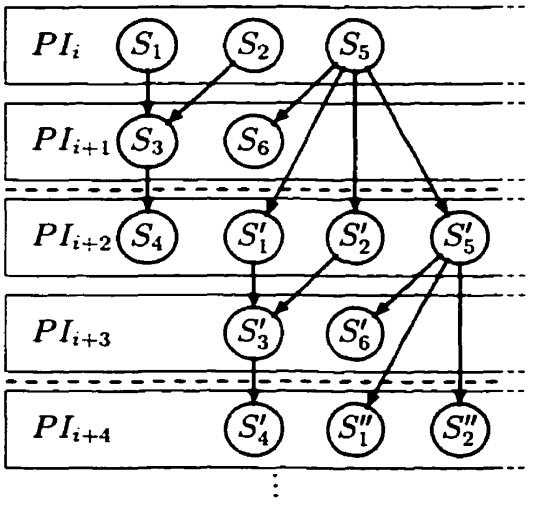
a) Executable code

```
S1:   ld [%r2+%r1],%f0     308
S2:   ld [%r3+%r1],%f1     216
S3:   fmuls %f0,%f1,%f2
S4:   st %f2,[%r4+%r1]     440
S5:   subcc %r1,4,%r1
S6:   bg S1                Taken
S7:   nop
S'1:  ld [%r2+%r1],%f0     304
S'2:  ld [%r3+%r1],%f1     212
S'3:  fmuls %f0,%f1,%f2
S'4:  st %f2,[%r4+%r1]     436
S'5:  subcc %r1,4,%r1
S'6:  bg S1                Taken
S'7:  nop
S''1: ld [%r2+%r1],%f0     300
      ⋮
```

b) Dynamic trace



c) Scheduling (all dependences)



d) Scheduling (no false dependences)

Figure 2.2: Packing Parallel Instructions: An Example

the operations from the stream and packs them into *parallel instructions* (PI). As the scheduler reads each operation in the trace, it inserts the operation into the earliest PI possible, while simultaneously respecting the dependences between that operation and all previous operations. The following types of dependences between operation S_j, inserted into PI_i, and a later operation S_k may exist:

- *Flow dependence:* if S_k reads a storage location (register or memory cell) which was most recently written by S_j, then S_k can be scheduled no earlier than PI_{i+1}.

- *Anti dependence:* if S_k writes a storage location which was most recently read by S_j, then S_k can be scheduled no earlier than PI_i. (It is assumed that the write and read can occur simultaneously, and S_j will read the proper value.)

- *Output dependence:* if S_k writes a storage location which was most recently written by S_j, then S_k can be scheduled no earlier than PI_{i+1} (assuming that the storage location is read by a later instruction).

- *Control dependence:* if S_j is the most recent conditional branch in the trace whose outcome must be decided before it is known whether or not S_k will be executed, then S_k can be scheduled no earlier than PI_{i+1}.

Figure 2.2 (c) shows how SITA would pack these operations into PIs if all four dependence types listed above were obeyed. (It is assumed that all memory accesses take one cycle). Plain arcs are drawn to show the flow dependences between operations. The arcs with small marks (e.g., from S_4 to S_5) indicate anti dependences. The dashed line represents a barrier caused by the conditional branch at S_6, which may prevent future operations from being scheduled before that barrier.[1] Parallelism is defined as the total number of sequential operations divided by the total number of PIs required by the scheduler.

The SITA tool is extremely flexible in the features it can model. Some features relax the dependence constraints, increasing the opportunities for exploiting parallelism in the program. Other features tighten the resource limitations, forcing some instructions whose dependences have been satisfied to wait for resources to become free. The following subsections describe some of the architectural features which SITA can model:

2.1.1 Memory Renaming and Disambiguation

A compiler can't always tell whether or not two memory accesses refer to the same location. A conservative analysis would assume that any two memory references

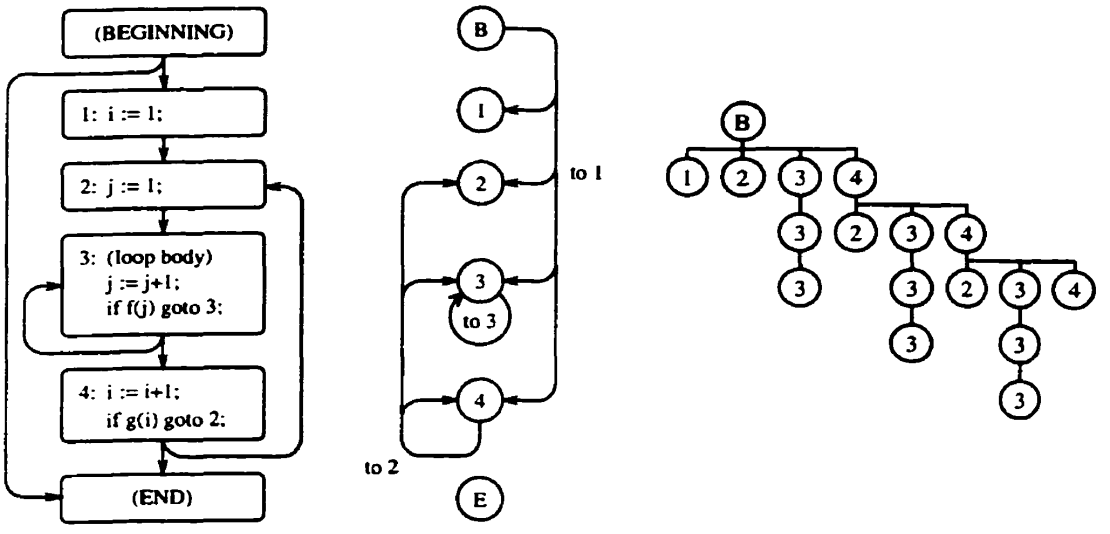
[1] For operations S_1–S_3 and S_5, there are also output dependences between corresponding operations in different iterations; these have been left out of Figure 2.2 for clarity.

could refer to the same memory location, in which case a dependence would exist between them. Thus, for instance, a conservative scheduler would have to assume that a flow dependence might exist between S_4 in one iteration and S_1 in the next iteration, making it harder to overlap separate iterations of the loop. However, SITA can check the addresses in the trace to determine whether two memory references really conflict. This models the potential effects of perfect compiler alias-analysis.

A false dependence (anti or output) exists between two operations when one must follow the other, *not* because the latter requires data produced by the former, but merely because the latter needs to reuse a storage location (register or memory cell) used by the former. An example of this in the code sample is the use of register %r1 for the loop index, which cannot be updated (by S_5) before being used to construct a memory address (by S_4). Thus, overlapping the iterations of the loop body, as in software pipelining, is impossible. False dependences also frequently occur in main memory, either through the update of data structures (such as arrays) in place, or the sharing of the stack by different procedures at the same call depth.

The inhibiting effects of false dependences can be eliminated by ensuring that each register or memory location is written only once. In the CPU, false dependences can be reduced by creating extra physical registers, and dynamically mapping registers in the instruction stream to these physical registers. This technique, called *register renaming*, is used in many state-of-the-art RISC processors to boost parallelism. False dependences in the memory can, in principle, be eliminated by only assigning to each variable once, as is done in programming languages such as Sisal [31]. False dependences in the stack can be eliminated, by organizing the memory frames for procedure invocation in a tree-like structure, as proposed in several dataflow and multithreaded architecture models [21, 92, 99].

Infinite renaming, in which a register or memory location can be renamed any number of times, is equivalent to ignoring all false dependences between objects of a particular type. For instance, if register renaming is applied to the trace in Figure 2.2(b), then S_5 can be executed in parallel with S_1 and S_2 of the same iteration, as the anti dependence with S_4 no longer exists. This moves operation S_6 up as well, so the conditional-branch barrier has moved up by 2 PIs. If perfect disambiguation is also used, then the iteration issue rate increases to once every other cycle, raising parallelism to 3, as shown in Figure 2.2(d).



a) CFG b) CDG c) DCDT

Figure 2.3: Control Dependence

2.1.2 Control Barrier Elimination

In many of the previous parallelism experiments, the schedulers made pessimistic assumptions about the effects of conditional branches on the instructions that followed. The most conservative assumption is that if a conditional branch instruction is placed in PI_i, then all instructions appearing later in the stream can be scheduled no earlier than PI_{i+1}. However, this restriction goes beyond the definition of control dependence in Section 2.1, because not all future instructions are truly control dependent on that branch.

This point is illustrated with the code fragment shown in Figure 2.3, in which a small "loop body" does some calculation inside a simple two-dimensional loop. Part (a) shows the loop drawn as a *Control Flow Graph* (CFG). The code is partitioned into *basic blocks*, each block being a maximal set of contiguous instructions which is only entered at the beginning and exited at the end. (Each set is maximal in the sense that if either the instruction following or preceding the block is added to the set, it no longer satisfies the definition.) The basic blocks are connected by arcs, which connect a basic block to all of its possible successors.

Only some basic blocks end with conditional branches, and each of those conditional branches only affects certain basic blocks. For instance, both block 3, the

body of the inner loop, and block 4, the end of the body of the outer loop, end with conditional branches. However, block 4 is not control-dependent on block 3; block 4 will eventually execute once at the end of the loop no matter how many times the inner loop executes. The CFG can be converted to a *Control Dependence Graph* (CDG) [32] which conveys this information, as shown in part (b). In this graph, the arcs labeled "to 2" whose source is block 4 (for simplicity, the arcs are shown as a single arc which forks) indicate that if the conditional branch at the end of block 4 is *taken* (i.e., to block 2), then one more instance of each of blocks 2, 3, and 4 is guaranteed.

What this means for code execution is shown in part (c). This part shows a *Dynamic Control Dependence Tree* (DCDT) [119], which is produced by dynamically unrolling the CDG according to the outcome of each conditional branch. The tree shows the control dependences between particular instances of basic block. A block is only control-dependent (according to the definition in Section 2.1) on its parent in the DCDT. Therefore, two blocks that do not have a direct ancestor relationship are control-independent, and could run in parallel provided there were no other dependences (flow, anti or output) to impede them. For instance, what Figure 2.3(c) shows is that while instances of the inner loop would have to be executed sequentially, the outer loops could be run in parallel. (This is assuming there are no other dependences. Given the fact that the loop counters i and j are shared and updated in place, a machine would have to use register or memory renaming, as described in Section 2.1.1, in order to exploit this parallelism.)

SITA supports the following options (listed in order of increasing power):

- The most pessimistic assumption is that *all* future instructions are control-dependent on a given conditional branch.
- If *procedure separation* is used, the barrier created by a conditional branch only affects other instructions in the same procedure, and those in procedures called by that procedure.
- *Control-dependence analysis* uses a CDG, generated by SITA, to limit the effects of conditional branches to those instructions that are truely control-dependent on them.
- With the *speculative execution* option, SITA can model the behavior of a

machine which tries to execute instructions before the conditional branches on which they are control-dependent have been executed.

- The *oracle* [90] makes the most optimistic assumptions by ignoring control dependences entirely, and provides an upper bound on parallelism by measuring the effects of flow dependences only.

2.1.3 Finite Resources

SITA normally optimistically assumes resources are infinite and all operations are uniformly fast. However, the user can specify constraints on the window size, latencies, and number of processors.

Normally, SITA assumes that an operation from anywhere in the trace can be placed in the earliest PI possible, subject to data and control dependences. In theory, the last operation in the sequential trace could be packed into the first PI. However, the user can specify a limit on how far apart two operations can be packed. This model is based on the assumption that the parallel machine can only look so far ahead of the program counter when looking for operations that are ready to be executed, much as today's out-of-order superscalar processors do. With a limited window size, SITA conceptually keeps future operations from the trace stream in the window and packs them into PIs from the window. When the trace stream fills the window, the analyzer must "issue" the lowest-numbered unissued PI, thereby making it unavailable for further packing, and remove the operations in that PI from the window.

SITA normally assumes that every operation takes 1 cycle. SITA can be configured to give higher latencies to certain operations, such as floating point operations or remote memory accesses. It is difficult to obtain accurate modeling for the latter from a sequential trace, because the sequential compiler does not partition the data, there being only one processor. Thus, the sequential trace provides no notion of "local" and "remote" processors. SITA, therefore, models remote accesses statistically; it designates, at run time, a randomly-selected number of memory accesses, giving each a user-specified latency. The ratio of remote accesses to total accesses is also set by the user.[2]

[2]In principle, data partitioning could be done by hand for each benchmark. However, the pattern of memory use on a sequential processor may be radically different from the pattern of

Source	Program	Description	Test Case	Ops ($\times 10^6$)	% of ops.		
					FP	Ld.	St.
DLX	Tex	Text formatting	draft (11 p.)	109	.04	15	8.0
Industry	Speech	Speech recognit.	recognize "he"	551	4.5	14	2.8
SPEC89 test suite	Espresso	Bool. minimization	bca.in	469	<.01	23	2.5
	Eqntott	Truth-table gen.	int_pri_3.eqn	1,770	0	33	0.7
	Fpppp	Quantum chem.	NATOMS=4	277	19	43	10
	Tomcatv	Data-parallel grid	N=257	3,018	17	48	13
	Doduc	Simulation	small	522	14	36	10

Table 2.1: Benchmarks Used in the Study

Finally, one can limit the number of processors that SITA models. Normally, the trace scheduler places each trace instruction in the earliest PI possible, consistent with all dependences with earlier instructions. The user can specify a maximum width to each PI, thus modeling a finite number of processors. If, after checking for dependences, SITA finds that the earliest legal PI is full, it must place that instruction in a later PI, the earliest one which is not full.

2.2 Experiments with SITA

Seven benchmarks were used in this study. They are presented in Table 2.1. The first four are irregular applications written in C. The others are numerical, floating-point-intensive applications written in Fortran.

2.2.1 Control Dependence Experiments

The first set of experiments measured the effects of control flow on parallelism. Each benchmark was run under four machine models. All four models have infinite renaming of registers and memory, perfect memory disambiguation, one-cycle operations, and infinite resources. The only parameter that is varied among them is the sensitivity to control dependences. The experiments explored the various options discussed in Section 2.1.2, except speculative execution, which is covered in a later

memory use on a parallel processor. Thus, for all but highly-regular benchmarks, a user-specified partition is not likely to yield a more accurate predictor of remote memory accesses than SITA's probabilistic approach.

Benchmark	Omniscient Oracle	Fine Dataflow	Coarse Dataflow	Smart Superscalar
Tex	192	5.31	3.04	2.08
Speech	8,105	45.4	6.30	1.80
Espresso	1,224	2.81	1.78	1.47
Eqntott	43,298	2.24	1.66	1.46
Fpppp	4,978	1,179	52.7	30.1
Tomcatv	8,417	6,014	19.2	19.1
Doduc	615	566	32.8	4.92

Table 2.2: Effects of Control Dependence

section.

- The *Smart Superscalar* model assumes that any future instructions must be control-dependent on a current conditional branch, and thus does not allow that code to run before the current branch has been executed.
- The *Coarse Dataflow* model performs procedure separation; thus, conditional branches only affects future code within the same procedure call.
- The *Fine Dataflow* performs full control-dependence analysis, as described in Section 2.1.2.[3]
- The *Omniscient Oracle* model pays no attention to control dependences when scheduling code.

The results of these experiments are shown in Table 2.2. All show very high levels of parallelism under the Oracle model. However, for lesser models, there are significant differences between the performances of the non-numerical applications (the first four) and the performances of the numerical applications. The non-numerical applications show a severe loss of parallelism when all true control dependences are obeyed, especially Tex and Eqntott. The numerical codes, on the other hand, continue to have high levels of parallelism with the Fine Dataflow model. This is because their control structures tend to be very simple and not dependent on the data computed, e.g., a loop with fixed bounds rather than a loop whose continuation depends on data computed within the loop body.

[3]This is similar to the CD-MF model used by Lam and Wilson [77].

Bench-mark	Omni. Ora.	Tree Ora.	% of OO	Linear Ora.	% of OO	Frugal Ora.	% of OO	Smart Super.	Stupid Super.	% of Smart
Tex	192	75.8	39.4	167	86.9	71.5	37.2	2.08	1.66	80.8
Speech	8,105	136	1.68	57.1	0.70	53.6	0.66	1.80	1.62	89.9
Espresso	1,224	126	10.3	906	74.0	124	10.1	1.47	1.37	93.2
Eqntott	43,298	1,742	4.02	1,314	3.03	1,314	3.03	1.46	1.43	98.2
Fpppp	4,978	4,978	100	71.0	1.43	70.9	1.42	30.1	2.88	9.56
Tomcatv	8,417	457	5.43	155	1.84	155	1.84	19.1	2.74	14.3
Doduc	615	614	99.7	25.0	4.06	25.0	4.06	4.92	2.33	47.4

Table 2.3: The Effects of Frugal Use of Memory

The numerical applications suffer their greatest geometric loss when going from the Fine Dataflow to the Coarse Dataflow model. This is mostly because they contain two-dimensional loops. As the DCDT in Figure 2.3(c) shows, control-dependence analysis often allows outer loops to run concurrently. Without this analysis, all iterations must run in sequence. Control-dependence analysis is less important for non-numerical applications; only Speech shows a loss greater than 50%.

2.2.2 Register/Memory Renaming Experiments

The second set of experiments measured how parallelism is affected by the reuse of registers and memory. The SITA tool allows separate control over the renaming of registers and of stack and heap objects.[4] The first three experiments were with "frugal oracles." These oracles are just like the Omniscient Oracle, but they don't have full memory-renaming capability. The *Tree Oracle* allows renaming of stack variables, to measure the limits of parallelism exploitable by a machine using a tree of stacks or some equivalent implementation, but does not allow renaming in the heap. The *Linear Oracle* retains the linear stack model, by not allowing stack elements to be renamed, but allows renaming in the heap. The *Frugal Oracle* has no memory renaming. The results of these experiments, shown in Table 2.3, demonstrate the importance of avoiding false dependences between operations that reuse memory objects.

[4] In this study, "heap" includes both dynamically-allocated and global (static) variables.

One final model, the *Stupid Superscalar*, tests the benefits of renaming and disambiguation at the "low" end of the scale. The Stupid Superscalar is like the Smart Superscalar, but has neither memory disambiguation nor register/memory renaming.[5] Results for this model are given in the rightmost two columns of Table 2.3.

Both superscalar models have the property that a conditional branch affects the scheduling of all future instructions. Thus, these machines can generally only find parallelism within basic blocks. Indeed, the rightmost column of Table 2.3, which gives the performance of the Stupid Superscalar relative to the Smart Superscalar, shows that for irregular code, renaming and disambiguation by themselves produce little benefit. Only Fpppp shows a big gain when renaming and disambiguation are added, probably due to its large basic blocks and regular code structure. For codes with frequent branches, we must go to more aggressive models, such as the Fine Dataflow model.

2.2.3 Finite Window Experiments

The next set of experiments demonstrate that restricting the choice of instructions to execute to a small "window" of instructions near a single program counter can significantly reduce parallelism. Two experiments were run, each with an Omniscient Oracle whose window has been restricted as described in Section 2.1.3. Window sizes of 64 and 2,048 were used. The results, shown in Table 2.4, show that even with a 2K window, there is a significant loss of parallelism.

Additional experiments measured the effects of finite window sizes for less-capable machine models [117, 116]. In most cases, restricting the window size down to 64 ops had little effect on such machines, because their other limitations already restricted parallelism to a high degree.

2.3 Speculative Execution and SITA

The data in Table 2.2 shows that, at least for non-numerical applications, there is a large gap between what could be achieved by an ideal machine with perfect

[5]The Stupid Superscalar is similar to Wall's "Stupid" model [128], except that Wall's model was limited to 64 processors, and could only schedule operations from within a window of 2,048 instructions. These limitations, however, had almost no additional impact on the Stupid model, since register and memory reuse, control dependences, and lack of memory disambiguation generally removed almost all available parallelism except that within basic blocks.

Benchmark	Omni. Oracle	2K ops Window	% of OO	64 ops Window	% of OO
Tex	192	62.1	32.3	12.2	6.37
Speech	8,105	90.5	1.12	12.4	.154
Espresso	1,224	35.1	2.86	12.9	1.05
Eqntott	43,298	141	.325	19.6	.045
Fpppp	4,978	75.6	1.52	15.8	.317
Tomcatv	8,417	86.4	1.03	22.6	.268
Doduc	615	104	16.9	11.4	1.85

Table 2.4: The Effects of Finite Window Size

knowledge of every branch outcome, and what could be achieved by a machine which is affected by control dependences. This observation and others have generated increasing interest in getting more parallelism from programs through *speculative execution.* This means executing the code at one or more destinations of a branch before the branch outcome is known.

This has primarily been used to try to prevent branches from disrupting long execution pipelines, as in superscalar machines. For instance, some machines use the past history of branches to predict the most likely destination and prefetch along that path. Various *branch prediction* techniques have been developed [78, 97, 105]. Some machines speculate further ahead; one superscalar uses *boosting* [106] to speculate past a branch many instructions before it is resolved, using "shadow registers" to maintain two program states, which are made consistent once the branch has been resolved.

With this in mind, we developed a model of speculative execution appropriate for highly-parallel computers, and a new model of branch prediction appropriate for this new model. This led to a new set of experiments with the goal of understanding the interaction of speculative execution and branch prediction, how they affect program parallelism, and what kinds of speculative execution and which branch prediction strategies lead to the highest potential amounts of parallelism.

2.3.1 Adding Speculative Execution to the DCDT

Ideal parallel execution of programs which use control-dependence analysis to maximize parallelism are modeled using the Dynamic Control Dependence Tree, described in Section 2.1.2. The CDG and the corresponding DCDT can be used to create a model of parallel execution in which the effects of control dependence are minimized. In the earlier section, as a CDG was "unrolled" into a DCDT, vertices in the CDG were successively control-enabled, and each new "enabling" of a node corresponded to a node in the DCDT. Time can be added to this model. We can say that a node begins execution as soon as it is control-enabled, and that its children (if any) are control-enabled some time later, after its branch or jump has been completed. We say that an instance of a block is *resolved* at the time that its branch or jump has been computed. (This could occur before all instructions in that block have completed, if the instructions are reordered so that the branch predicate is computed before the end of the block.)

The Fine Dataflow model, described in Section 2.2.1, is the base case of a machine with control-dependence analysis, but no speculation. For this reason, it can also be called a *0-Level Speculator* [119]. Suppose a machine begins running according to the model, and starts executing blocks 2, 3, and 4 at the same time. If the branch in block 3 is taken (destination is 3), then another instance of block 3 will be initiated. A 0-Level Speculator would have to wait until the first instance of block 3 is resolved before starting another iteration.

However, a speculative machine could assume that the branch at block 3 will be taken, and start executing the next iteration of the inner loop before the *first* branch is resolved. Simultaneously, it may start new occurrences of blocks 2, 3, and 4 before the first occurrence of block 4 has finished. This would be an example of *1-Level Speculation.* In general, an *n-Level Speculator* speculatively executes blocks up to n conditional branches past the last resolved conditional branch, i.e., from each resolved branch in the DCDT, it generates the descendants of that branch in the DCDT, down $n + 1$ levels (the first level requires no speculation) and begins executing all blocks in parallel.

There are two general techniques for predicting the outcome of a branch: static and dynamic. Static prediction assigns a most likely outcome to each branch or jump in the object code, and this prediction is made every time that particular instruction is executed. This prediction can be based on the direction of a branch

(e.g., backwards branches are always taken and forwards branches are never taken) or on statistics gathers from a sample run of the program.

A dynamic predictor guesses the outcome of a branch or jump by looking at the outcomes of previous occurrences of that instruction, and possible of other related branches. It is hoped that the predictor can learn from and adapt to changing branching patterns [78]. The conventional approaches used for sequential processors were shown to be inappropriate for parallel machines, and a new method of dynamic prediction, based on the DCDT, was developed [118, 119]. However, experiments then showed that in almost all cases, static and dynamic prediction produced almost the same amount of parallelism, so this section only reports on experiments using static prediction.

2.3.2 SITA Experiments on Speculation

For the experiments, only the first five of the seven benchmarks in Table 2.1 were used. The last two, Doduc and Tomcatv, show almost no difference between the 0-Level Speculator (Fine Dataflow model) and the Omniscient Oracle. In both cases, there are a lot of loops in which the loop predicate is not data-dependent on the body of the loop (e.g., iteration across a fixed range). In such a case, a 0-Level Speculator could iterate and test the loop counter as soon as the loop body begins, so one loop iteration could begin every few cycles. Speculating past the test would save only a few cycles, because the initiation of loop iterations would still be constrained by the data-dependences involved in the incrementing of the loop counter. Since the 0-Level Speculator and Omniscient Oracle represent the lower and upper bounds of what could be achieved with speculation, speculation has little potential benefit for these two applications.

Each of the remaining five benchmarks were run through SITA under an aggressive speculative execution model (see Table 2.5. In this model, the machine speculates an infinite number of levels past each resolved branch, but only executes along the branch path chosen by the branch predictor. Like the original Fine Dataflow machine and Omniscient Oracle, the speculators have unlimited processors, scheduling windows, and register/memory renaming. The results show that speculative execution can produce significant increases in parallelism. Improvements were moderate in only two cases: Tex, which does not have much parallelism in any case,

	0-Level Speculator	Infinite Speculation	Infinite/ 0-level	Omniscient Oracle	Oracle/ Infinite
Tex	5.31	36.8	6.93	192	5.14
Speech	45.4	1,192	26.3	8,105	6.80
Espresso	2.81	53.0	18.9	1,224	23.1
Eqntott	2.24	2,625	1,172	43,298	16.5
Fpppp	1,179	4,166	3.53	4,978	1.19

Table 2.5: Speculation Success

and Fpppp, for which the 0-Level Speculator already does so well.

These results assumed that SITA would speculate past an infinite number of unresolved branches. Since that would be impossible in practice, a new group of experiments was performed to measure the benefits of speculation when limited to a *finite* number of levels. Figure 2.4 shows, for each benchmark, the parallelism result obtained by SITA when speculation depth was limited to the number on the horizontal axis. Three of the five benchmarks achieved the same level of parallelism with 16 levels of speculation as with an infinite depth, so no experiments beyond 16 levels were performed. Beyond 16 levels, Espresso showed only minuscule improvements. Only Eqntott required more than 16 levels to get close to the performance of the infinite-level model. In fact, most of the benchmarks achieved significant benefits even from only 2 or 4 levels of speculation.

2.4 Discussion

The SITA studies have shown the parallelism theoretically attainable for various applications under various abstract architectural models. How applicable are these studies to future research in parallel machines? What conclusions can be drawn from these results?

One of the main goals of the SITA studies was to see how much parallelism could potentially be exploited in various applications. The first question to ask is: how do the results produced by SITA apply to parallel machines? After all, SITA calculates parallelism by scheduling the instructions from a trace generated by the execution of a sequential program. The control structures might be significantly different in a parallel version of the program.

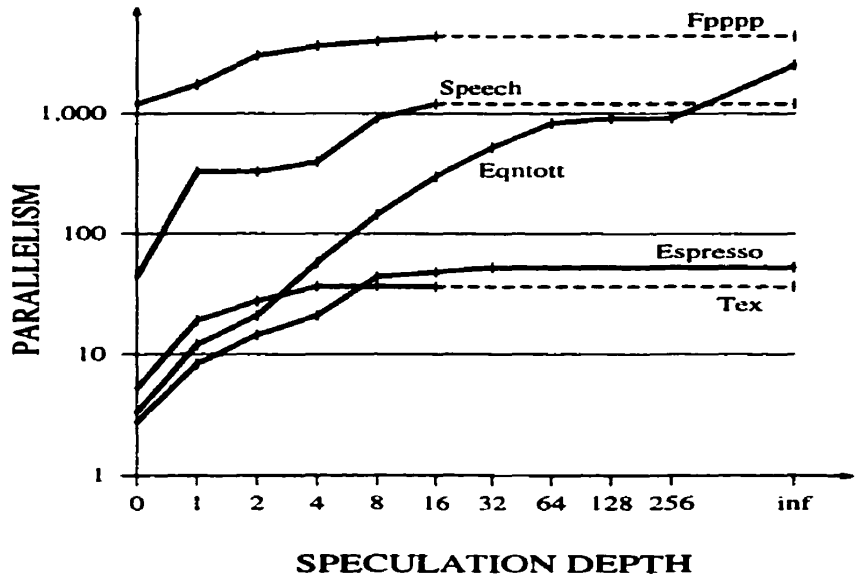


Figure 2.4: Finite Speculation Depth

However, SITA allows one to remove many of the artifacts of programming in a sequential language. All models reported in this section, except the Stupid Superscalar model, assume adequate renaming mechanisms, single-assignment use of memory, and effective alias analysis, and thus ignore the effects of false dependences. All models, except those covered in Section 2.2.3, allow code from different parts of the program, no matter how far "apart" in the sequential sense, to execute at the same time. As a result, models such as the Fine Dataflow machine were able to achieve impressive amounts of parallelism for some of the applications.

But this reveals a limitation of these studies. High parallelism was possible under the numerical applications because the control structures are simple, and they are not likely to be substantially different in a parallel machine. The non-numerical applications, on the other hand, are more problematic. We found disappointingly low amounts of parallelism for some of the irregular programs under the Fine Dataflow model. Other studies have seen similar results.

However, it is possible that some of the control dependences which throttled the performance of the Fine Dataflow model on these applications may also be simply due to sequential programming. Some applications, like Tex, intuitively seem to have much more potential parallelism than found by our Fine Dataflow model (5.31) or Lam and Wilson's similar Multithreaded model with control-dependence analysis

(6.18) [77]. This program performs the layout of an 11-page document, and it would seem that each paragraph could be formatted independently, with its final vertical position determined at the end of the program. Obviously, there are control and data dependences which prevent this from happening, but many of these may be due to the programmer's assumption of a sequential execution model, not to inherent sequentiality in the application.

This has important implications for future directions in architecture. It is sometimes believed that to exploit maximum parallelism, we only need to rewrite existing algorithms in a better language, such as a functional language, so that an efficient compiler can extract all the available parallelism without being burdened by such sequential artifacts as memory reuse. But the Fine Dataflow model gets poor results, even though it eliminates false data dependences, and represents what could be achieved by an efficient parallel running well-compiled programs.

This would suggest that in many cases, large-scale parallelism will not be achieved merely by rewriting existing imperative-language programs in different languages. They must be redesigned completely, starting at the algorithmic level. Whether this is best done by having the programmer write programs with an explicit parallel machine model in mind, or use a more abstract programming paradigm, such as logic programming, is an interesting and ongoing area of research.

In some cases, the Omniscient Oracle may give a hint of what could be achieved with a suitably-programmed parallel machine. Our oracle models, like oracles in other studies [90], ignored control dependences entirely. While the results obtained from the oracle are an upper bound, and may be unrealistically high, we believe that suitably-designed programs with appropriate parallel algorithms can get parallelism results which are much closer to the oracle. As an example, consider the Eqntott program. This program is dominated by calls to a quicksort routine, which is inherently sequential. The control dependences force the swapping of elements on either side of the pivot element to occur sequentially. Removing these control dependences, as the oracle does, permits most of these swaps to occur in parallel. The only serializing is between different levels of recursion in quicksort; this serializing is caused by the *flow* dependences. But this behavior represents what would happen under an efficient *parallel* sorting algorithm, which would allow many swaps on the same level of recursion to occur simultaneously.

If such changes are impractical, then one should consider some sort of speculation

for programs with irregular control flow. SITA showed that this can significantly improve program performance.

One other important result from the SITA studies was the importance of executing code from different sections of the program at the same time. Performance was severely degraded when instructions for parallel execution were limited to a small window near a single "program counter." This is because the parallelism within a basic block is limited (as shown by the performance of models such as the Smart Superscalar, which only has good results for programs with unusually large basic blocks such as Fpppp). This does not bode well for machines with a single thread of control, such as the superscalar and VLIW machines, and suggests a reason for their poor IPC ratings as reported in Section 1.1.

To come closer to the performance levels suggested by SITA, architects will need to consider execution models which permit greater flexibility in the scheduling of parallel instructions. The next section describe two such models, the *dataflow* model and the *multithreading* model.

Chapter 3

Predecessors of EARTH

Section 1.2 identified several key issues which designers of multiprocessors need to face, such as latency, synchronization, and programmability. It was argued in previous chapters that SIMD and MIMD approaches to parallelism do not adequately solve all of the major problems. It is our belief that *multithreaded* architectures derived from *dataflow* principles have the potential for addressing the problems inherent in these other machines, thereby fulfilling the promises of efficient multiprocessing.

This chapter traces the evolution of ideas leading to the EARTH system. The EARTH program execution model, presented in the next chapter, can trace its roots back to the dataflow machines designed at MIT in the 70s and 80s. Therefore, Section 3.1 surveys dataflow concepts and traces the evolution of dataflow machines, focusing on those that can be considered ancestors of EARTH. Experience with real dataflow machines have shown that certain properties make efficient implementation difficult, as shown at the end of Section 3.1. Recognition of these shortcomings have led to attempts to combine dataflow with traditional von Neumann processing. in a way which allows characteristics of each model to compensate for weaknesses in the other. This is covered in Section 3.2, which includes a survey of multithreaded machines most relevant to EARTH.

Multithreaded machines based on conventional von Neumann principles, rather than dataflow, have been investigated for a long time and are still an ongoing area of research. Some of these machines are surveyed in Chapter 9.

3.1 Dataflow Machines

Dataflow machines [41, 47] are a radical break from traditional von Neumann computers. The latter have a single program counter which determines which instruction to execute next. This imposes a total order on the instructions. The order in which two instructions are executed is fixed, even if there are no dependences between them and a tool like SITA (see Section 2.1) would say that they could execute in either order or at the same time. In the dataflow model, on the other hand, there is only a *partial* order between instructions. The fundamental principle of dataflow is that *any instruction can be executed as long as its operands are present.*

Programs are represented abstractly by *dataflow graphs*, which capture this concept. Dataflow graphs are covered in the first subsection. The remainder of this section describes various designs for machines that execute programs based on dataflow graphs.

Traditionally, the two major classes of dataflow machines have been *static* and *dynamic*. These are covered in separate sections. Both of these classes have specific weaknesses, and attempts to eliminate these deficiencies have led to a convergence of the two classes. They are identified in this paper as *semi-dynamic* and are covered in the last section. One of the static designs, the *argument-flow* machine, is described in detail (in Section 3.1.2). For the other machines, it is only necessary to point out how they differ from the argument-flow machine.

3.1.1 Dataflow Graphs

A dataflow graph is a directed graph. The vertices are called *actors* and the edges are called *arcs*. An arc from node X to node Y is called an *output arc* of node X and an *input arc* of node Y. Some special edges do not have nodes at both ends. These are called input arcs, or output arcs, of the *graph* if the beginning of the arcs, or the end of the arcs, respectively, are not connected to any actors.

Figure 3.1 shows a dataflow graph which computes the product of the complex numbers $(a + bi)$ and $(c + di)$. There are four input arcs (the real and imaginary components of the two numbers) and two output arcs. A *token*, drawn as a black dot in the figure, represents one unit of data. The figure shows a complete set of tokens at the input arcs. The "dataflow" is represented by tokens flowing through the graph.

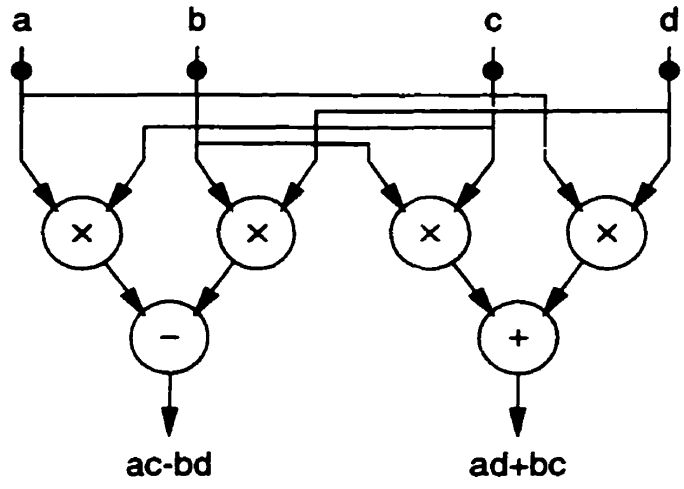


Figure 3.1: Dataflow Graph for Complex Multiply

The tokens flow according to set rules. Arcs simply propagate tokens from the output of the actor at the source end of the arc to the actor at the destination end. If an arc forks, the token is replicated and propagated to each destination. An actor moves tokens along by "consuming" tokens appearing at its input arcs and producing tokens on its output arcs. This is called "firing" an actor. Each actor has a set of conditions indicating when and how it may fire, called the *firing rules.* These rules include the requirement that there be tokens on a pre-defined subset of an actor's input arcs (usually the whole set). When an actor has met all the conditions of its firing rule, it is said to be "enabled." When the enabled actor does fire, it removes the relevant tokens at its inputs and places tokens on some or all of its outputs.

The most common actors are simple arithmetic operations, such as add and multiply. These actors have simple firing rules: all input arcs must have tokens present, and when the actor fires, all input tokens are consumed. Dataflow graphs also contain actors with specialized firing rules used to support conditionals and function calls [8, 24]. With the addition of these actors it is possible to support loops and other such high-level constructs.

Dataflow machines are divided into two general classes, *static* and *dynamic,* based on the relationship between tokens and arcs [109]. In a static architecture, two tokens cannot occupy the same arc at the same time. Therefore, the firing rules include a stipulation that all output arcs which would receive tokens must be empty,

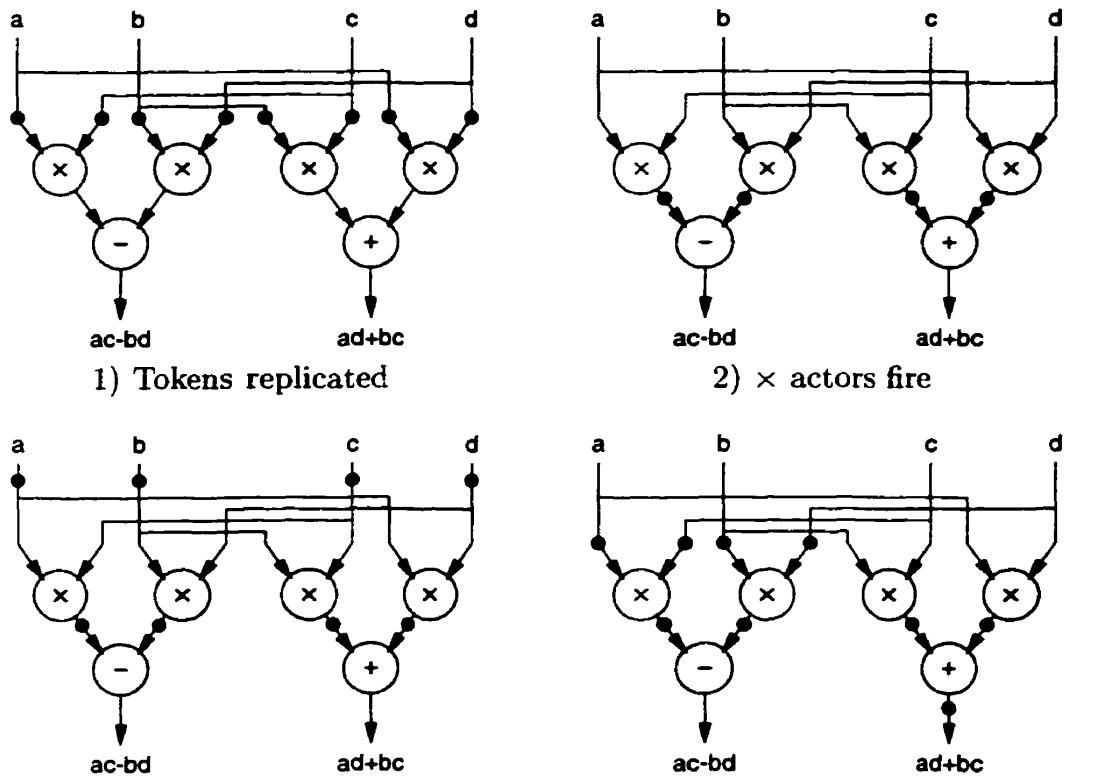


Figure 3.2: Execution of Complex Multiply

so that they have room to receive the new tokens. In a dynamic architecture, there may be many tokens on an arc at a given time. These tokens represent the results of different calculations using the same instructions (e.g., different iterations of a loop, or different invocations of the same function). It is necessary to differentiate between these tokens, so that two tokens representing different calculations do not get mixed together. Therefore, each token is given a *tag* (or *color*) [8], which identifies the function instance and/or iteration number, and the firing rules include the stipulation that all tokens to be used in one firing of the actor must have identical tags.

Figure 3.2 shows how an execution might proceed on the graph in Figure 3.1, if static firing rules were used. The diagrams show four successive states of the graph, representing the following stages of computation:

1. The tokens are propagated to the multiply actors, enabling them.

2. Each multiply actor may fire, producing a token at its output.

3. Once the input arcs have cleared, new tokens may be put there. However, the multiply actors may not fire, because their output arcs are occupied.

4. Once the addition actor fires, two multipliers may fire, but the other two remain blocked by their occupied outputs.

Execution will continue until two pairs of tokens are taken from the output arcs.

The example above is a very simple graph. Graphs corresponding to actual useful programs are much larger, and contain loops and function calls. Techniques exist for converting programs written in high-level languages to dataflow graphs. Conventional imperative languages can be converted to dataflow graphs, but the sequential semantics of those languages tends to limit potential parallelism. High-level languages based on functional language principles, such as Val [23], Sisal [31] and Id [94], usually generate dataflow graphs with more parallelism.

3.1.2 Static Dataflow

All static dataflow machines share the property that in the dataflow graphs on which they are based, an arc can only hold one token. Consequently, if there is a section of a program which is executed repeatedly (e.g., a loop body or a subroutine), the corresponding section of the dataflow graph cannot allow simultaneous execution of more than one instance of that code, as the example in Figure 3.2 illustrates. There are two ways to solve this problem:

- Pipeline the execution of the graph (Figure 3.2 illustrates pipelining, as the second set of input tokens can start flowing through the graph before the result of the first set of input tokens has been fully computed);

- Replicate the graph.

Pipelining the graph for maximal parallelism requires that the graph have a structure analogous to pipelined processors. The graph must be organized neatly into stages, with no internal cycles, and all paths through the graph must have the same length. Thus, shorter paths need to be filled with "identity" actors that simply pass tokens along [46, 40].

Replicating the graph works well when the number of iterations can be determined at compile time, as in regular numerical applications, but not when iteration counts are determined dynamically, e.g., irregular loops or binary recursion. The tradeoff between static and dynamic dataflow is that static dataflow puts more burden on the software (programmer and compiler), but reduces hardware complexity because there is no need to check tags.

The following subsections outline the development of static dataflow machines, starting at MIT in the 70s and continuing at McGill University in the late 80s.

Argument-Flow Machines The earliest proposal for an architecture to execute dataflow graphs was by Dennis and Misunas at MIT [27]. The basic idea was to convert a dataflow graph into an essentially isomorphic structure which would be more amenable to execution on real hardware. Since the main characteristic of dataflow graphs is that arguments to actors flow on arcs as tokens, machines similar to the Dennis-Misunas architecture [26, 122] are known as *argument-flow* machines.[1]

In an argument-flow machine, each actor is converted to an *instruction cell*. Thus, the abstract graph in Figure 3.1 is transformed into an *instruction cell program* (ICP) as shown in Figure 3.3. Each instruction cell has the following elements:

- An *opcode* identifies the type of function to be performed, e.g., a floating-point multiply.
- There are storage locations for the *operands* of each instruction cell.
- Each cell has a *destination list*, consisting of the *result list* and the *signal list*.
  - The *result list* lists the cells which are to receive the results of the computation, and corresponds to the output arcs in a dataflow graph. If the result list has more than one destination, it corresponds to a split arc in the graph. Each element of the list must specify both the destination cell and the position of the specific operand within that cell.
  - The *signal list* is needed to enforce the firing rules for static dataflow graphs. This rule is enforced by requiring a cell to send an acknowledgment to the source of each of its operands. A signal arc, represented by a

[1] This term was not used originally, but was coined later to distinguish this class of architectures from *argument-fetch* machines, a newer class described in the next section.

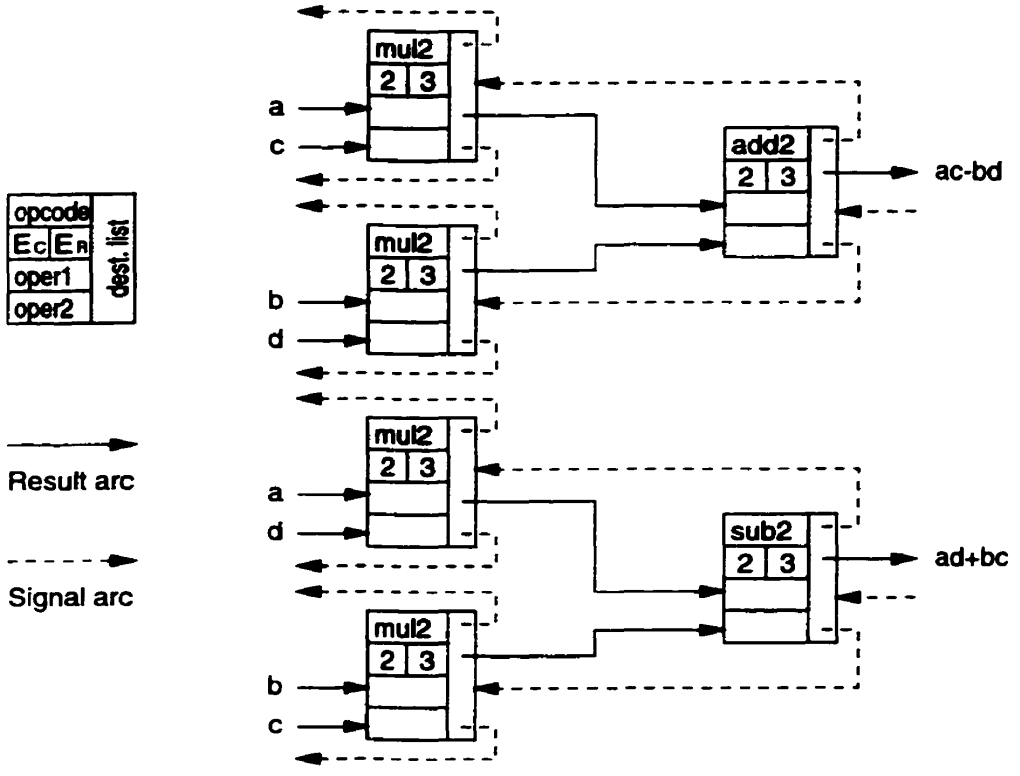


Figure 3.3: Instruction Cell Program for Complex Multiply

dashed line in Figure 3.3, tells the source cell that the results have been consumed, which means that the operand storage location is empty and ready to receive a new datum.

- Finally, each cell has two integer counts, the *reset count* (E_R in Figure 3.3) and the *sync count* (E_C). The reset count specifies the total number of results and signals which a cell must receive before it can fire. This number does not change during program execution. The sync count specifies the *current* number of results and signals which a cell needs before it can fire. This number changes and typically ranges from 0 to E_R. When the sync count is decremented to 0, the cell becomes enabled (ready to fire). When a cell finally does fire, the sync count is reset to E_R.

Figure 3.3 shows the initial state of the code, before any tokens have entered. (Since sync counts and operands change during execution, a drawing of an ICP can only represent a snapshot of program execution at a given time.) In this code,

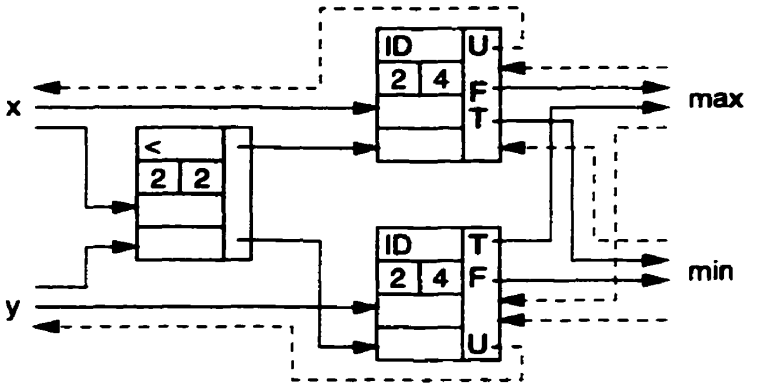


Figure 3.4: Instruction Cell Program for 2-Input Sort

there are two input arcs labeled *a*, two labeled *b*, etc. Since the two arcs labeled *a* represent the same value, they actually come from the same cell, which therefore has (at least) two destinations in its destination list. Both the acknowledgment arcs corresponding to *a* (one in each of two *mult2* cells) will go to that cell. Each cell needs two data tokens, and must have room to output one token, so the reset count of each cell is 3. Because there are no tokens currently in the graph, all output arcs are free; since this means no further acknowledgement is needed, all sync counts start at 2.

If maximal pipelining is not needed, then the code may be "linearized" by removing some of the signal arcs, which will lower the number of signals required during execution. For instance, the *sub2* and *add2* cells could signal the producers of the input tokens (*a*, *b*, *c*, and *d*) directly, rather than signaling the *mult2* cells. This would reduce the number of signals needed per iteration of the ICP from 12 to 8. There is a tradeoff, for while this would cut both storage requirements and signal traffic, the code could no longer be pipelined.

Figure 3.4 illustrates the role of conditional operators in argument-flow dataflow, by showing one possible implementation of a simple program fragment that sorts two numbers, producing the larger number at *max* and the smaller at *min*. First, there are instructions with boolean outputs, such as compare operators. In this example, the comparison operator produces a boolean token indicating whether or not x is smaller than y. Second, any ordinary operator can be augmented to accept a boolean value as an additional operand. In such an augmented instruction, each

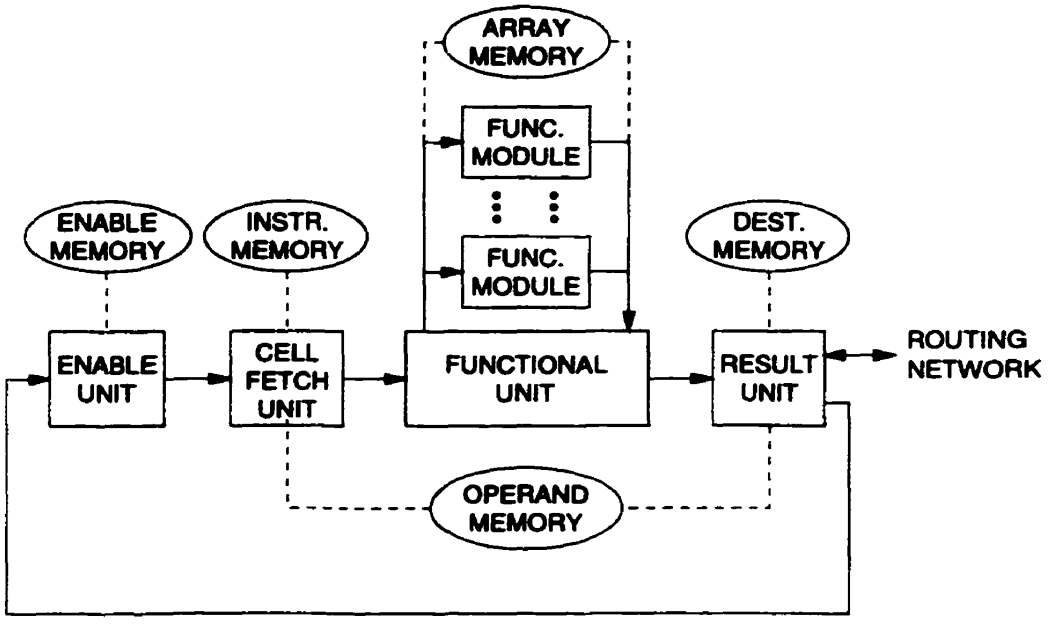


Figure 3.5: Processing Element Block Diagram

entry in the destination list can be tagged T (true), F (false), or U (unconditional). When the instruction fires, the special boolean value is used to determine which result and signal arcs are used. If the boolean input is T, then only the arcs tagged T or U are used. A boolean input of F selects only those arcs tagged with F or U. In this example, *identity cells* (which output their input unchanged) are used to route x and y to their proper destinations.

Note that there are no signal arcs from the ID cells to the comparison operator. Since all of these cells depend on the same inputs, the comparison cell can't possibly receive any new inputs until the ID cells have consumed the result of the old comparison and acknowledged their firing to the creators of x and y. Therefore, there would be no benefit to adding signal arcs from the ID cells to the comparison cell; no pipelining is possible. Correct pipelining would require additional ID cells between the creators of x and y and the existing ID cells.

One possible implementation of a processing element (PE) to execute instruction cell programs is shown in Figure 3.5. The PE can be divided into 4 pipeline stages arranged in a loop. The path in the main pipeline is drawn with a solid line, while auxiliary connections (e.g., to memory) are dashed.

There are five types of memory used in a Processing Element:

- The *Operand Memory* (OM) has a storage location for every input to every instruction cell.
- The *Instruction Memory* (IM) stores the opcodes of the instruction cells. Its contents do not change during run-time.
- The *Destination Memory* (DM) contains the result and signal lists for the instruction cells, which also do not change during run-time.
- The *Enable Memory* (EM) holds the sync count and reset count of each instruction cell.
- The *Array Memory* (AM) stores large arrays of data used by the program.

Broadly, the non-memory units of the PE make up two main sections: the Cell Execution section and the Execution Control section. The Cell Execution section takes cell numbers given to it by the Execution Control section, fetches operands, and produces results. It is pipelined in a manner similar to the execution units of modern conventional processors.

What makes dataflow unique is the Execution Control section. In a conventional processor, instruction execution is controlled by a simple counter with provision for overriding by a jump instruction. In a static dataflow machine, the Execution Control section is responsible for sequencing the firing of instructions. It must propagate tokens from one cell to the next, keep track of how many tokens each cell has received, and determine when cells are enabled (ready to fire).

The first module in the Cell Execution section is the *Cell Fetch Unit* (CFU). The CFU receives a cell number from the Execution Control section, fetches the opcode from the IM, fetches the operand(s) from OM, and passes the cell number, opcode and operands to the next stage, the Functional Unit (FU).

The CFU also fetches the boolean control value from the OM. The control tags in the DM determine whether or not the boolean value has any meaning. If a cell is not conditional, then all of its result and signal arcs are treated as if tagged U, and the boolean input will not be included in its reset count value. Therefore, the CFU can retrieve the boolean bit, and it will be simply ignored by the Execution Control section if the cell is not conditional.

The *Functional Unit* (FU) decodes the opcode to determine which operation to perform on the operands, and decides which functional module is to be used

to compute the result. There may be several functional modules, such as floating-point arithmetic units, integer arithmetic units, and boolean logic units. The AM is accessed as one of the modules. Since some of these modules may be pipelined (e.g., floating-point multipliers), the FU must keep track of the scheduling of operations in the pipelines, in order to match results with the cell numbers that initiated them. The FU outputs the cell number, the result of the operation, and the boolean control value which had originally come from the OM.

The *Result Unit* (RU) is the first module of the Execution Control section. The RU takes the cell number from the FU and fetches the destination list from the DM. The RU reads the destination list, and writes the result from the FU into every location in the OM specified in the list, provided that the tag in the list indicates the arc is a result arc, and that the tag matches the boolean tag passed along by the FU. For each entry in the list, result or signal, whose tag matches the boolean tag from the FU, the RU outputs the specified cell number to the final stage of the PE. *The RU must write a result into the OM before outputting the corresponding cell number to the next stage.*

If the PE is part of a multiprocessor, then the RU also has an interface to an interconnection network. In this case, cells may transmit results and signals to cells in other processors. Therefore, the destination list must include PE identifiers in its addresses. If an entry in the destination list specifies a different PE, a packet is formed (containing the PE number, cell number, operand number, and the value) and sent over the network. When a packet is received from the network, the value is written into the specified operand of the specified cell, and the cell number is sent to the next stage, just as if it had been in the destination list of a local cell.

The cell numbers from the RU go to the *Enable Unit* (EU), which controls the firing of instructions.[2] When the EU receives a cell number, it decrements the sync count for that cell. If the result is non-zero, the new sync count is put back into the EM, as this cell is not yet ready to fire. If the result is 0, the cell is ready to fire. The EU continuously looks for enabled cells to fire. When it fires a cell, it copies the reset count of that cell into the sync count, and outputs the cell number to the CFU. This completes the cycle.

The four units described above form a closed loop, and each is independent of the others. Therefore, all units can operate in parallel, and each individual unit can

[2]A prototype Enable Unit chip was designed and tested at MIT [38].

be pipelined.

An important property to note is that *there are no hazards in a proper dataflow program.* Pipelining a conventional processor introduces potential hazards. A read-after-write hazard, for instance, occurs when one instruction uses the result of a preceding instruction which is closer than the length of the pipeline. Guarding against these hazards is a major source of complexity in conventional pipelined processors.

However, the dataflow conventions prevent these kinds of hazards in proper dataflow programs.[3] In a proper dataflow graph, a read-after-write hazard can't occur, because if cell B requires an operand produced by cell A, then A must signal B directly or signal a chain of intermediate cells which signals B. In either case, B cannot fire until the RU has sent B's cell number to the EU, which will eventually fire B). However, the RU will not send B's cell number to the EU until it has written the result of A's operation into the OM, so that B is guaranteed to read the proper data value.

Argument-Fetch Machines One shortcoming of the argument-flow implementation is the need for excess storage for and copying of operands. For instance, the complex components in the program in Figure 3.3 must be duplicated. The storage overheads in the sort routine in Figure 3.4 are even worse due to the identity cells; if maximal pipelining were not required it would be more efficient if x and y could be compared, and swapped in place if necessary.

Argument-fetch dataflow was proposed as a way to address this weakness. In an argument-fetch machine, data values are not attached to specific cells, but can be stored anywhere in the OM. This means that instruction cells must contain references to those locations. Data no longer "flows" from one cell to another; only signals flow. The program in an argument-fetch machine looks much less like a dataflow graph, though it is functionally equivalent if it is constructed properly.

The components of an argument-fetch processor are similar to the argument-flow PE shown in Figure 3.5. They differ in the following ways (terms from Section 3.1.2 are used even though they may be named differently on other machines):

- Each instruction must specify the address or addresses in the OM where the

[3]Loosely speaking, a proper dataflow program is one which is isomorphic to a classical dataflow graph, meaning, for instance, that no operation reads an operand until being signaled (directly or indirectly) by the producer of that operand, and the program is fully determinate.

operands for this instruction are located. The Cell Fetch Unit must perform this fetch operation as well as fetching the original instruction.

- The Result Unit no longer has to write multiple copies of the result. Instead, it writes it to a specific place in the OM, a location encoded in the instruction. It still requires a destination list to teli successive cells that their inputs are ready and to tell preceding cells that they can overwrite their previous results, but these are now purely signals. They go directly to the Enable Memory and have no effect on memory.

- The interconnection network may or may not support the argument-fetch principle. Fetching an item stored in a remote PE requires either performing a round-trip through the network or adding support for shared memory. Alternately, one could require that interprocessor transfers be done according to argument-flow principles, which would make network transfers simple and fast, but at the cost of not having a single paradigm cover the entire machine.

These changes in the execution pipeline make the Cell Execution section (CFU and FU) act more like a regular RISC pipeline. In 1988, the CFU, FU, and result-writeback part of the RU were combined into a single unit, the *Pipelined Instruction Processing Unit* (PIPU), and the remaining stages were combined into the *Dataflow Instruction Scheduling Unit* (DISU) [25]. Essentially, those features of the argument-fetch machine which are unique to dataflow were placed in the DISU, leaving the PIPU with all the normal RISC-like features. This made it feasible to implement the PIPU with a standard RISC pipeline unit with minimal modifications.

At the interface between the two halves, the DISU sends cell addresses to the input of the PIPU, much as the EU sends cell IDs to the CFU in Figure 3.5. When the PIPU has finished executing a cell, it writes the computation's result back into the OM and sends the cell identifier back to the DISU. The DISU then looks up the signal list for that cell and updates the sync counts of all affected cells.

This is a natural division for a dataflow processor. The PIPU performs the actual execution of each dataflow instruction, while the DISU is responsible for all the synchronization. This division of labor has since been used in other dataflow/multithreading systems, including EARTH.

3.1.3 Dynamic Dataflow

One shortcoming of static dataflow is that multiple loop iterations and multiple calls to the same function are not possible unless code is explicitly replicated at compile-time. Dynamic dataflow addresses this problem by allowing simultaneous use of one section of dataflow code by more than one computation. This is equivalent to allowing more than one token to be on an arc of a dataflow graph at the same time. For instance, the 2×2 sorting code fragment in Figure 3.4 could be used as the basis for a larger sorter using binary recursion. On a dynamic dataflow machine, a large sort problem could be decomposed into a large number of 2×2 sort operations, and all of these could be entered into the same piece of code. Many 2×2 sorts could be executed concurrently (subject to flow dependences).

Data values can no longer be stored in specific slots in memory, as in the argument-flow and argument-fetch machines. Their storage must be as dynamic as the code itself. The solution is to keep data bound to the tokens all the way until they are ready to be consumed. (By contrast, the static machine in Figure 3.5 separates data from control in the Result Unit, which writes result values to the OM while sending the corresponding signals to the Enable Unit.)

It is also necessary to keep the various computations from interfering with each other. To keep them apart, each token has a tag. This tag, first proposed for the *U-Interpreter* (or *Unraveling Interpreter*) [8] has fields identifying the specific instruction in the code which is to receive the token, as well as which operand it is ("left" or "right"; all operations are monadic or dyadic). This is similar to the argument-flow machine. However, there are two additional fields, one which can uniquely identify different occurrences of the same function body, and another which can identify different iterations of a loop. Whenever a function is called, a new tag field for that function is created. Thereafter, tokens created within that function bear that field, except those tokens explicitly being returned to the calling function. Similarly, the first iteration of a loop is given an iteration field of 1, and special explicit instructions are provided for generating new tags with an iteration field which is 1 higher.

The *Tagged Token Dataflow Architecture* (TTDA) [10], based on the U-Interpreter, has a pipeline structure similar to Figure 3.5, with the following important differences:

- Instead of an Enable Unit, there is a *Waiting-Matching Section* which contains an associative token-matching memory. Tokens generated as the output of dataflow instructions go into this stage (possibly after going through the interconnection network). When a token goes into this stage, the associative memory checks to see if the other token corresponding to the incoming token is already there. If it is, then both data values are retrieved and passed to the next stage of the execution pipeline. If the other one is not yet there, the new token is placed in the memory to wait for its partner.

- The Result Unit does not write directly to memory, but merely generates tokens with the data values included, and sends these to the Waiting-Matching Section.

While the TTDA has been simulated, other machines based on the same concepts have been built. These include the Manchester Machine [48] and the Sigma-1 [55].

3.1.4 Semi-Dynamic Dataflow

The static dataflow machines in Section 3.1.2 are not able to reuse the same code to execute several instantiations of that computation at the same time. This makes it impossible to execute something as simple as a binary-recursive function,[4] even sequentially. If function f calls itself, tokens belonging to the caller must remain in the instruction cells for f, which prevents a clean execution of another instance of f. Dynamic dataflow, as described in Section 3.1.3, allows simultaneous reuse of code blocks, permitting full exploitation of parallelism in recursive functions and loops. However, this flexibility comes at a high price: the associative token-matching logic is simply too complex to be used in a practical machine [98].

Proponents of both static and dynamic dataflow have addressed their respective shortcomings by adding features which amount to a convergence of the two styles of computing. All of them essentially use static dataflow rules within a function and dynamic dataflow to schedule functions, which basically allows parallelism to be exploited in recursive functions, but not in loops. We propose, therefore, that machines fitting this description, such as the three designs surveyed here, constitute a separate class called *semi-dynamic dataflow* machines.

[4] Tail recursion can be converted to an appropriate loop.

In the multiprocessor proposed by Rumbaugh [99], functions are executed on separate processors called *activity processors*. Each activity processor has local memory which holds the state of one function invocation. called a *procedure activation*, which contains instructions, sync counts and room for operands for one function invocation. Within a function body, the activity processor acts much like a static dataflow machine. Only one function may be run on an activity processor at a time, so if a function is called (using the *Apply* operator), the called function must be started on another processor. The new processor initializes its local memory with a procedure activation for that function, and begins executing that function. The new procedure activation has a pointer to the caller so that it knows where to return the value of the function. Thus, at any point in time the program state is represented by a *tree of procedure activations*. If a function is called and no processors are available, an existing procedure activation must be swapped out into an auxiliary *swap memory*.

The Monsoon multiprocessor built at MIT [98] evolved from the dynamic TTDA (see Section 3.1.3). Thus, its internal synchronization is more similar to dynamic dataflow than to static dataflow. When a function is called, an *activation frame* is allocated from general-purpose memory. If a function has n instructions, then the frame holds an $n \times 2$ array of values, each value tagged with a full/empty bit. Each token includes the address of the base of its activation frame. All instructions have only one or two operands. Together, these features ensure that any legally-generated token corresponds to a specific place in memory, and if that operand is part of a dyadic instruction, then the other operand is easily located. Therefore, synchronization is straightforward. There are neither sync counts nor an enable unit, just a simple mechanism that goes directly to the right place in memory and checks the full/empty bit of the other operand if necessary.

The McGill Data Flow Architecture (MDFA) [42] is an enhanced version of the argument-fetch machine described in Section 3.1.2, designed to address the lack of recursion in static dataflow. Like the argument-fetch machine, the MDFA performs dataflow scheduling in the DISU and instruction execution in the PIPU. The MDFA processor includes a new unit, the *Memory Overlay Manager* (MOM), which handles frame allocation. When a function is called, the MOM allocates a frame from memory and sends the frame's base address to the DISU. Thereafter, tokens in that function invocation are tagged with this address. The frame itself contains sync counts and local variables. All addresses for sources and destinations of instructions

are coded as offsets from the base of the frame, and the actual base address is added to these offsets at run time by the PIPU.

3.1.5 Problems with Dataflow

The dataflow machines discussed in this section offer superior primitives for fine-grain synchronization. However, these synchronizations aren't free, and the cumulative cost of fine-grained synchronization can be quite high [36, 62, 88]. For instance, if most operations in a dataflow program are dyadic, then an average of two synchronization events must occur for every instruction executed. Simulations on the MDFA have demonstrated the negative impact on program performance of too many fine-grain synchronizations [62].

Another problem that results from requiring synchronizations between individual instructions occurs when the program reaches a sequential section and, due to dependences in the application, there are no other instructions outside of this section to execute. If a dataflow execution pipeline such as the one in Figure 3.5 has n stages (meaning that if cell A signals cell B, and B has a sync count of 1, then a minimum of n cycles elapse from the time A is fired to the time B is fired), then the processor utilization while executing a purely sequential piece of code (e.g., flow dependences between every pair of successive instructions) is only $\frac{1}{n}$.

A further shortcoming of dataflow machines is that they do not exploit locality effectively [88, 89]. As described in Section 1.2.1, single-thread processors exploit the fact that most data have short lifetimes, meaning most data are consumed and discarded shortly after being produced. Therefore, single-thread processors can keep most data in a small set of registers, which can be placed close to the execution pipe and consequently made fast. Though experiments have shown that such locality also exists in dataflow programs [89], this locality is much harder to exploit in a dataflow processor, because the indeterminate order of instructions will increase the average lifetime of each datum, and make it impossible for a compiler to analyze and predict these lifetimes. This is borne out by reviewing the dataflow machines in the previous section. The space required to hold the operand memory in Figure 3.5 or any of the equivalent structures in the other programs is much larger than the size of the register sets in a typical RISC processor. Requiring the execution pipe to have fast uniform access to such a set can force a reduction in the clock speed

because of longer delays in the data paths.

3.2 Hybrid Von Neumann/Dataflow Machines

Hybrid von Neumann/dataflow architectures [64] seek to reduce fine-grain synchronization costs and improve use of locality in dataflow architectures by combining dataflow actors into threads, or, alternately, to add latency-tolerance and efficient synchronization to conventional multithreaded machines by adding dataflow synchronization to the thread model. Hybrid machines use a dataflow-like form of synchronization, but combine two or more dataflow actors into a thread which is executed sequentially, as in a multithreaded machine. This "threading" of the dataflow code leads to several benefits:

- The number of synchronizations between dataflow actors is reduced by a factor equal to the average number of instructions per thread. For even a small thread size, this reduces the cost of instruction synchronization from substantial (even greater than the cost of the computation itself) to manageable. There are collateral reductions in the signal lists and the space used for sync and reset counts.

- By running a sequential thread, the processor can keep intermediate results in a conventional set of registers, both increasing speed and decreasing the memory needed to hold the program state (the operand memory in a static dataflow machine or the activation frame in a semi-dynamic dataflow machine.

The Super Actor Machine at McGill followed the MDFA (see Section 3.1.4) and was designed to address the problems encountered with the MDFA (and with dataflow in general). This machine may be considered the closest to an immediate ancestor of EARTH, and is discussed in detail in the next subsection. This is followed by a brief survey of other multithreaded machines and models which are derived from dataflow principles.

3.2.1 The Super Actor Machine

The *Super Actor Machine* (SAM) [56, 62] approaches the problem of making a hybrid design from the dataflow side, so it is primarily dataflow-oriented. A program is

written in a high-level language and converted into a dataflow graph using one of many available translators. An algorithm is used to analyze the graph and combine dataflow actors into multiple-actor units called *super-actors* [62]. The algorithm attempts to minimize the number of synchronization arcs that cross super-actor boundaries. The super-actors are then translated into threads that can be loaded into the SAM.

Internally, the SAM has a pipelined execution unit and a dataflow scheduling unit similar to the PIPU and DISU of the argument-fetch dataflow machine. The bulk of data is stored in a standard main memory. An additional unit handles remote accesses.

An innovative feature of the SAM is its *Register Cache*. The execution pipeline has several sets of registers, only one of which can be accessed by a given thread. A thread accesses its assigned register with short bit fields, as in a regular RISC processor. While the thread is accessing these registers, another unit of the SAM can be filling other registers with values that will be accessed by a thread that will run later.

The benefit of this is the elimination of local memory latencies. The standard dataflow scheduling paradigm eliminates busy-waits because consumers are not enabled until all their inputs are ready. However, this does not guarantee that all inputs are *immediately accessible* to the execution pipe. Failure to guarantee this could cause load stalls when a thread tries to access something which is in main memory. (Dataflow machines get around this problem simply by assuming that all operand memory is quickly accessible, an assumption we argued against in the previous section.)

The SAM solves this problem by adding extra states to the normal dataflow actor states (not-enabled and enabled), as shown in Figure 3.6. First, because threads will take much longer to execute than simple dataflow actors, there is an "active" state which says that the super-actor is currently in the execution pipe. More important is the distinction between the "enabled" and "ready" states. When the last required input reaches a super-actor, the scheduling unit changes its state from "dormant" to "enabled," as in a conventional static dataflow machine. However, the inputs will be located in the processor's main memory, and would cause load stalls in the execution pipe if they were accessed directly. Therefore, when a super-actor becomes enabled, an auxiliary unit takes an unused set of registers from the Register Cache

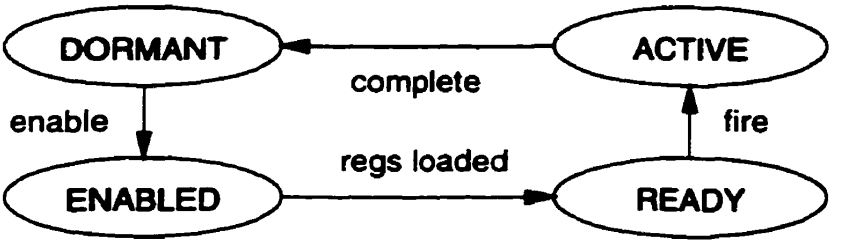


Figure 3.6: Super-Actor States

and begins copying the data from main memory to this set. When all inputs have been loaded into the registers, the super-actor enters the "ready" state, there to wait until eventually being executed by the execution pipe. While the super-actor is in the "ready" state, the register set assigned to it must wait and cannot be deallocated. When a super-actor has finished, it goes back into the "dormant" state.

3.2.2 Other Dataflow-Based Multithreaded Machines

The Multi-Level Execution model [88] was an early proposal designed to address the problems of high synchronization overheads in pure dataflow. In this model, individual scalar actors in a dataflow graph are combined into *macro-actors* to amortize synchronization overheads. Macro-actors are matched and synchronized in a *Matching Store Unit* similar to the Waiting-Matching Section of the TTDA (see Section 3.1.3). Internally, these actors are converted to microcode and executed on parallel functional units using ordinary registers with forwarding logic. The execution of macro-actors is similar to the execution of sequential code on modern multi-issue superscalar processors.

The Hybrid Multiprocessor [64, 63] combined dataflow ideas with sequential execution to define a hybrid model. This eventually led to Empire, a multithreaded architecture project at IBM. This architecture uses local frames, like the semi-dynamic dataflow machines in Section 3.1.4. Frame locations have *presence bits* indicating whether or not the data is valid, and if a thread tries to read an invalid location, it is suspended until the location is filled. Other features include processor ready queues with process and packet priorities, and support for efficient process migration to facilitate dynamic load balancing.

Another descendent of the MIT dynamic dataflow work is P-RISC [92], in which

sequential threads are controlled with fork and join primitives. The P-RISC proposal explored the possibilities of constructing a multithreaded architecture around a RISC processor. The synchronization primitives were to be controlled by special multithreading instructions added to a regular RISC processor.

The Datarol processor [5] is a dataflow processor in which each instruction explicitly identifies its successor instruction(s). Registers are used for short-term storage. Its successor, Datarol-II [67], combines instructions into sequential threads, but gives each thread its own set of registers. Each thread has a list of successors which are signaled when the thread is finished; the successors are associated with dataflow-style counters for controlling threads.

The EM-4 [102] is based on dynamic dataflow principles, but uses frames for local variables, and connects instructions within "strongly connected" subgraphs of a function body into sequential threads. Continuation of this research has produced the EMC-Y processor [70]. The RWC-1 project [101] of the Real World Computing Partnership in Japan aims to build multithreaded multiprocessors with RICA (Reduced Interprocessor Communication Architecture) nodes. A RICA node has a custom microprocessor with a superscalar RISC core and embedded mechanisms for fork-join thread synchronizations.

One characteristic of all the preceding designs is a custom processing unit. Because of the difficulties of building processors with performances approaching today's commodity sequential processors, particular for academic and research groups, only the EM-4 (among the list above) has actually been built. However, there are other projects which have looked at supporting dataflow-based multithreading with off-the-shelf technology.

*T [91] is a multithreaded design which, like the Argument-Fetch machine (Section 3.1.2), separates dataflow-like synchronization and instruction execution into different units. The difference is that these units are implemented with off-the-shelf microprocessors, both of which are user-programmable. A follow-on architecture called *T-NG [7] specifies the addition of a network interface unit to each of four PowerPC 620s in a "site" where a multiprocessor system consists of many sites. Built-in snoopy mechanisms in the processors are used for cache consistency in a site, while the tasks for cache consistency between sites are relegated to one of the four processors in a site. The same processor dedicated to inter-site cache consistency is responsible for handling split-phase memory requests and synchronizations.

A completely different approach to multithreading is to build a translator for converting multithreaded programs into code which can be run fairly efficiently on off-the-shelf processors, rather than trying to specify a custom architecture. One example of this approach with dataflow roots is the Threaded Abstract Machine (TAM) [21]. TAM uses a tree of activation frames, like many of the semi-dynamic dataflow machines, but the memory hierarchy is tailored to standard multiprocessor hierarchies. Furthermore, synchronization between threads (e.g., passing data or signals) is done in software, using *inlets*, small user-programmable message handlers included in every function. Both conventional and functional languages are translated to an intermediate language TL0, which is then compiled for one of several target machines.

This software-based approach yields results quickly, as a translator can be designed and implemented much more quickly than a custom processor. Other parallel systems using this approach are covered in Chapter 9. It is also shown in Chapter 7 to be one way of implementing our EARTH model.

Chapter 4

Definition of the EARTH Model

The previous section presented many possible designs for parallel computer systems. In an ideal world, our desire would be to implement one of these, optimizing all components of the system for achieving large-scale parallelism. In the real world, such an undertaking would be difficult and expensive. Parallel machines must compete with modern commodity microprocessors, whose thousands of engineer-years in design time and billions of dollars in capital costs can be amortized over a large sales volume. For this reason, most parallel machines today are based on off-the-shelf processors and other components.

Unfortunately, neither these processors nor the programming systems that typically run on them provide adequate support for features important to parallel machines, such as latency tolerance and interprocessor synchronization. Section 1.2 argues that this will limit their effectiveness for many applications. But these processors are unlikely to include efficient hardware support for parallelism in the near future, since they are designed for a highly competitive uniprocessor market. If we want parallel systems to make effective use of these processors, we must address the problems at a different level.

Specifically, we need a parallel programming model which addresses the performance issues important to parallel machines, yet which can be implemented on a computer built with off-the-shelf processors. The performance of such an implementation may not be ideal, for software cannot always compensate for hardware deficiencies. Nevertheless, the establishment of this model can pave the way for the addition of better hardware support for features of the programming model not supported by current processors.

The problem statement of this dissertation proposes an *evolutionary* or gradual approach to building a full-scale multiprocessor. The basic plan is to start with a parallel system based on stock hardware, and move step-by-step toward a fully customized implementation. Each step should be a viable, functional system offering improvements over the preceding system, albeit at some cost. Intermediate steps move some of the functions of the architecture to custom chips or dedicated coprocessors, with the remainder being implemented in off-the-shelf hardware. Software for such a system may follow a similar path making use of off-the-shelf packages, since compilers and operating systems also require large investments.

It is important to keep transition costs down for each step of the way. This includes not only design costs, but also the costs to the users. Moving an application from one machine to a more "evolved" machine should require no more than recompiling. Since this means the application source code should remain the same from one end of the evolutionary path to the other, the essential characteristics of the model should remain constant over this path. (Of course, as performance improves in each step, the programmer may modify the code to take advantage of these improvements, e.g., to generate more parallelism, but this should not be a requirement.)

The challenge is to construct a parallel programming model which is portable along the path without sacrificing efficiency. To elaborate, our goal is to design a model with the following characteristics:

Efficiency: It adequately solves the performance issues discussed in Section 1.2 (latency, bandwidth, and synchronization).

Programmability: It allows programmers to express parallelism in their applications efficiently and easily. This should be true both for applications with regular control and data distributions and for "irregular" applications.

Simplicity: It is simple enough to be implemented at the near end of the spectrum (using an existing off-the-shelf multiprocessor) without sacrificing the performance gains that can be realized by moving to custom hardware at the far end of the spectrum.

Flexibility: The machines at the near end of the spectrum will use commodity hardware, and later machines, though custom-designed, may still make use

of existing component designs (e.g., processor cores or modules). This architecture should have the flexibility to take advantage of different components at different stages in the path, or even in the same stage to satisfy different price-performance requirements. Therefore, the architecture shall make as few specifications as necessary of such features as the instruction set, clock speed, bus and memory requirements, etc.

This chapter presents the *Efficient Architecture for Running THreads* (EARTH), a multithreading model meeting these goals [60, 61, 59, 83]. EARTH falls into the class of hybrid von Neumann/dataflow machines described in Section 3.2. However, unlike the designs covered there, EARTH is amenable to the kind of evolutionary design process described above.

It is very important to clarify what EARTH is and what it is not. Since we are proposing an evolving series of computers based on increasing amounts of custom hardware, EARTH does not refer to any specific machine or design in this series. Instead, it refers to a particular model of multithreading, as presented in this chapter, and any machine which can adequately implement this model can be called an EARTH computer. Chapters 7 and 8 give examples of real and simulated machines, which implement the EARTH model at various points in the evolutionary path.

An architecture can be presented at several levels. The most common view of an architecture is the *Instruction Set Architecture* (ISA), which gives specific details of instructions, registers, and their interactions, usually including an operational semantics with enough detail that the programmer can accurately predict a given program's behavior at a specified level of detail. The ISA is generally specific to one processor or family of processors. Alternatively, one can talk about the components of a system at a more abstract level, describing, in general terms, the objects visible to the programmer, the operations which can be performed on these objects, and the general method for representing and executing computations on this machine. We use the term *Program Execution Model* (PXM) for this abstraction.[1] Somewhere between these two points lies a high-level abstraction of the components of the machine and the way they interact, which we call an *Architecture Model.*

The PXM for a conventional sequential machine would include an addressable memory component, some of which is divided into frames placed on a stack. Programs are divided into functions; when the machine executes a function, a frame on

[1]This concept is similar to Valiant's concept of a "bridging model" [127].

the stack is assigned to this function for its private use. Instructions are executed in sequential order, except when a branch instruction explicitly redirects execution to another location. A program counter refers to the next instruction in the sequence to execute. The PXM ignores ISA details such as the opcodes of specific instructions, and is thus universal for all modern general-purpose sequential processors. It is this universality that has made standard programming languages based on this PXM, such as C, so portable among ordinary processors.

We present the PXM for EARTH in the next section. The EARTH PXM extends the conventional PXM above with objects and operators specifically for supporting parallel multithreading. We discuss the representation of programs as collections of threads, the synchronization among these threads, and the context of each thread. This section concludes with a discussion of a memory model and a basic set of primitives supported by the PXM. In Section 4.2, we present an architecture model for EARTH, a general blueprint for all the implementations in Chapters 7 and 8.

We have deliberately kept EARTH simple to achieve our goal of portability from one end of the evolutionary path to the other. The early papers on EARTH [60, 61] suggested dividing EARTH into "levels" of complexity, in which higher levels support more features. We sought to identify a minimum set of features necessary to support efficient multithreading, so that an implementation using off-the-shelf processors would be feasible. The implementations in this thesis are based on the lowest level. However, many mechanisms have been proposed over the years for expressing parallelism more easily or efficiently. In the last part of this chapter, we consider some which could be added to EARTH, particularly further along the evolutionary path where implementations would be more efficient.

The next chapter presents specifications for two slightly different complete sets of EARTH operations. These serve the same function as an ISA, in as far as being a target for compilation from a higher-level language. However, a detailed ISA is not suitable for this project, given our desire for flexibility and platform portability, since an ISA would bind EARTH to a specific processor family. Instead, we describe an *EARTH Virtual Machine* (EVM) for each. Each EVM is a *partial* specification of the instruction set of an EARTH machine. The EVM defines a set of instructions which must be present in any EARTH computer, and defines the semantics of these instructions in relation to the EARTH PXM. It leaves open both how these EARTH instructions are *implemented* (e.g., in software or hardware) and how they

are *represented* (e.g., the opcodes used). Since some implementations of EARTH use off-the-shelf processors, pre-specified opcodes would be meaningless to such processors. Illegal-instruction traps could be used, but these are generally inefficient because of the operating system overheads involved in processing the trap. Instead, the EARTH instructions could be signaled using accesses to memory-mapped I/O or, in the case of a pure software implementation, could be converted to instructions native to the off-the-shelf processor. The EVM only specifies which instructions must exist, and describes the semantics of these instructions.

4.1 The EARTH Program Execution Model

The Program Execution Model for EARTH differs from the PXM of a sequential computer in the following important respects:

- Instead of a single program counter, there can be multiple program counters, allowing concurrent execution of instructions from different parts of the program.
- Programs are divided into small sequences of instructions in a two-level hierarchy of threads.
- The execution ordering among threads is determined by data and control dependences explicitly identified in the program, rather than by program order.
- Frames holding local context for functions are allocated from a heap rather than a linear stack.

The following subsections present the EARTH PXM in more detail. The most unique aspect of this PXM is its threading model, which is covered first. Section 4.1.2 discusses the EARTH memory model. The final subsection gives a list of the fundamental objects of the EARTH PXM and the operations on these objects which are basic to the smooth functioning of EARTH. Our goal in this section is to justify the need for each feature of the PXM, rather than merely present it. Therefore, a simple example, based on computing Fibonacci numbers, is developed during the discussion

4.1.1 EARTH Thread Model

The EARTH *thread model* is the most important defining characteristic of EARTH, distinguishing EARTH both from conventional parallel paradigms and from most other multithreaded machines. What this model has in common with other multithreaded machines is the division of a program into multiple sections of code, generally called *threads*. Multithreaded machines based on conventional processing models divide a program into threads to identify computations which can run concurrently; they *parallelize a sequential program*. Multithreading models derived from the dataflow model, such as those in the previous chapter, combine individual instructions into threads to reduce and amortize the overheads of synchronization and improve data locality; they *sequentialize a parallel program*.

Our goal of making EARTH's PXM suitable for off-the-shelf processors has led to a two-layer hierarchy of *fibers* and *threaded procedures*. The following subsections present this two-layer model and describe its individual components. The first defines *fibers*, which are in the lower layer, and shows how their properties are essential for an efficient off-the-shelf implementation. The second part gives an example of a simple parallel program, and shows the difficulties that this program can present for a parallel machine based on a single-layer model. We show how a two-layer model is an effective solution. The remaining parts present the two layers in greater detail.

4.1.1.1 EARTH Fibers

In ordinary sequential code, the next instruction executed is completely determined by the preceding instructions and the input data. The EARTH model maintains the ordering constraints among instructions within one thread, but loosens the constraints between different threads, allowing the processor to adapt better to runtime conditions such as unpredictable latencies. In the EARTH model, a thread is a *sequentially-executed, non-preemptive, atomically-scheduled* set of instructions. All three qualifiers are basic to the model, and are necessary for efficient execution on conventional processors. Because many other multithreading systems, such as those described in Section 3.2 and Chapter 9, use the term "thread" for entities with different properties, the EARTH PXM introduces the term "fiber" to refer specifically

to the type of thread above.[2] We use the term "thread" only generically, to refer to a block of instructions which can run concurrently with other blocks.

Sequentially-executed means that when a fiber is executed, instructions within the fiber are scheduled according to a sequential semantics. In other words, instructions within the fiber are ordered using an ordinary program counter, which increments to the next instruction unless modified by a branch instruction. Both conditional and unconditional branches may be used, but only to destinations within the same fiber. Modern processors perform sequential execution very efficiently, even when there are many dependences among the instructions, and can takes advantage of the data locality which is usually present due to these dependences. Techniques used by modern superscalar processors to increase the instruction issue rate, such as out-of-order execution and branch prediction, may be used to exploit instruction-level parallelism *within* a fiber, so long as the results are the same as executing the instructions in purely sequential order. "Sequentially-executed" in this case does not mean "one instruction per cycle," but simply that the dynamic ordering of instructions within a fiber conforms to the sequential semantics of the code.

EARTH fibers are also *non-preemptive*. Once a fiber begins execution, it remains active in the CPU until the last instruction in the fiber is finished. If the CPU should stall (e.g., due to a cache miss), the fiber will not be swapped out. This is a fundamental design decision based on the goal of using existing processors. At any point in a fiber's execution, there is likely to be some essential *context* (such as live register values). Ordinary processors don't support rapid context switching, so if a fiber is interrupted, the CPU would have to save the live registers and load some registers for the next fiber.[3] An automatic mechanism for fiber suspension, such as one based on interrupts, would have to make conservative assumptions about which registers are live and would probably save a large number of them. This takes time, both for the triggering of the interrupt and the saving and restoring of registers, and the frequent use of such a mechanism would severely limit system performance.

A corollary of non-preemptive execution is *atomic scheduling*. If a fiber cannot be interrupted, then it should not be started until it is guaranteed to finish without any major stalls. The EARTH implementation is responsible for making this

[2]This term, like "thread" itself, comes from the lexicon of textile making. A fiber is typically a short strand of material. It is the smallest unit in the "thread model" of textiles.

[3]Fibers may be interrupted for special exceptions such as arithmetic traps, but these should be assumed to be unusual cases and not normal occurrences such as cache misses.

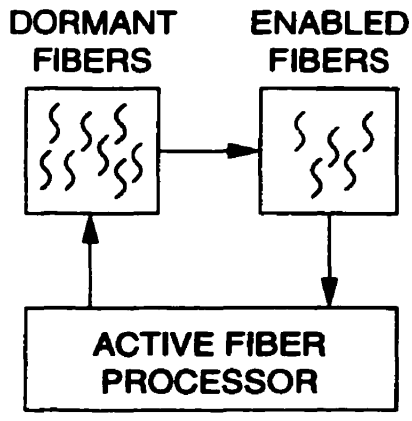


Figure 4.1: Abstract Fiber Execution Engine

determination (using mechanisms discussed in a later section) and deciding when a fiber can start according to this restriction. Borrowing terms from dataflow (see Section 3.1), we say that when the system decides a fiber is ready to execute, it *enables* the fiber. Since the CPU may still be busy with other fibers at that time, there may be a delay between the time a fiber is enabled and the time it starts running. We call the first state *enabled* and the second state *active*. A fiber that is not ready to begin execution is *dormant*.

Figure 4.1 shows an abstract model of a machine for executing EARTH fibers. There are two pools of fibers, one for dormant fibers and the other for enabled fibers. When fibers are enabled, they are moved from the dormant pool to the enabled pool. When the *Active Fiber Processor* has free resources for executing a fiber, it takes one from the enabled pool, making it active, allocates the resources needed by this fiber and begins execution. When a fiber finishes execution, the processor returns it to the dormant pool and frees the resources. The EVM should specify a special instruction marking the last instruction of a fiber, in order to simplify the implementation of fibers on off-the-shelf processors, which need to know where the fiber "ends."

4.1.1.2 EARTH Thread Hierarchy

Figure 4.2 shows a simple recursive program for computing the n^{th} Fibonacci number. Notwithstanding that this is actually a terrible way to compute Fibonacci numbers, it's a good example to illustrate the basic threading model, as well as a

```
int fib(int n) {
    int left, right;
    if (n<2) {                  /* S1 */
        return 1                /* S2 */
    } else {
        left = fib(n-1);        /* S3 */
        right = fib(n-2);       /* S4 */
        return left+right;      /* S5 */
    }
}
```

Figure 4.2: Sequential Fibonacci Example

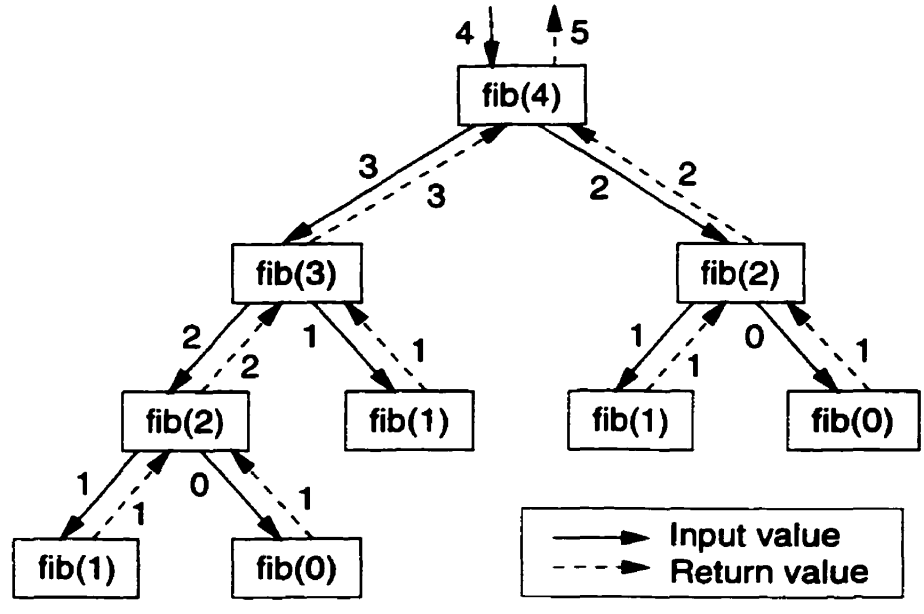


Figure 4.3: Call Graph for Sequential Fibonacci

good benchmark for measuring multithreading overheads, and will be used throughout this thesis. (More realistic examples of binary recursion are presented in later chapters.) For simplicity, negative inputs aren't checked. Figure 4.3 illustrates the call graph resulting from calling `fib(4)`.

An obvious way to parallelize this program is to run separate function calls in parallel. For instance, the call to `fib(4)` could spawn separate processes to compute fib(3) and fib(2), and these could run on other processors. But what should be done

with the execution of `fib(4)` before `fib(3)` and `fib(2)` have returned their results? If we want to use the processor for some other computation (such as one of the child functions), we must suspend `fib(4)` and switch to another context.

However, this suspension would contradict the non-preemptiveness of EARTH fibers. Furthermore, the commodity processors we would like to use in a multithreaded machine have no mechanisms to check that data accessed by an executing fiber is actually valid or to suspend the fiber if it isn't, except for normal register checks such as register scoreboarding. Register checks are enough for sequential computation, because program instructions are in strict sequential order, which is enough to guarantee data dependences are satisfied. However, if `fib(3)` and `fib(2)` run concurrently, their results are returned at indeterminate times, and `fib(4)` must have some way to know that both values have returned before it can add them. Even if `fib(4)` were to call one of the child functions sequentially and fork the other on another processor, it would still need to stall after the sequential return and wait for the return of the value from the parallel function.

The solution is to split the function into several fibers, *each* of which can run non-preemptively. Since the recursive function calls take indeterminate time, the function body should be split after these calls, into two fibers. The first fiber (f_0) executes statements S_1–S_4, that is, tests `n` and either returns 1 or invokes the children. The second fiber (f_1) executes statement S_5, adding the values produced by the children and returning the sum to its parent. Figure 4.4 shows the threaded version of the call graph for `fib(4)`. Each instance of `fib` has been replaced by a pair of fibers f_0 and f_1.

This example shows a tight coupling between f_0 and f_1. Every instance of f_1 must have been preceded by a corresponding instance of f_0, and most instances of f_0 (except for leaves) lead to a corresponding instance of f_1.[4] Furthermore, paired instances of f_0 and f_1 need to share some data. For example, both f_0 and f_1 need access to the partial result variables `left` and `right`. This example shows that the PXM requires something one level higher than an individual fiber.

This need leads to EARTH's *two-level thread hierarchy*. In the Fibonacci example of Figure 4.4, the `fib` routine is a *threaded procedure*, while f_0 and f_1 are *fibers* within this procedure. Threaded procedures and fibers differ in their contexts, their

[4]Throughout this work, an *instance* of a fiber or procedure is a specific dynamic instantiation of a given fiber or procedure, with its own context.

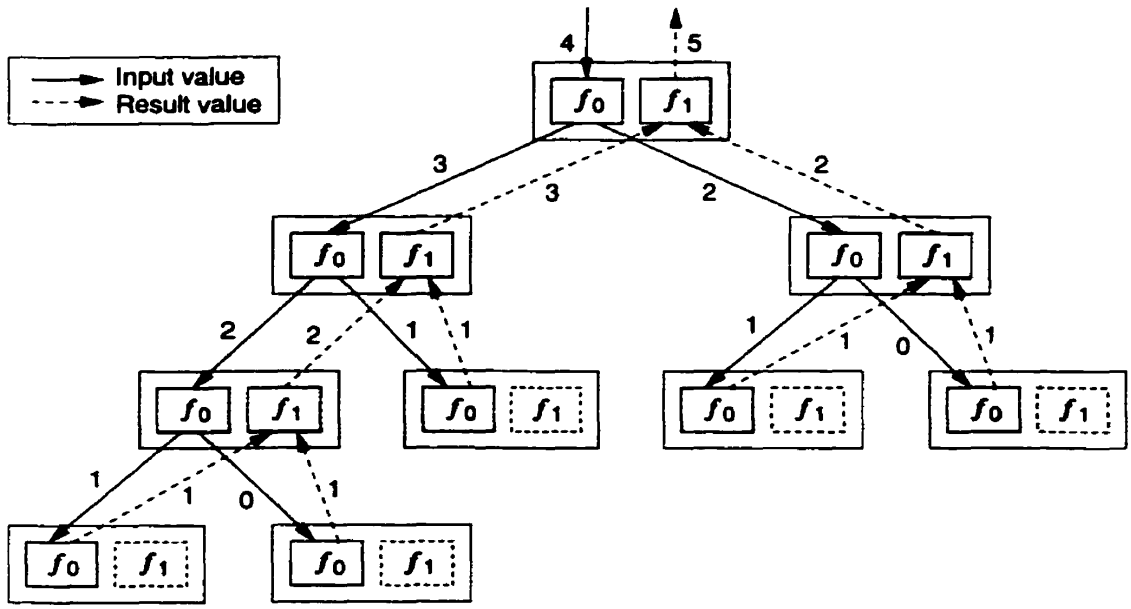


Figure 4.4: Threaded Fibonacci

lifetimes, and their manner of invocation.

The context of an instance of a threaded procedure is similar to the context of a function call in a conventional language such as C.[5] This context includes both local variables and parameters passed to the procedure. Both are accessible by all fibers contained within the threaded procedure. Variables and parameters persist from one fiber to the next, and thus can be used for exchanging values between fibers within the same procedure instance. Fibers have a much smaller context, consisting only of registers and specialized state variables (such as condition codes). Thus, fibers are extremely lightweight, and can be entered and exited quickly, making them suitable for fine-grained tasks where the overheads of normal function context-switching would outweigh the costs of the computation performed. Register values do not persist beyond a fiber's termination, but a fiber can exchange values with other fibers in the same procedure by accessing variables in the procedure context.

Procedures are invoked explicitly by the application program. When the program invokes a procedure, the machine creates a context for this procedure, initializing the input parameters with the values passed to this procedure. On the other hand,

[5]The term "threaded procedure" is used, rather than "threaded function," because threaded procedures cannot return values, for reasons explained later.

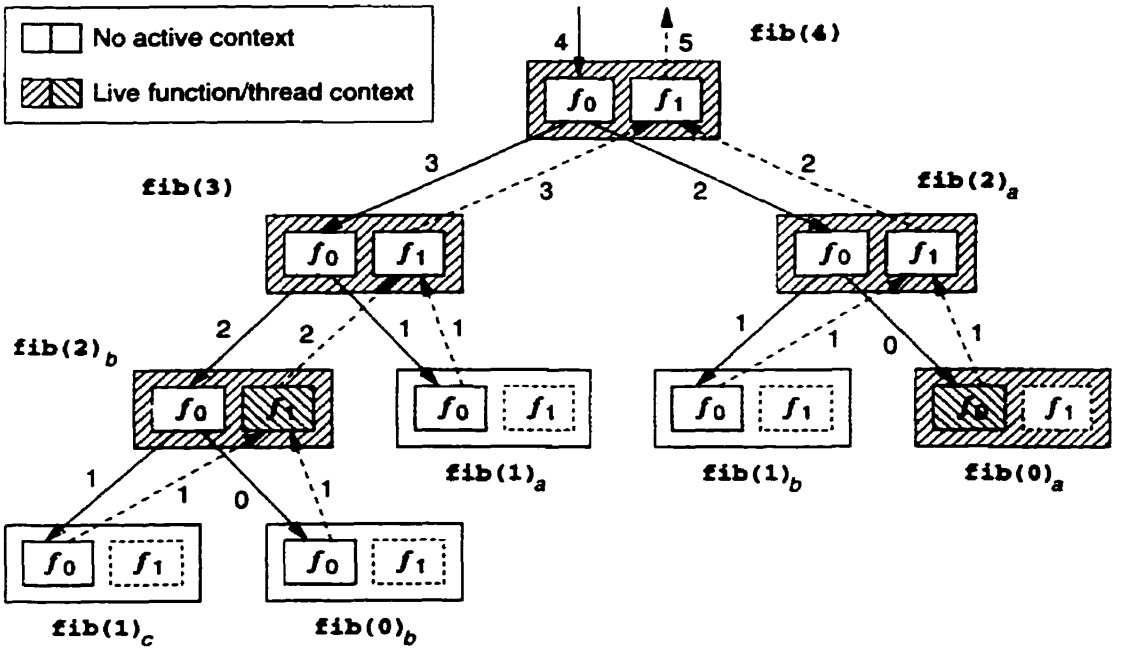


Figure 4.5: Snapshot of Threaded Fibonacci Context State

fibers are started automatically, using a mechanism described in Section 4.1.1.4. Once a fiber finishes executing, the fiber and its context terminate and are removed from the processor. On the other hand, a threaded procedure instance remains "live" even if none of its fibers are active or enabled; a threaded procedure must explicitly terminate itself.

Figure 4.5 illustrates context lifetimes. For clarity, procedure instances are labeled, and multiple instances with the same arguments are subscripted a, b, etc. This figure shows one possible state of a two-processor EARTH computer executing `fib(4)`. The topmost instance of `fib` has invoked `fib(3)` and $\texttt{fib(2)}_a$. The left child `fib(3)` has invoked its children, and fiber f_1 of its first child $\texttt{fib(2)}_b$ is currently running on one processor. Thus, f_1's context (consisting of a few registers) is live. The other child's context is not active, either because it has not been initialized or because it has already run and terminated. Similarly, the other instance $\texttt{fib(2)}_a$ has one child whose context is still live and has a fiber f_0 running on the other processor. Thus, two fibers are currently active, and five procedure contexts are live. Three procedure instances currently have no fibers active, but fiber f_1 in each will run once data from their children have produced values.

Some objects may need to exist outside of a particular frame's context. Objects may have a lifetime beyond a single procedure, or may have a size which is dynamic or which can't be determined at the time a frame is allocated, or may be shared among multiple procedures. For this reason, the EARTH PXM includes static variables and a heap for allocating compound data structures such as arrays.

4.1.1.3 Threaded Procedures

In EARTH, a fiber is always part of an enclosing threaded procedure. All fibers in a procedure share the local variables and input parameters of that procedure. When a threaded procedure is invoked, a new frame is allocated in memory for this particular instance of the procedure. All fibers access this frame through a *frame identifier* (FID), which is part of a fiber's context and is normally kept in a register for quick access. This is similar to the frame pointer found in conventional block-structured languages. Given a frame identifier, it is possible to access any local variable or input parameter in the corresponding procedure instance's context.

However, there are several essential differences between EARTH procedures and conventional functions. The main differences are

1. Frame allocation
2. Invocation and scheduling
3. Parameter passing

Frame allocation: Conventional frames can be kept on a simple linear stack, due to their sequential execution. The state shown in Figure 4.5 would be impossible for a sequential implementation of Fibonacci because the two functions in the second generation, `fib(3)` and $\texttt{fib(2)}_a$, cannot be active at the same time. On a parallel machine, there can be two sibling procedures active at the same time, or even a parent and child procedure running on different processors. A linear stack would not work in either situation.

Therefore, EARTH frames are dynamically allocated from a *heap*. (This was first proposed for dataflow [100] and later used in other multithreaded systems [21, 91].) When a procedure is invoked, the machine must allocate an appropriately-sized block of memory from the heap, and initialize the input parameters with the values

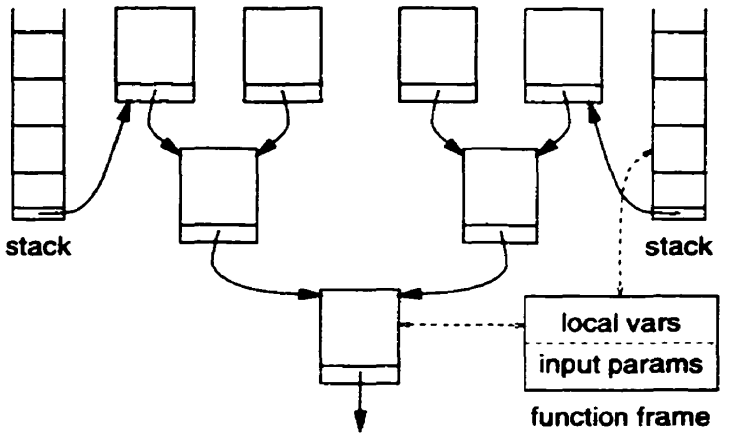


Figure 4.6: Tree of Procedure Frames with Sequential Stack

passed to the procedure, before any fibers in that procedure instance can start executing. Thereafter, the frame can be accessed via its frame identifier (FID). When a procedure terminates, the block is returned to the heap's free list. Because procedures are explicitly terminated, no garbage collection of frames is needed.

To allow the use of libraries and legacy code, the EARTH PXM also supports conventional *sequential functions*. A sequential function is called from a fiber in the same way it is called from another sequential function. Sequential functions use conventional call/return mechanisms, and a fiber which calls a sequential function suspends until the function returns. While this may appear to violate the non-preemptiveness property, the digression into the sequential code should be viewed as a part of the fiber itself. The sequential code is non-preemptive, and will return control to the calling fiber when completed. To guarantee this, no calls to threaded procedures are allowed within sequential code. Sequential functions require a stack since they may call other sequential functions. The stack is only needed while such sequential code is executing, and can be removed when the current sequential call is terminated. (How the stack is allocated depends on the implementation.)

Figure 4.6 illustrates frame allocation for a typical recursive program. Most threaded procedures need to communicate with other procedures, such as their callers. For this, they need references to the other procedures' frames or to variables within the frames. If each procedure has the identifier of the frame of its caller, a tree of procedure frames is generated as shown in the figure. Also shown are two stacks

allocated for sequential calls originating from two of the procedure instances, both of which are active. This example shows links from children to parents connecting the call structure together. Most purely recursive divide-and-conquer algorithms will show a similar structure, but the EARTH model is flexible enough to permit an arbitrary interconnection pattern among a set of concurrently executing procedures, such as peer procedures and producer-consumer relationships, as will be illustrated in later chapters.

Invocation and scheduling: When a sequential program calls a function, execution of the caller suspends at the point of the function call and does not continue until the caller returns. When EARTH code invokes a threaded procedure, the fiber invoking the procedure can continue to execute. This is one source of parallelism in EARTH programs, for invoking a procedure can turn one execution stream into two.

A threaded procedure consists of one or more fibers. One fiber in each procedure is called the *initial fiber* and has the special property that it is the first fiber to run when a procedure is invoked. When a procedure is invoked, the system must first allocate and initialize the frame, as described above. Once this is done, the initial fiber is enabled, meaning that as soon as there is spare processing power, the initial fiber can begin executing. All other fibers in the procedure must rely on other mechanisms (described in the next section) to become enabled. Once the initial fiber has finished, it cannot run again in this procedure instance.

Since all fibers within a threaded procedure share data in a single frame, all fibers in a given procedure instance must run on a processor or processors with access to the same context. Typically, locality considerations will require all fibers in a procedure instance to run on the same processor. However, such concerns do not apply to an invoked procedure, which can run on another processor if one is available. The implementation of EARTH (whose programmer interface is specified in the EVM) may allow the caller to select this processor, or the programmer can ask the EARTH system to assign processing resources to the procedure instance automatically. In the latter case, the system will attempt to assign it to lightly-loaded resources, with the goal of achieving a uniform load distribution. The choice of manual or automatic selection of a processor depends on the application; some regular applications have patterns which are easy to balance statically using manual selection, while

many irregular applications have a work distribution highly dependent on runtime conditions, requiring dynamic load balancing.

Parameter passing: Threaded procedures, like sequential functions, can pass scalar values to threaded procedures they invoke. However, a threaded procedure cannot return a value to its caller using a conventional value return mechanism. This is because, as stated in the previous section, a fiber which invokes a procedure continues to execute, while the semantics of the return statement imply that the caller waits until the callee finishes so that the value can be used. Therefore, a threaded procedure must send values to its caller using the same mechanism as is used to send data between fibers, which is covered in the next section.

4.1.1.4 Fibers and Synchronization

In EARTH's two-level thread hierarchy, fibers are the smallest unit of scheduling above individual instructions. Fibers are non-preemptive, so a threaded procedure should be divided into fibers wherever there are likely to be long or unpredictable latencies. This includes threaded procedure invocations, as in the Fibonacci example, and the fetching of data from remote processors.

Each fiber is part of an enclosing threaded procedure. A given instance of a fiber is associated with one particular procedure instance. To maintain this association, the context of a fiber includes a reference to the frame (FID) for this procedure instance, normally kept in a register which is loaded once when the fiber begins execution. The frame identifier (FID) is essential, for it prevents two active instances of a fiber from accessing the same context. For instance, the two active fibers in Figure 4.5 are supposed to read different copies of the input parameter n, and this is guaranteed because they have different frame identifiers.

Within the code of a threaded procedure, each fiber is given a unique identifier called the *instruction pointer* (IP). Each *instance* of a fiber can be uniquely identified by a pair (FID,IP) consisting of a frame identifier, to uniquely identify the frame for the particular instance of the procedure containing this fiber, and an instruction pointer, to specify the fiber within the procedure. As a single unit, the pair (FID,IP) is called a *fiber identifier*.

When a program is divided into procedures and fibers, there will exist (in most cases) data and control dependences among the fibers and procedures. Because the

EARTH model can allow fibers to be scheduled in arbitrary sequences, the machine must check and verify dynamically that all data and control dependences have been satisfied before enabling a fiber. This is done explicitly using a mechanism adapted from the static dataflow machines reviewed in Section 3.1.2.

Control and data dependences are made explicit in the EARTH code using *synchronization signals* and *synchronization slots*. A synchronization (or *sync*) signal is sent from one fiber to another, either in the same or another procedure instance, to tell the recipient that a specific control or data dependence has been satisfied. For instance, the sending fiber may have produced data required by the receiving fiber; since the latter could not run before that data was produced, the producer must tell the consumer that the data is now ready. If a fiber depends on more than one datum or control event, it needs to be sure that *all* dependences have been satisfied before it is enabled, since the fiber can't be preempted once started. A counter is used to count the incoming signals so it is known when a fiber is ready to be enabled.

In the Fibonacci example, each non-leaf instance of `fib` receives two integers from procedures that it invokes. Since fiber f_1 adds these integers, it needs both before it can start. Therefore, a count of two is associated with f_1 at the time the procedure instance is initiated. As each datum arrives from a child procedure, the count is decremented. When the count reaches 0, f_1 has all the data it needs and can be enabled.

In static dataflow (see Section 3.1.2), a count is associated with each actor. Rather than associating a counter with each fiber, the EARTH model separates the counters from the fibers, which allows reuse of the counters for controlling different fibers at different times, or allows several counters to control the same fiber.[6] Counts are maintained in synchronization (or sync) slots. A sync slot is a triple (SC, RC, IP) containing respectively a *sync count,* a *reset count,* and an *instruction pointer*. The instruction pointer (IP) binds the sync slot to one of the fibers in the procedure. The sync count indicates the number of sync signals that have to be received by the sync slot before the specified fiber can be enabled. When a sync signal is received, the sync count is decremented. If the count reaches 0 the fiber specified by the IP is enabled and the sync count is set back to the value of the reset count. The use of a reset count allows fibers to be enabled multiple times, as explained later.

[6] An example of the latter is in Section 6.2.3.

Because sync counts are used for controlling the enabling of fibers, they must persist beyond the lifetime of a fiber, and therefore must be part of the context of a procedure rather than a fiber. They are, in fact, a type of local variable. However, as is shown later, there are some strict ordering constraints between manipulation of sync slots and the sending of sync signals. To ensure portability of the EARTH model across different implementations, sync slots should *not* be accessible to the user, except through special instructions, defined as part of the EVM, which guarantee these ordering constraints are obeyed. Since a sync slot is bound to a particular procedure instance, it can only control fiber instances within that procedure instance. Thus, the third element of a slot only needs to be an IP, rather than a complete fiber identifier, because the FID is fixed.

In the Fibonacci example, each non-root invocation of `fib` produces a value which must be sent to its parent. The sync slot merely counts the number of integers received; there has been no discussion yet of data transfer. Sending a sync signal is sufficient if both sender and receiver are in the same procedure instance, since local variables can be used to transfer the data from one fiber to another. However, if sender and receiver are in different procedure instances, special mechanisms may be needed to transfer the data, particularly if sender and receiver are on different nodes in a distributed memory machine.

The EARTH PXM provides *atomic* operations for sending data and a sync signal together, which guarantees that the data has been properly transferred before any fibers are enabled as a result of the sync. Such a data-transfer/sync-signal operation may be initiated by the *producer* of data, which sends local data to another location, or by the *consumer* of data, which sends a request for remote data to the system, which retrieves the data and copies it into a local location before signaling the sync slot. The latter operation is called a *split-phase transaction* because the request and data transfer may occur in distinct phases.

Figure 4.7 shows a graphical representation of an EARTH implementation of parallel Fibonacci, which would produce a call graph like Figure 4.4. The code is written in a C-like pseudocode since no programming language or EVM for EARTH has yet been presented. The new `fib` procedure takes, in addition to the index parameter `n`, two references **`result`** and **`done`**, which refer to an integer and a sync slot, respectively. After `fib(n)` computes the n^{th} Fibonacci number, it sends the answer to the location referenced by **`result`**, and sends a sync signal to the slot

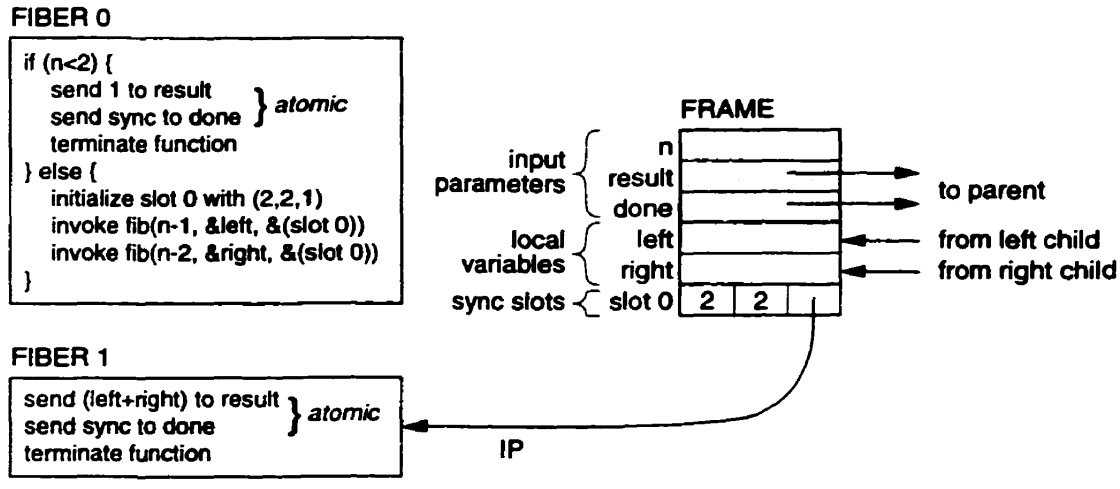


Figure 4.7: Thread Graph for Fibonacci

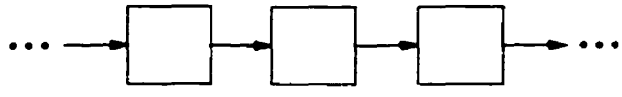


Figure 4.8: Pipelined Program Structure

referenced by **done**, bundling the two in an atomic operation. The local variables include two integers for holding intermediate results. if `fib` is not a leaf, it invokes two instances of itself and passes references to these integer locations to the children. A single sync slot is initialized with a count of 2, and a reference to this slot is passed to each child. Thus, when the recursive values are computed, they are stored in the caller's local variables; after the second value arrives, fiber 1 is enabled.

The Fibonacci example has been useful for introducing the EARTH PXM, but it is a trivial example which is easily supported by parallel paradigms besides EARTH. The EARTH thread model can be used to partition programs in a way suited to a particular application. Fibonacci is a simple example of binary recursion, in which each fiber is only run once. But many applications require other program structures, such as producer-consumer. For example, the pipeline structure in Figure 4.8 illustrates an application in which each of a set of modules reads data from its input, transforms it, and sends the result to the next module.

In this application, it is assumed there is a large set of inputs, and parallelism is

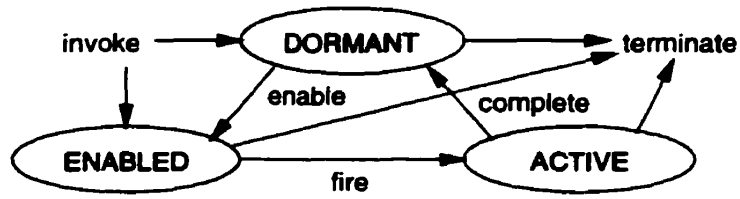


Figure 4.9: EARTH Fiber States

achieved by running the modules concurrently. Each module needs to run repeatedly in order to process multiple data sets. Otherwise, one or more procedures need to be invoked for each input datum. This could impose large overheads, especially if a lot of state information needs to be passed to a procedure to get it started. It would be better to invoke the procedures once, and allow a fiber to start again whenever new data arrives. Fortunately, this is possible in EARTH.

The complete state diagram for an EARTH fiber is shown in Figure 4.9. When a procedure is invoked and initialized, all fibers but the initial fiber begin in the dormant state; the initial fiber begins in the enabled state (which is why the "invoke" arc leads to both states). Any active fiber in this procedure instance can initialize sync slots in its frame. Once a sync slot is initialized, any fiber can send sync signals to that slot. If a sync count reaches 0, the fiber whose IP is in the slot is enabled. An enabled fiber is fired (moved to the active state) when processing resources are available to start executing the fiber. After an active fiber is completed, the fiber returns to the dormant state, *where it can be subsequently re-enabled if the sync count returns to 0 again.* Any active fiber can issue a command to terminate the entire procedure, which causes all fibers in that procedure to be removed and the frame deallocated.

It is the ability to re-enable a fiber that has already run that makes producer-consumer synchronizations possible. Fibers can be pipelined in a manner similar to the pipelining used in dataflow graphs in Section 3.1.1. When a fiber has finished processing data and has sent its output to the next module, the fiber terminates; since its sync count was reset, it can be re-enabled when new input data arrives and synchronizes the same sync slot. A real benchmark using this technique is described in a later section.

4.1.1.5 Benefits of the EARTH Thread Model

The beginning of this chapter claimed that the EARTH model addresses issues that affect parallel processing as listed in Section 1.2: latency, bandwidth, synchronization, programmability, and manufacturability. Having presented the basic model, we can now show in general how features of the EARTH PXM address these issues.

Latency: The multithreading in EARTH does not eliminate interprocessor latencies, but allows computers to *tolerate* latency by performing useful computations while long-latency operations are in progress. This is done by ensuring that the initiators of such operations and the recipients of the data produced by these operations are in different fibers, allowing other fibers to run while these operations are being performed.

Bandwidth: Latency-hiding is not enough to guarantee a high degree of processor utilization. If too many fibers issue remote reads or writes, the network will become saturated and limit overall system performance. While good hardware is important in applications with high communications requirements, the PXM can play a major role as well. The use of frames local to procedure instances encourages writing code with locality in mind. Requiring all dependences to be satisfied before a fiber is enabled may encourage the movement of data in blocks rather than in smaller units, causing network transfer overheads to be amortized. The separation of data communication from the production and consumption of this data removes data transfer from the critical path (because the processor can switch to another fiber), which may give the system's transport layer opportunities to optimize network utilization (e.g., by combining data transfers).

Synchronization: The EARTH PXM separates synchronizations from the computation, allowing synchronizations to be performed independently, possibly by a separate unit. Processors are not blocked waiting for other events to occur, as long as there are other fibers eligible to run. The fiber synchronization mechanism enables the use of non-preemptive fibers, limiting context-switching costs to the boundaries of fibers, where they can be more easily controlled. Unlike dataflow machines, synchronization costs are contained by restricting the instructions within fibers to sequential ordering, where traditional superscalar

techniques are effective at achieving limited ILP.

Programmability: Section 1.2.4 argued that solving the three preceding problems can go a long way toward making parallel machine easier to program. The EARTH PXM further contributes to ease of programming by providing different ways of expressing parallelism, such as divide-and-conquer or producer-consumer, according to what is best for the application. EARTH allows tasks to be divided into an arbitrary and dynamically varying number of procedure instances, and supports automatic load balancing of these instances, which is a boon especially to irregular applications.

Manufacturability: EARTH was designed for efficient execution by off-the-shelf processors. The properties of EARTH fibers (non-preemptiveness, sequential execution, and atomic scheduling) eliminate the need for fine-grain switching between multiple concurrent contexts (as is proposed for some multithreaded processors) and make conventional sequential processors suitable for EARTH.

4.1.1.6 Summary of the EARTH Thread Model

Under the EARTH model, the instructions of a program are divided into three layers:

1. Threaded procedures,
2. Fibers,
3. Individual instructions.

The upper two layers form EARTH's two-layer thread hierarchy. Each layer defines ordering constraints between components of that layer and a mechanism for determining a schedule which satisfies those constraints.

Individual instructions are at the lowest level. Instructions obey sequential execution semantics, where the next instruction to execute immediately follows the current instruction unless the order is explicitly changed by a branch instruction. Instruction-level parallelism among these instructions is allowed so long as it is consistent with this semantics.

Fibers are collections of instructions sharing a common context, consisting of a set of registers and the identifier of a frame containing variables shared with other

fibers. When a processor begins executing a fiber, it executes the designated first instruction of the fiber; subsequent instructions within the fiber are determined by the instructions' sequential semantics. Branch instructions are allowed, but only to other instructions within the same fiber. Calls to sequential procedures are also permitted; such procedure calls are part of the fiber, by definition, even if the code is defined elsewhere. A fiber finishes execution when an explicit fiber-termination marker is encountered. The fiber's context remains active from the start of the fiber to its finish.

Fibers are non-preemptive; once a fiber begins execution, it is not suspended, nor is its context removed from active processing. Thus, fibers are scheduled atomically. A fiber is "enabled" (made eligible to begin execution as soon as processing resources are available) when all data and control dependences have been satisfied. Sync slots and sync signals are used to make this determination. Sync signals (possibly with data attached) tell the recipient that a dependence has been met. A sync slot records how many dependences remain unsatisfied; when this count reaches 0, a fiber associated with this sync slot is enabled, for it now has all data and control permissions necessary to execute. The count is reset to allow a fiber to run multiple times.

Threaded procedures are collections of fibers sharing a common context which persists beyond the lifetime of a single fiber. This context consists of a procedure's input parameters, local variables, and sync slots. The context is stored in a frame, dynamically allocated from the heap when the procedure is invoked. Threaded procedures are explicitly invoked by other procedures. When a threaded procedure is invoked and its frame is ready, the initial fiber is enabled, and can only run once. Other fibers in the same procedure instance may only be enabled using sync slots and sync signals. An explicit terminate command is used to terminate both the fiber which executes this command and its procedure instance, which causes the frame to be deallocated. Since procedure termination is explicit, no garbage collection is needed for these frames.

4.1.2 EARTH Memory Model

As mentioned in Section 1.2.4, parallel programming is hobbled by the lack of a universal program execution model. One of the fundamental issues dividing parallel

machines is the sharing of memory. Should memory be shared among all processors, allowing a processor to access any memory location in the system, or does each processor have its own private memory, inaccessible to others? To maximize portability, the EARTH model does not take a firm stand on this issue.

Most large multiprocessors use either *distributed memory* or *distributed shared memory*. Distributed memory architectures have separate memories for each processor, or group of processors. These are separate both *physically* and *logically*. A process can access data in remote memory only indirectly, by communicating with a process that has access to that memory. For this reason, they are sometimes called *message passing* machines. Distributed shared-memory architectures have a *global* address space, which allows a processor to access any memory location in the system. Unlike centralized systems, the memory here is distributed among the processors, with some memory physically close to a processor and the rest with other processors. These are also called *Non-Uniform Memory Access* (NUMA) machines. Conventional wisdom says that distributed memory scales more easily than distributed shared memory, while shared memory computers are generally easier to program.

The minimal requirements of the EARTH PXM fall somewhere between these two options. To support the EARTH PXM, a machine's memory system must have the following properties:

1. An active fiber must have direct, low-latency access (through load and store operations) both to its private fiber context and to the frame it shares with other fibers in the same procedure instance.

2. An active sequential function call must have direct, low-latency access both to its local linear stack and to the frame belonging to the fiber which initiated the sequential function call (either directly or through other function calls).

3. Instruction pointers are uniform throughout the system. The code for all threaded procedures and sequential functions is accessible from all processing elements of the machine, and a given IP value has the same meaning on all such elements. The instruction addresses used in sequential function calls must be the same on all processing elements.

4. All objects in the EARTH system which may be accessible by more than one

threaded procedure, including frame identifiers, sync slots, and any data addresses which may be bound with sync signals (as described in Section 4.1.1.4) must be *globally unique* and accessible by special *EARTH operations.*

The first two requirements are straightforward and ensure rapid execution within a fiber. Requirement (1) would imply, for instance, that on a distributed-memory machine, a threaded procedure instance can run on only one processor. Two fibers from the same procedure instance could not run on different processors because they would not be able to share frame data. Requirement (2) ensures that sequential functions are entered and exited quickly; the second part ensures a rapid return to the initiating fiber.

The third item requires code to be loaded on (or at least available to) all processors. This guarantees that procedures and sequential functions can be invoked on any processing element without any restrictions (other than the first two points above).

The last requirement is critical to the EARTH model, for it allows fibers to communicate with other procedures. Consider the Fibonacci example of Figure 4.7. When the `fib()` procedure invokes children, it passes references to local variables to its children, and the children write their results to the referred locations using an atomic data-transfer/sync-signal operator. If a child is invoked on another processor, as EARTH allows, then when it does this transfer, the destination argument will refer to a non-local location. This reference must be meaningful on the processor running the child.

Therefore, memory references, at least those passed between procedures, must be globally unique across the machine, and each processor must be able to determine the exact location of any given memory reference. However, note that the requirement above only says that the address must be accessible *to the EARTH operation*; it does not say that the address must be accessible by normal load/store operations. Furthermore, accesses are not expected to be fast, as with the local loads and stores in requirements (1) and (2). These "remote" accesses may have long latencies, but will not necessarily stall the processor because other fibers may run during the accesses.

If loads and store instructions to remote addresses are not required, then it is trivial to form a globally-shared address space by combining a local address with a node identifier. Such a pair can be used whenever an EARTH operator expects

a reference. A processor or router that needs to forward the message extracts the node identifier to find the destination of the message, while an SU that accesses memory extracts the local address. Therefore, *if a program, based on the EARTH PXM, accesses non-local memory locations solely through EARTH synchronizing operations, that program can be supported on a distributed memory computer.*

However, it should be emphasized that EARTH's PXM is neither restricted to distributed memory machines nor limited by them. Some distributed shared-memory machines use local caches, driven by complex coherence protocols, to reduce the costs of remote access (*Cache Coherent NUMA* or CC-NUMA architectures). However, even with CC-NUMA machines, there are situations where EARTH's model of fiber synchronization would bring benefits. Shared memory machines still need to enforce data and control dependences, and EARTH provides a simple mechanism which is independent of any memory system. Also, remote latencies will still exist in a shared memory computer even with the addition of local caches.

For example, if the producer of some data signals a remote consumer that the data is available, the producer likely has updated its copy recently, meaning the consumer's cache copy is probably stale. If the consumer tries to read the stale data, the load will miss and stall pending the completion of the remote fetch. If, on the other hand, the producer sends the data to the consumer using an atomic data-transfer/sync-signal operator, the data is guaranteed to be in the consumer's local memory, and no miss is possible. Similarly, explicit fetches by the consumer, if they are likely to miss, can be made on an EARTH machine in a split-phase manner, with the data being used in another fiber. This would allow the local processor to continue execution (or switch to another fiber) in spite of the miss.

The two EVMs presented in the next chapter are designed to work for a distributed memory machine with a globally-shared address space. But these will also work for a shared memory architecture. In fact, shared memory might improve performance by decreasing the fetch time of split-phase transactions.

4.1.3 EARTH Operations

The previous sections presented an abstract model for representing a program as a hierarchy of instructions, fibers, and threaded procedures whose executions are coordinated using sync slots and sync signals. A model for the execution of a

thread-based program was discussed. This execution relies on various operations for sequencing and manipulating the fibers in this hierarchy. These operations perform the following functions:

1. Invocation and termination of procedures and fibers;
2. Creation and manipulation of sync slots;
3. Sending of sync signals to sync slots, either alone or atomically bound with data.

Some of these functions are performed automatically, generally as a result of other EARTH operations. For instance, the sending of a sync signal to a sync slot with a current sync count of 1 causes the slot count to be reset and a fiber to become enabled. Eventually, that fiber becomes active and begins execution. But some operations, such as procedure invocation, are explicitly triggered by the application code. The purpose of this section is to list and define eight explicit (program-level) operations which are essential to a machine implementing the EARTH thread model, and argues why these are essential.

This is still a part of the description of the EARTH PXM, so operations here are still defined at an abstract level. The goal here is to define a common set of primitive operations that must be present in any machine that supports the EARTH PXM. The arguments are defined in general and minimal terms. An EVM, if desired, may include a richer variety of operators or add more arguments to operators to improve efficiency or expressiveness. The next chapter presents two different EARTH Virtual Machines, one of which is used for the experimental studies in this work.

The following data types and functions are used by the eight operators:

FID A frame identifier is a unique reference to the frame containing the local context of one procedure instance. It is possible to access the local variables, input parameters, and sync slots of this procedure, as well as the procedure code itself, using the FID, in a manner specified by the EVM. The FID is globally unique across all nodes; no two frames, even if on different nodes, have the same FID simultaneously.

IP An instruction pointer is a unique reference to the start of the code of a particular fiber within a particular threaded procedure.

It does not specify a specific *instance* of the fiber. (An FID and IP combined do this.)

PP A procedure pointer is a unique reference to the start of the code of a particular threaded procedure (but not a specific *instance*). Through this reference, the computer must be able to access all information necessary to start a new instance of a procedure.

SS This is a reference to a unique synchronization slot, consisting of a sync count (SC), reset count (RC), and fiber (F). The first two are non-negative integers and the third is an IP. The expression $SS.SC$ refers to the sync count of SS, etc. However, this is for descriptive purposes only; these fields should not be manipulated by the application program except through the special EARTH operators listed below. Each SS is associated with a specific instance of a procedure, hence a particular FID, which can be referred to as $FID\_of(SS)$.

The SS type must include enough information to identify a single sync slot which is unique across all nodes. How much information is required depends on the operator and the EVM. In some cases, the sync slot may be restricted to a particular frame, which means that only a number, identifying the slot within that frame, is needed. In other cases, a complete global address is required (such as a pair consisting of a frame ID and a sync slot number).

T In the list of EARTH operators, type T means an arbitrary object, either scalar or compound (array or record). This class of objects can include any of the reference data types listed above (FID, IP, PP, SS), so that these objects can also be used in EARTH operations (e.g., they can be transferred to another procedure instance). Type T can also include any instance of the reference data type that follows.

reference-to-T For each object of type T, there is a reference to that object, of type reference-to-T, through which that object can be accessed

or updated. In accordance with the memory requirements in Section 4.1.2, this must be globally unique and all processing elements must be able to access the object of type T using the reference. The term "reference" is used, instead of "pointer" or "address," to prevent any unwarranted assumptions about the kinds of operations that can be performed with these references.

The following lists the eight operations, describing the role of each operation, why it is important, and the behavior which must be supported by the EVM. The list also suggests options which might be added in the EVM. In the list, the "current fiber" is the fiber executing the operation, and the "current frame" is the FID corresponding to the current fiber.

4.1.3.1 Thread Control Operations

Thread control operations control the creation and termination of threads (fibers and procedures) based on the EARTH thread model described in the previous section. The primary operation is procedure invocation, as illustrated in the Fibonacci example. There must also be operators to mark the end of a fiber and to terminate a procedure. No explicit operators to create fibers are needed, as fibers are enabled implicitly. One fiber is enabled automatically when a procedure is invoked, and others are enabled as a result of sync signals.

A program compiled for EARTH must designate one procedure which is automatically invoked when the program is started. Only one instance of this procedure is invoked, even if there are multiple processors. Other processors remain idle until procedures are invoked on them. This distinguishes EARTH from parallel models such as SPMD, where identical copies of a program are started simultaneously on all nodes.

INVOKE(PP *proc*, **T** $arg1$, **T** $arg2$, ...)

This operator invoke procedure *proc*. It allocates a frame appropriate for *proc*, initializes its input parameters to $arg1, arg2$, etc., and enables the IP for the initial fiber of *proc*. The EVM may set restrictions on what types of arguments can be passed, such as scalar values only. The system must guarantee that the frame

contents, as seen by the processing element which executes *proc*, are initialized before the execution of *proc* begins.

TERMINATE_FIBER()

This terminates the current fiber. The processing element which ran this fiber is free to reassign the processing resources used for this fiber, and to begin execution of another enabled fiber, if one exists. (If there are none, the processing element waits until one becomes available, and begins execution.)

TERMINATE_PROCEDURE()

This is similar to TERMINATE_FIBER(), but it also terminates the procedure instance corresponding to the current fiber. The current frame is deallocated. This description does not specify what happens to any other fibers belonging to this instance if they are active or enabled, or what happens if the contents of the current frame are accessed after deallocation. The EVM may define behavior which occurs in these cases, or define such an occurrence as an error which is the programmer's responsibility to avoid.

4.1.3.2 Sync Slot Control Operations

Sync slots are used to control the enabling of fibers and to count how many dependences have been satisfied. They must be initialized with values before they can receive sync signals. It would be possible to make sync slot initialization an atomic part of procedure invocation. But our experiences with programming multithreaded machines have shown that the number of dependences may vary from one instance of a procedure to the next, and may depend on conditions not known at compile time (or even at the time the procedure is invoked). Therefore, it is preferable to have an explicit operation for initializing sync slots. (Of course, a particular *implementation* of EARTH may optimize by moving slot initialization into the frame initialization stage if the initialization can be fixed at compile time.)

INITIALIZE_SLOT(SS *slot*, **int** *SC*, **int** *RC*, **IP** *F*)

This initializes sync slot *slot*, giving it a sync count of SC, a reset count of RC, and an IP of F. Only sync slots in the current frame can be initialized (hence, no FID is required). Normally, sync slots are initialized in the initial fiber of a procedure. However, an already-initialized slot may be re-initialized, which allows slots to be reused much like registers. (Reusability is explored in a related architecture project which is based on EARTH [86].)

There is the potential for race conditions between the initialization or re-initialization of a thread and the sending of sync signals to that thread. The EVM and implementation should guarantee sequential ordering between slot initialization and slot use within the same fiber. For instance, if an INITIALIZE_SLOT operator which initializes *slot* is followed in the same fiber by an explicit sending of a sync signal to *slot*, the system should guarantee that the new values in *slot* (placed there by the initialization) are in place before the sync signal has any effect on the slot. On the other hand, it is the programmer's responsibility to avoid race conditions *between* fibers. The programmer should also avoid re-initializing a sync slot if there is the possibility that other fibers in the system may be sending sync signals to that slot.

INCREMENT_SLOT(SS *slot*, **int** *inc*)

This operator increments *slot.SC* by *inc*. Only slots in the local frame can be affected. The ordering constraints for the INITIALIZE_SLOT operator apply to this operator as well.

Although not mandatory for the EARTH thread model, this is a very useful operation for procedures where the number of dependences is not only dynamic, but cannot be determined at the time a sync slot would normally be initialized. An example is traversing a tree where the branching factor varies dynamically, such as searching the future moves in a chess game, where the number of moves to search at each level is determined at runtime. Such an application could be coded in the following way (pseudocode used):

Initial fiber	Fiber 1

```
...                                    ...
INITIALIZE_SLOT(s,1,1,Fiber_2);        m = 0;
...                                    for (all potential moves) {
                                         if (potential move is legal) {
                                           INCREMENT_SLOT(s, 1);
                                           INVOKE(search_move,
                                                  &results[m++], s, ...);
                                         }
                                       }
                                       INCREMENT_SLOT(s, -1);
                                       ...
```

In this code, an array is allocated for holding result data, and each child is given a reference to a different location to which the results of one move are sent. Each children sends a sync signal to sync slot **s**, and fiber 2, which chooses a move from among all the sub-searches, should be enabled when all children are done. Since the number of legal moves varies from one instance to the next, the total number of procedures invoked is not known when the slot is initialized in the initial thread. The INCREMENT_SLOT operation is used to add one to the sync count in **s** before invoking a child. If, after the first child is invoked, the child sends a sync signal back before the loop in fiber 1 performs another INCREMENT_SLOT, the count **s**.*SC* could decrement to 0, prematurely enabling fiber 2. To avoid this possibility, the count should start at 1, ensuring that the count is always at least 1 provided the slot is incremented before the INVOKE occurs. When all increments have been performed, it is safe to remove this offset, after which the last child to send a sync signal back will trigger fiber 2. An INCREMENT_SLOT with a negative count (-1) does this. (Actually, a SYNC operation, covered next, would have the same effect, and might be more efficient since it doesn't need an argument.)

4.1.3.3 Synchronizing Operations

The synchronizing operators give EARTH the ability to enforce data and control dependences between procedures, even those not directly related, enabling the programmer to create many parallel control structures besides simple recursion. Thus,

the programmer can tailor the control structures to the needs of the application.

These operators manipulate sync slots as described in Section 4.1.1.4. Each of the following operators sends a sync signal to a specified slot, which is used to count dependences. However, there may be fibers which only need to wait for one datum or control event, which would imply a sync slot with a reset count of 1. For such cases, the EVM may define special versions of the operators which enable the fiber directly rather than going through a sync slot, saving time and sync slot space. These are optional, however, as the same effect can be achieved with regular sync slots.

The static dataflow machines in Section 3.1.2, the designs which inspired EARTH's synchronization mechanism, had to avoid the erroneous condition in which two tokens try to exist on the same arc at the same time. There are analogous race conditions in EARTH. One example is enabling a fiber while another instance of the same fiber in the same procedure instance is active or enabled. This could result from having initialized the sync slot with too low a sync count, or, conversely, having generated too many sync signals for that slot. Another error is sending a sync signal to a slot before the slot has been initialized. The exact behavior is left undefined at this level. A specific EVM may specify behavior, or even provide a non-erroneous interpretation for such an occurrence, or indicate to what extent the programmer is responsible to avoid such conditions.

The first case is not, strictly speaking, an error under the EARTH PXM, but will work properly only under special conditions. Figure 4.10 illustrates the situation arising from having two instances of the same fiber in the same procedure instance simultaneously active. Technically, each fiber has its own context, so it would be possible for the two to run concurrently without interfering with each other. However, note that they still share the same frame, and any input data they require must come from this frame, either directly (the data is in the frame itself) or indirectly (a reference to the data is in the frame), since all local fiber context except the FID itself come from the frame. If both fibers copy the same data and references, they will operate redundantly. If each loads its initial register values from values in the frame and then updates the frame values, it is possible for the fibers to work concurrently on independent data. The figure shows each fiber working with a different element of an array `x`; the snapshot shows the state after each fiber has copied the reference to register `r2`. But correct operation of this code under all circumstances

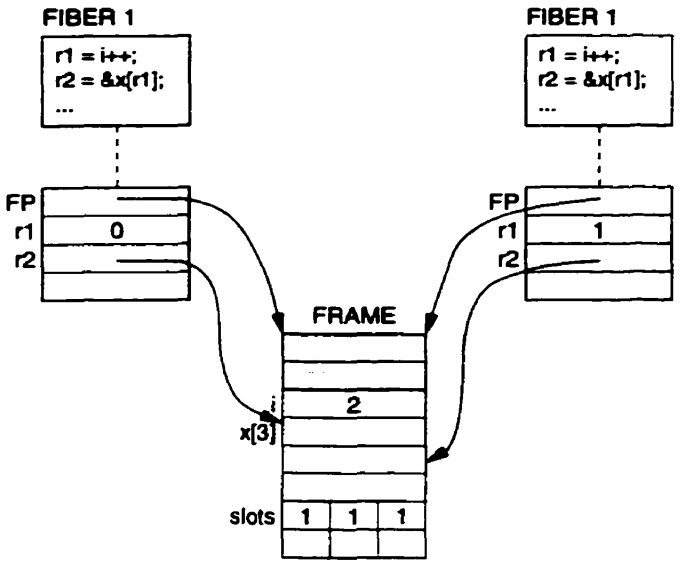


Figure 4.10: Two Instances of Same Fiber

requires two conditions:

1. If the hardware allows the two fibers to run concurrently, it must support atomic access to the frame variable `i`, e.g., a fetch-and-add primitive. These are not universal among processors and may require expensive OS solutions.

2. If the fibers were triggered by separate sync signals bound with atomic data transfers (note the first slot in the frame has a count of 1 and triggers fiber 1), the two producers of the data (assume in this case that it is sent to `x[]`) must be programmed to send the two values to separate locations in `x[]`.

While it is possible to program around these restrictions, it would be better if these situations were handled explicitly in the PXM. Section 4.3 discusses some simple extensions to the EARTH PXM which would simplify this example.

Three basic synchronizing operations are offered by the EARTH PXM: synchronization alone, and both producer-oriented and consumer-oriented versions of synchronization bound with data transfers.

SYNC(SS *slot*)

This is the basic synchronization operator. The count of the specified sync slot ($slot.SC$) is decremented. If the resulting value is 0, the fiber ($FID\_of(slot), slot.F$) is enabled, and the sync count is updated with the reset count $slot.RC$. Otherwise, the sync count is updated with the decremented value. The implementation must guarantee that the test-and-update access to the SC field is atomic, relative to other operators that can affect the same slot (including the slot control operators).

SYNC_WITH_DATA(T *val*, **reference-to-T** *dest*, **SS** *slot*)

Section 4.1.1.4 mentioned the importance of binding data transfers with sync signals, to avoid a race condition in which a sync signal indicates the satisfying of a data dependence and enabled a fiber *before* the data in question has actually been transferred. This binding is done in EARTH by augmenting a normal SYNC operator with a datum and a reference. The system copies the datum *val* to the location referenced by *dest*, then sends the sync signal to *slot*.

The system must guarantee that the data transfer is complete before the sync signal is sent to *slot*. More precisely, the system must guarantee that, at the time a processing element starts executing a fiber enabled as a direct or indirect result of the sync signal sent to *slot*, that processor sees *val* at the location *dest*. (A direct result means that the sync signal decrements the sync count to 0, while an indirect result means that a subsequent signal to the same slot decrements the count to 0.) The system must also guarantee that, after the sync slot is updated, it is safe to change *val* (this is mostly relevant if *val* is passed "by reference," e.g., as is usually done with arrays).

SYNC_WITH_FETCH(reference-to-T *source*, **reference-to-T** *dest*, **SS** *slot*)

The final operator of the EARTH set also binds a sync signal with a data transfer, but the direction of the transfer is reversed. While the previous operator takes a value as its first argument, which must be locally available, the SYNC_WITH_FETCH specifies a location which can be anywhere, even on a remote node. A datum of type T is copied from the source to the destination. The ordering constraints are the same as for SYNC_WITH_DATA, except that *val* (in

the previous paragraph) now refers to the datum referenced by *source*.

This operator is primarily used for fetching remote data through the use of *split-phase transactions*. If a procedure needs to fetch data which is likely to be on another node, the fiber initiating the fetch should not wait for the data, which may take a long time. Instead, the consumer of the data should be in another fiber, with a SYNC_WITH_FETCH used to synchronize a slot and enable the consumer when the data is received. The next chapter has examples of code using split-phase transactions.

This operation is considered "atomic" only from the point of view of the fiber initiating the operation. In fact, the operation typically occurs in two phases: the request is forwarded to the location of the source data (on a distributed-memory machine), and then, after the data has been fetched, it is transferred back to the original fiber. The SS reference is bound to both transfers, so that the system guarantees the data is copied to *dest before* any fibers begin execution as a direct or indirect result of the sync signal sent to *slot*.

4.2 EARTH Architecture Model

The previous sections presented an abstract threading model and a set of operations performing the necessary functions of this model. This section considers the structure and organization of a real machine that can support this model. Since our goal is to support a range of machines containing varying amounts of off-the-shelf hardware, we can only present an *abstract* model of an architectural implementation of EARTH. This section also clarifies terms which were intentionally left vague in the previous section to avoid unwarranted or overly-restrictive assumptions about the hardware architecture.

According to the model, an EARTH computer consists of one or more EARTH *nodes* connected by a network. Each node has the following five essential components:

1. An *Execution Unit* (EU) for executing active fibers;

2. A *Synchronization Unit* (SU) for scheduling and synchronizing fibers and procedures, and handling remote accesses;

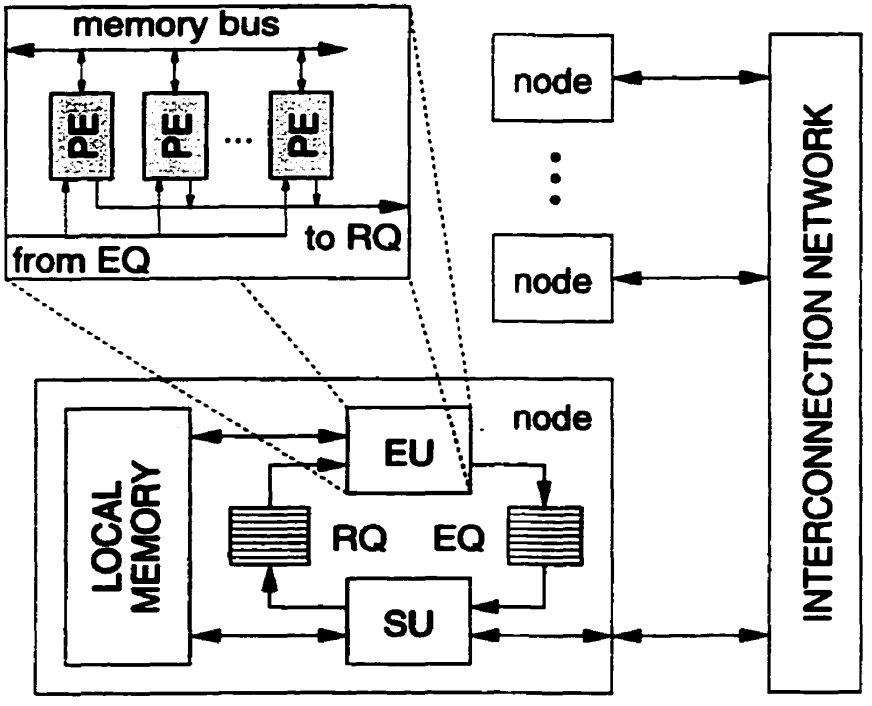


Figure 4.11: EARTH Architecture

3. Two queues, the *Ready Queue* (RQ) and *Event Queue* (EQ), through which the EU and SU communicate;

4. Local memory, shared by the EU and SU;

5. An interface to the interconnection network.

This division allows the implementation of multithreading architectures with off-the-shelf microprocessors mass-produced for uniprocessor workstations [61], one of our primary goals. An EARTH computer is shown in Figure 4.11.

4.2.1 Execution Unit

The EU executes individual fibers. As stated in Sections 4.1.1.1 and 4.1.1.6, the instructions within fibers are ordered according to sequential semantics. Modern uniprocessors are very efficient at sequential execution. Mechanisms such as by-passing or forwarding [71, 124] reduce hazards due to sequential dependences, while

small-scale instruction-level parallelism can be achieved with register renaming, out-of-order execution, register scoreboarding, and branch prediction [105]. Processors using these techniques would make good execution units for EARTH machines.

Most of the instructions executed in the EU are the normal arithmetic, load/store, and branch instructions that come with stock processors. To ensure flexibility, the EARTH model does not specify a particular instruction set. Instead, ordinary arithmetic and memory operations use whatever instructions are native to the EU processors. The context of a fiber is whatever registers are provided by the hardware. The EARTH operations specified by the PXM and EVM are for synchronization and communication. These are unlikely to exist in any form in ordinary processors, but these can be mapped to native EU instructions according to the needs of the specific architecture. For instance, on a machine with ASIC SU chips, the EU EARTH instructions would most likely be converted to accesses to memory-mapped addresses, which would be recognized and intercepted by the SU hardware.

Many details of the EU's behavior have been left unspecified in the EARTH PXM in order to leave considerable latitude in the design of the EU. Nothing in the preceding discussions of the EU should be taken to mean the EU is a single uniprocessor or superscalar executing one fiber at a time, even though this *would* be a correct implementation. In fact, the EU in the model can have processing resources for executing one *or more* fibers simultaneously. This is shown in Figure 4.11 as a set of parallel *Processing Elements* (PEs). These elements have the following properties:

1. Each PE runs a separate fiber independent of the other PEs.
2. Each PE has a *logically* private context for that fiber.
3. PEs share access to the memory system.
4. All PEs share access to the same Event Queue and Ready Queue, though each PE accesses the queues asynchronous with the other PEs.

Consequently, programmers should avoid unwarranted assumptions about concurrent fiber execution.

The drawing of the EU in Figure 4.11 is only an abstraction. Any structure

may be used so long as it can adhere to this abstraction. For instance, the Execution "Unit" could be a *set* of conventional processors, each executing one fiber at a time (EARTH has been implemented on a system consisting of clusters of Sparc processors, as mentioned in Chapter 7). Another possible choice for an EU would be a processor which holds multiple register contexts simultaneously and interleaves between them [3, 4, 103]. Even though such a processor would seem to eliminate the need for a non-preemptiveness requirement, such processors still have only a limited number of register contexts, and can't afford to have too many of them inactive while waiting for hazards to be resolved. The EARTH model would prevent such a situation from happening while taking advantage of an interleaving processor's ability to tolerate short-term latencies, such as floating-point operations. Finally, several processor organizations have been proposed in which multiple execution pipelines feed shared functional units. For instance, Simultaneous Multithreading [126] has demonstrated an ability to use multiple functional units efficiently by sending several threads of instructions to superscalar-like instruction scheduling units feeding the functional units, provided there are enough independent threads to hide instruction latencies. A processor based on such principles would be an excellent platform for EARTH, because EARTH's PXM can provide all the threads needed while EARTH's fiber scheduling policies would reduce the need for these threads.

4.2.2 Synchronization Unit and Queues

Off-the-shelf processors are suitable for dynamic scheduling of instructions within a single fiber. But they are not good for general-purpose multiprocessing, since they do not adequately address the performance issues discussed in Section 1.2: latency, bandwidth and synchronization. EARTH's solution is to separate the tasks not supported well by existing processors into a separate unit, called the Synchronization Unit, leaving the EU free to perform the tasks it does best. This has its roots in the Argument-Fetch Dataflow Architecture [25], discussed in Section 3.1.2).

The EU and SU messages to each other are buffered using queues called the *Ready Queue* (RQ) and *Event Queue* (EQ). These may be implemented using off-the-shelf devices such as FIFO chips, incorporated into a custom SU, or implemented in software. Various implementations are discussed in later chapters.

The EU executes fibers as if they were normal sequential code. The SU does not

execute any user code, so all the EARTH operators in Section 4.1.3 are executed within active fibers running on the EU. When the EU encounters one of these operators, it forwards the operator to the SU, which actually handles the request. The EU writes the request to the EQ, where it stays until being read by the SU.

The SU processes each event it reads from the EQ. Some events will lead to fibers being enabled. Enabled fibers (each fiber is a pair (FID, IP)) are placed in the RQ. When a PE in the EU terminates a fiber, it reads the next entry from the RQ and begins executing that fiber. If the RQ is empty, then the PE becomes idle and must wait until the SU writes a fiber into the RQ. If there is enough parallelism in the program, there will usually be at least one fiber in the RQ, and the EU will never be idle.

4.2.3 Node Memory

In the EARTH model, each node has memory which is accessible by the SU and all Processing Elements within the EU. Whether this memory is physically and logically local to the node or shared with other nodes depends on the implementation (memory issues are discussed in Section 4.1.2). There may also be memory which is private to the SU, but some of its functions (such as data transfers) require that it use the memory which is common with the EU.

Memory in most small-scale multiprocessors (as might be used for a *single* EARTH node) is sequentially consistent, because it is easy to support in a single-bus system with snooping caches.[7] However, an EARTH node can get by with a weaker consistency model, so long as it runs correct code which conforms to the EARTH thread model, avoiding the race conditions and other operation errors mentioned in Section 4.1.3. The use of synchronizations to order fibers according to data and control dependences ensure that fibers can run *only* when explicit sync signals tell the SU that these dependences have been satisfied. Therefore, it is only necessary that data accesses be ordered relative to these synchronization operations (as described in Section 4.1.3.3), which makes the minimal consistency requirements more like weak ordering.

[7]This refers only to consistency within a single node.

4.2.4 Network Interface

On a multinode machine, the final component of an EARTH node is the interface to the network connecting the node to other nodes. To maintain the portability of EARTH applications across machines with different network interfaces, the EARTH model does not provide for direct access to the network by the application code (i.e., the active fibers). As shown in Figure 4.11, all network traffic goes through the SU.

All the synchronizing operators in Section 4.1.3.3, as well as the INVOKE operator of Section 4.1.3.1, can specify remote destinations. Some operations may involve both local and remote references, but such operations will have multiple phases, and the message in each phase will have a specific destination. For instance, a SYNC_WITH_FETCH operation used for a split-phase transaction will usually specify a remote source and a local destination. A message must first go to the remote node where the source is located; this node will send back a message containing the data.

As with other components of the EARTH architecture, the behavior of the network is minimally specified to broaden design choices. However, several generalizations about the network can be made. First, to allow nodes to operate asynchronously, there should be no synchronization required between sender and receiver. For instance, a sender should be able to send a message to another node even if the other node is busy and not explicitly waiting for a receive. This implies a reasonable amount of buffering within the network and the network interface, and some kind of automatic flow control. Second, the implementation (either in the SU or in the network itself) should guarantee reliable end-to-end transmission. Third, to improve performance, some networks may have multiple paths between a given pair of nodes or message priorities allowing some messages to overtake others. Applications programs should *not* assume that messages are received in the same order as sent, and should strictly rely on the EARTH fiber synchronization operations to guarantee correct program behavior.

4.2.5 Functions of the Synchronization Unit

The SU's primary task is to support the EARTH threading model by implementing the operations listed in Section 4.1.3. To implement these operations, the SU must perform the following functions:

1. EU and Network Interfacing,
2. Event Decoding,
3. Sync Slot Management,
4. Data Transfers,
5. Fiber Scheduling,
6. Procedure Invocation and Load Balancing.

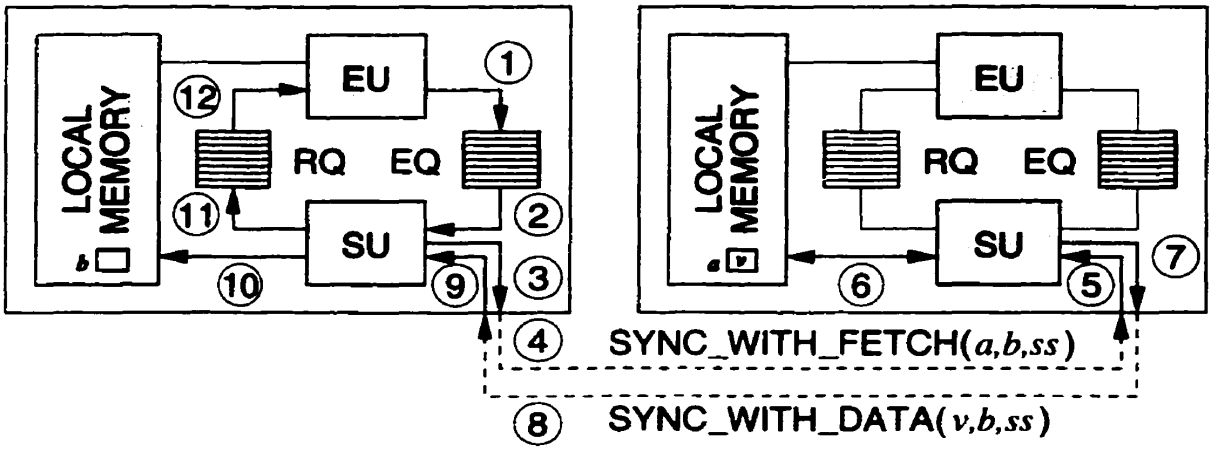
This section summarizes each of these functions.

EU and Network Interfacing The SU's interfaces with the EU are the Event and Ready Queues. The SU reads the Event Queue regularly and dequeue events from this queue as quickly as possible, in order to prevent the EQ from becoming full, which might cause the EU to stall (either by blocking the write or causing an exception in the EU to be generated, depending on implementation). On the other side, the SU tries to keep at least one enabled fiber in the Ready Queue at all times so that the EU never goes idle.

The SU also needs to maintain contact with the network, since all network traffic must go through the SU. The SU monitors the network interface regularly and reads any incoming messages promptly, and sends outgoing messages to remote nodes whenever the network is available. The interface does whatever protocol conversions are required by the network.

Event Decoding All events (operators) received from either the EQ or the network are decoded to determine what operation is needed. The SU needs to know if the destination is on its node or a remote node. (For split-phase transactions, the SU determines the specific destination of the first uncompleted phase.) For operations with remote destinations, messages are sent to the network interface with the destination node indicated. Local operations are queued for local processing.

Sync Slot Management To support fiber synchronization, the SU manages sync slots, being responsible for their allocation, initialization, and maintenance. The SU decrements slots in response to synchronizing operations, and decides when fibers are enabled.



1. Local EU sends SYNC_WITH_FETCH(a, b, ss) to EQ
2. Local SU reads event from EQ
3. Local SU determines location of a, sends request there
4. Network transfers message
5. Remote SU reads message from network
6. Remote SU reads value v from reference a in its memory
7. Remote SU sends message to network
8. Network transfers message
9. Local SU reads message from network
10. Local SU writes value v to reference b in its memory
11. Local SU decrements sync count of ss, places fiber in RQ
12. Local EU reads fiber from RQ

Figure 4.12: Steps in Split-Phase Transaction

Data Transfers The data transfer operations (SYNC_WITH_DATA and SYNC_WITH_FETCH) require reading and writing local memory, and transferring scalar data and blocks of data over the network. The SU must ensure that each transfer is properly bound with its synchronization signal, so that the ordering constraints in Section 4.1.3.3 are obeyed.

One benefit of the division of labor between the EU and SU is that split-phase transactions (remote fetches) do not require the intervention of the EU on the remote node. When an SU receives a SYNC_WITH_FETCH request from its local EU, the SU sends a message to the remote node. The remote SU decodes the message, fetches the requested memory contents, and sends the data back to the originating SU. This is illustrated in Figure 4.12.

Fiber Scheduling The Ready Queue pictured in Figure 4.11 suggests a simple FIFO between EU and SU for enabled fibers. However, the only restrictions imposed by the EARTH model on fiber scheduling are the synchronization rules implemented by the sync slots. A fiber can begin execution any time after the sync count in its corresponding sync slot reaches 0. While the simplest scheduling policy is a FIFO scheduling in which newly-enabled fibers are placed at the end of the RQ and the EU always read from the beginning, more elaborate policies are possible in implementations that allow efficient random access to this queue [61]. For instance, fibers can be prioritized to favor fibers known to be on critical paths, or fibers can be scheduled in LIFO order to benefit from register locality.

The need for register spills and reloads can be reduced by taking advantage of temporal locality in the scheduling of fibers. This means that the SU can favor (prioritize) fibers in the RQ which are immediately related to the currently executing fiber. (In the Threaded Abstract Machine [21], this would refer to fibers in a *quantum.*) For instance, if the SU determines that a fiber just enabled uses the same frame as a currently-executing fiber, and the registers corresponding to these frame values are the same, it can put the new fiber at the head of the RQ and advance the IP past the (now-redundant) register-loading section of the fiber. If the processor has multiple register sets (as in some current off-the-shelf processors), a *Register Use Cache* (RU-Cache) can keep track of which register set is assigned to which procedure instance. For example, a register file with n register sets will have an RU-Cache of n entries. The RU-Cache would be visible to the SU so that when a new fiber is enabled, the RU-Cache would be associatively searched for a frame identifier which matches the FID of the new fiber. A match in the cache would indicate that that thread should be prioritized. Moreover, the register set to which it is assigned should be included in the RQ entry so that when that fiber identifier is read by the EU, the proper register set would be used.[8]

Procedure Invocation and Load Balancing The final task of the SU is procedure invocation. Upon receiving an INVOKE request, the SU allocates a frame from the heap, initializes the frame with the procedure's input parameters, and enables the procedure's initial fiber, taking care to obey the ordering constraints given in

[8] This is a simple form of the *registering process* in the register-cache of the Super-Actor Machine [62, 57].

Section 4.1.3.1.

That section only stated that procedures are invoked, but did not talk about *where* they are invoked, leaving the details to the specification of the EVM. Since the initial procedure begins on only one node, there must be some way to invoke procedures on other nodes. Two straightforward choices are to let the programmer choose the node or let the EARTH's runtime system or hardware decide. The first choice is highly effective for applications with predictable load distributions, for the programmer has explicit control over where each procedure runs.

However, for many applications, especially irregular ones, the programmer may not have a good understanding of how to distribute data and the workload of the computations, or explicit programmer decision may simply be inconvenient. (An example would be the Fibonacci example used throughout this chapter.) For these cases, it is best to let the EARTH system decide where to invoke the procedure. The system uses various heuristics based on the current state of the nodes to choose what it thinks is the best node to run the procedure.

4.3 Extensions to EARTH

EARTH was originally designed to be a programming model that could be easily implemented on parallel machines based on off-the-shelf processors. For this reason, the operation set was kept to a minimum to reduce implementation complexity, while including enough operators to support efficient parallel processing. While the results of the next four chapters show that we have largely achieved our goal, there is room for improvement. Our original position papers on the design of EARTH [60, 61] suggested dividing EARTH into "levels" of complexity, in which higher levels support more features. The PXM presented so far in this chapter reflects, with minor changes, level 0 in the original concept. Certain features were seen as having potential benefits, but were postponed pending construction of the base machine.

Experience with implementations of our base model has shown that while most parallel algorithms can be represented in EARTH, some operations which are common in parallel programs require somewhat complex or convoluted programming in the EARTH PXM. Some of these were anticipated in our original paper, while others were discovered while programmers were attempting to port specific programs

to EARTH. The following extensions to the EARTH PXM involve no fundamental changes to EARTH and could be integrated into the current model.

4.3.1 Mutual Exclusion and Parallel Reduction

The deterministic nature of EARTH execution makes reduction operations (e.g., adding a set of numbers or finding the max of the set) difficult. Consider the problem of finding the maximum of n floats, each of which is generated by a separate instance of procedure P_1, and using the maximum in a procedure P_2. There are several ways to do it:

1. Allocate an array of n floats in P_2's frame, and pass to each instance of P_1 a reference to a different element of that array. Create a sync slot in P_2 with a count of n. Each P_1 sends its value to its element and syncs the slot in P_2. Then P_2 finds the max in a sequential loop. This is shown in Figure 4.13.

   This does no more work than any other method ($n-1$ comparisons are needed in any case), but this can be a bottleneck if getting the final max answer in P_2 is in a critical path (as in the tomcatv benchmark, used in later chapters, in which the reduction in P_2 is part of a barrier synchronization). It can also be problematic if n is unknown or unbounded at compile time.

2. Create a binary compare tree with $n/2$ leaves (a fragment is shown in Figure 4.14). If some of the instances of P_1 send values before others, then there will be some work to do in the tree while other work is still outstanding (assuming there are spare processing resources). This will shorten the amount of time needed by the sequential part which occurs when the last (N^{th}) instance of P_1 returns a value. However, this is complex and requires each call of P_1 to point to one of the leaves of the binary tree. This implies transferring pointers to the instances of P_1 through P_2. The effort might be worthwhile if this reduction is performed repeatedly, but would be a large overhead for a single event. This technique also has difficulties if n changes from one reduction to the next.

3. Traditional methods for guarding critical sections of code (e.g., semaphores) can be employed in the application code. But these impose high overheads on fine-grain applications and are an extra burden to the programmer.

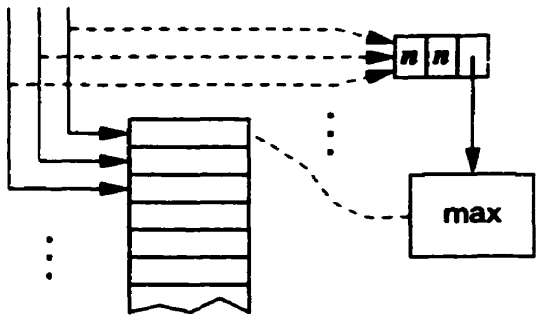


Figure 4.13: Reduction with Single Fiber

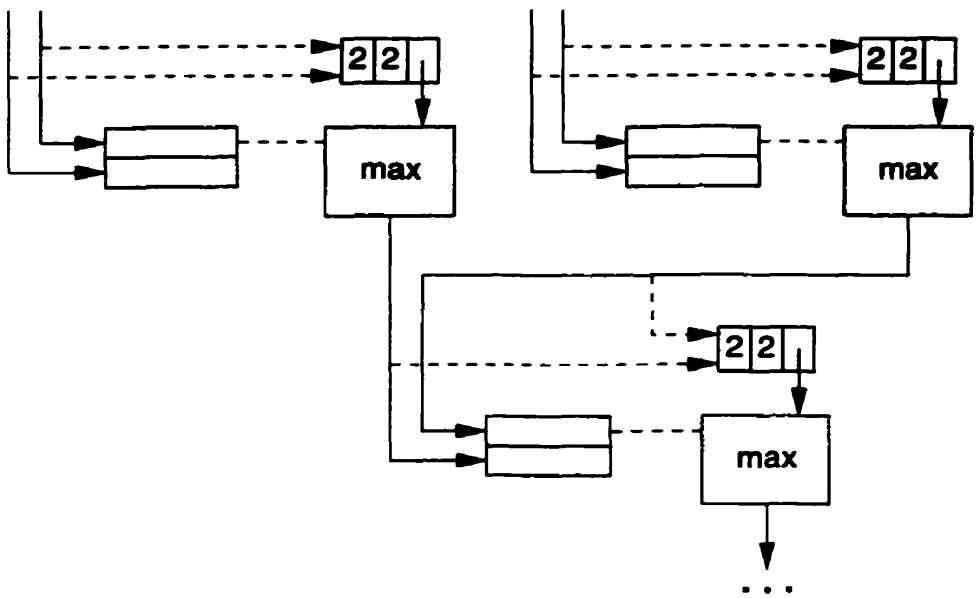


Figure 4.14: Reduction with Binary Tree

All three methods have been successfully used for benchmarks in EARTH. Some of the programming effort could be removed by providing special reduction library routines. But given the other disadvantages cited above, a more natural and efficient solution would look something like the abstract dataflow graph of Figure 4.15, in which all values are sent to a single fiber, which saves the internal state (the largest value found so far) until all values have been sent.[9]

[9]This non-deterministic merging solution would not work for numerical analysis problems in which the order of floating-point operations is important to the correctness of the algorithm. In such cases, one of the other solutions can be used.

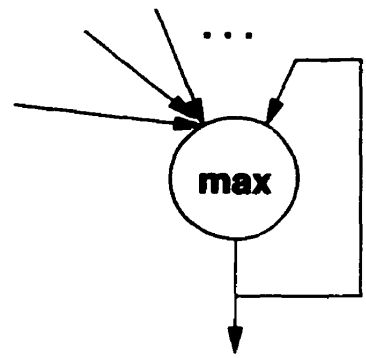


Figure 4.15: Reduction with Mutual Exclusion

The current PXM cannot support this type of graph, for two reasons:

1. If two values are sent to the same place at about the same time, one could overwrite the other before the latter has been used.

2. The general EARTH Architecture Model allows Execution Units to have multiple processing elements. Therefore, there is no way to guarantee atomic access to the critical resource (in this case, the partial maximum).

An enhanced EARTH PXM should include some support for mutual exclusion. Fortunately, this is possible with little change.

One of the higher "levels" in the first EARTH design paper [61] allows sync slots to be designated mutually exclusive. If the count in such a slot is decremented and it reaches 0, the fiber is fired as usual, *but the sync count is not reset.* This "locks" the fiber, which should have exclusive access to the critical resource (this is the programmer's responsibility). Any subsequent SYNC_WITH_DATA transfers involving the same sync slot are placed in a queue without updating the memory. When the fiber terminates, the slot's sync count is reset, "unlocking" the fiber. When this occurs, any pending SYNC_WITH_DATA transfers to the same slot may continue.

4.3.2 Support for Speculative Execution

Many non-numerical programs, and even some numerical programs, have the potential for big improvements in speed if some computation is performed *speculatively*,

i.e., before its need has been determined [119]. There are also many applications in which the program can search down many paths in the search space simultaneously, but a success in any subsearch satisfies the search conditions, i.e., an OR condition. The subsearches can often be done in parallel. The generation of parallel searches and speculative procedures and fibers is easy enough, but combining the results in a meaningful way is more complex, because nondeterminism is required. An efficient parallel search creates a race between the subsearches; the (temporally) first subpath to return a successful result is chosen as the solution to the next higher level of the search.

The previous extension for supporting mutual exclusion can be further modified to support this kind of merging. The mutex mechanism allows all incoming SYNC_WITH_DATA messages to be processed eventually. A modified mechanism would permit only one, or perhaps a predetermined number, of such signals to be accepted, causing all others to be discarded.

This would be far more efficient if there were a mechanism for killing computations whose results are no longer needed (e.g., parallel subsearches still in progress). A procedure that wished to kill some of the procedures that it invoked could send a signal to those procedure instances, which in turn would have to kill any processes they created, propagating down the call tree until all descendents are killed. Most of this could be done using the basic PXM, e.g., adding a special "kill" fiber to each threaded procedure. However, the more such features are built into the PXM, the better the hardware support for such operations can be made, leading to fewer redundant or unnecessary computations as superfluous fibers are killed faster.

4.3.3 Other Extensions

Section 4.2.5 suggested that the SU could make some fibers jump to the front of the Ready Queue in order to take advantage of register locality. This would be transparent to the programmer. However, priorities could also be made explicit to the programmer (or compiler) by introducing a more general priority mechanism, in which each thread (fiber or procedure instance) is given a priority level. Thus, one could identify critical paths in the code that need to be run quickly, and other parts that can be deferred until processor load is light. Such a priority scheme would also assist in the speculative execution raised in the previous section.

I-structures [11] have been proposed as a convenient way of decoupling producers and consumers, so that consumers could request data without knowing if the data has been produced yet, so long as the consumers know where the data will be when it finally *is* produced. The semantics of I-structures are similar to the EARTH SYNC_WITH_FETCH operator; the consumer requests data in a split-phase transaction. The key difference is that the SYNC_WITH_FETCH operator assumes the data is already present at the specified source location. An I-structure read operation, on the other hand, checks to see if the data has been produced, and does not return data from that location until it is known to be valid. Presense or absence of the data is determined by a full/empty bit attached to each I-structure location. This bit starts empty and is set when a producer writes data to that location. Multiple reads from the same location can be pending; when the location is finally written, the pending split-phase transactions are completed and the data is sent to each requester. A recent extension [14] allows these objects to be reused.

I-structures would simplify some programming, particularly in the beginnings of programs where parallel data and control structures are initialized. Several programs written for EARTH could benefit from I-structures. The semantics of I-structures are close enough to the EARTH SYNC_WITH_FETCH operator that the latter could be easily modified to support the former. The remote SU would simply have to hold onto the SYNC_WITH_FETCH until the data is written. (The PXM should require that the write be done with a SYNC_WITH_DATA operator rather than an ordinary store, so that the SU is alerted when the write occurs.) In fact, I-structures have already been implemented on EARTH in software [6], so it is not essential to add them to the PXM. However, including them in the PXM would mean that they would be supported in the EARTH hardware or system software, and this support would be more efficient than applications-level code.

Chapter 5

The EARTH Virtual Machine

The previous chapter presented the EARTH Program Execution Model (PXM). The PXM consists of a high-level view of the structure of an EARTH computer, an abstract model of how a program is divided into threads and how the threads are synchronized, and a set of eight fundamental operations which are essential to execute the threads correctly and efficiently. As stated in the chapter, much has been left unstated, ranging from specific details such as how the EARTH operations are encoded, to fundamental questions such as if and how memory is shared among nodes. The EARTH PXM is intended to be a general model for multithreaded execution which is applicable to as many platforms as possible.

An actual implementation of EARTH requires a more precise definition of the EARTH operations. This definition cannot be a complete Instruction Set Architecture (ISA), for that would bind the definition to a particular processor family. However, it should still fill the ISA's traditional role of specifying the interface between the programmer/compiler and the hardware. Our goal is to define a common set of operations in sufficient detail to permit both multiple hardware platforms and multiple high-level languages to be designed around this set.

A definition at this level is called an *EARTH Virtual Machine* (EVM). An EARTH Virtual Machine defines the following:

Memory model: How memory is distributed among the nodes, if and how it is shared, how memory locations are represented, what operations are available to the processors, and what form of memory consistency is supported (if any);

Thread model: How procedures and fibers (and their contexts) are represented, how they are invoked, and how sync slots are represented;

Data types: A set of data types used by the EARTH operators in this EVM;

EARTH instructions: A set of functions based on the EARTH operations outlined in Section 4.1.3, as well as restrictions on their use or constraints on their ordering. (Such constraints need to be specified so that code generators for EARTH can make optimizations, such as code reordering, without causing incorrect code behavior.)

Functions are chosen as the form for representing EARTH instructions because they are a suitable target for compilation. Furthermore, a function instantiation can easily be translated into a form suitable for whatever platform along the evolutionary path is desired. For an off-the-shelf multiprocessor without special SU hardware, a function call can be left as is (calling a library function) or replaced with inline code (see Section 7.1.3). At the other end of the path, where it is assumed the EU has builtin support for EARTH, the function call can be replaced with special opcodes.

This flexibility makes it possible, in principle, to compile code in two stages. First, a compiler converts a high-level language which supports EARTH (such as the Threaded-C language in the next chapter) to native processor code interspersed with the EARTH function calls. In the second pass, a post-processor, programmed for a specific implementation, converts the EARTH instructions to the native equivalents, whatever they may be. This allows use of existing compilers for off-the-shelf processors when they make up the EU. (Direct conversion from the high-level language would probably produce more efficient code, because function calls generally require fixed registers for parameter passing, which constrains the ability of the compiler's register allocator to optimize.)

This thesis proposes two virtual machines, both for distributed-memory computers. Both machines assume that memory is not shared, meaning that accesses to remote memory are not possible using ordinary load and store instructions. Since the EARTH model requires explicit communication of data and synchronization

between specified locations in different nodes, there must be some way to specify remote memory locations to EARTH operations.

The main difference between the two machines is the way in which memory locations are represented. The first uses "global addresses" to refer to memory locations. It is assumed that processors cannot use ordinary load and store operations to access memory on remote nodes, but that they can use global addresses, which are unique across the machine, to access remote locations *indirectly* using the provided set of EARTH instructions. The second virtual machine uses unique "frame identifiers" to refer to the contexts of threaded procedure instances, with frame offsets used to access individual memory locations within these contexts.

5.1 An EVM Based on Addresses

The simplest way to make it possible to reference memory locations globally is to extend the basic concept of a memory address to make it global. This is the basis of the first virtual machine, which is called EVM-A (an EVM based on *Addresses*). The next section presents an alternative machine based on frames.

5.1.1 The EVM-A Memory Model

The EVM-A assumes the platform is a nearly-homogeneous system in which all Execution Units are object-code compatible. The number of nodes is not determined at compile time, so the code should be designed to work with different numbers of nodes, but the number is fixed at the time a program begins execution. As previously stated, memory in this machine is not shared. Instead, each node in the EVM-A has its own memory and its own private address space. At least some part of the address space must be identical on all nodes. This part of the address space, called the *replicated address space*, is used for the following:

1. A copy of the executable code, consisting of all threaded procedures, sequential functions, constants, and static variables (not bound to a particular procedure) used by the program, is placed at the same address in the replicated space on each node. The threaded code is linked assuming that address as the base address. This ensures that any addresses appearing in the executable (branches, calls to sequential functions, or addresses of constant data structures) refer

to the identical object or code on every node. It also guarantees that each threaded procedure is associated with a single starting address which is identical on all nodes, which is important to the function invocation operations described later.

2. In some cases, objects are placed in the replicated space so that they may be found by the program in a fixed location independent of the node. For example, if all nodes in the machine share some read-only data which is generated at runtime (so that it can't be loaded with the program), this data can be copied to a predefined address in each node's replicated space. Subsequently, a thread running anywhere in the machine can access the same data by looking up the common address.

At least some part of the address space is "non-replicated." This is not to say that the address spaces *must* be distinct on different nodes; on most platforms, they are likely to be identical. It means that this memory is intended for non-replicated objects. Mainly, this storage is used for:

1. Frames holding the contexts of instances of threaded procedures;

2. Dynamically-allocated objects, i.e., a heap.

Figure 5.1 shows the memory for a four-node EARTH machine based on EVM-A. The code (e.g., the procedures `fib` and `func2`) is replicated on all nodes such that any given procedure or fiber has the same address everywhere (e.g., address 1e560 for the start of `func2`). Frames for instances of `fib` are shown in the non-replicated address spaces.

As pointed out in this chapter's introduction, it is essential for fibers to send data to remote locations, which means that addresses must be globally unique. For instance, if the instance of `fib(1)` on node 3 (Figure 5.1) sends its result to `fib(2)` on node 2, it must be able to specify that location in node 2's non-replicated address space. The solution is to create a special type of address, a *global address*, which uniquely identifies a particular local address on a specific node.

Global addresses cannot be used in load and store instructions, for memory is not shared. Even if the SU could be programmed to respond to such requests by asking a remote SU for memory contents, the EU would be stalled during this time, so it is more efficient to use the split-phase transactions introduced in Section 4.1.3.3.

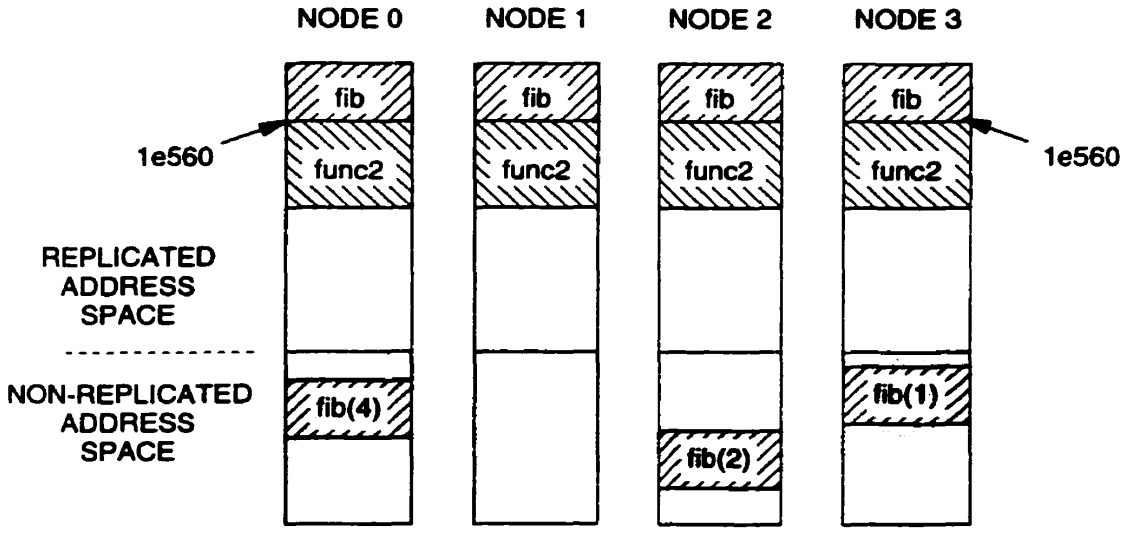


Figure 5.1: EVM-A Address Spaces

Instead, global addresses are passed to the SU using the EARTH operations. From any given global address, it must be possible to extract both the identity of the node and the local address on that node. The former is needed by the SU on the node that originates the EARTH operation, because it needs to know where to forward the request. The latter is needed on the node containing the specified memory location, since that node needs to access the local contents.

The non-replicated memory on a node contains dynamic state which is unique to that node. This mainly consists of heap and stack areas. For each Processing Element in the EU, there is a separate stack, which permits sequential function calls to be performed on that PE using conventional call instructions (see Section 4.1.1.3). The heap is used for allocating and deallocating both frames (contexts of procedure instances) and blocks of memory (e.g., dynamically-allocated arrays).

The latter can be created using conventional memory allocation techniques and shared among different procedures. Two procedures on the same node can share an object directly using loads and stores (with the object's local address). Procedures can access data structures on remote nodes, but only indirectly through global addresses, by using EARTH operations to read and write these structures.

5.1.2 The EVM-A Thread Model

The code for a threaded procedure is stored at the same address in each node (in the replicated space), so this address can serve as a globally-uniform reference to that procedure (a Procedure Pointer in Section 4.1.3). The same is true of individual fibers within a procedure. All information needed by the SU to invoke and initialize a procedure, including the address of the initial fiber, must be accessible given the address of the start of the procedure.

Within the code of a procedure, each fiber except the initial fiber is identified with a positive number. The initial fiber is not numbered because no EARTH instruction needs to identify this fiber; it is automatically enabled by the SU during procedure invocation. Fibers may have arbitrary numbers, but some implementations may produce more efficient code if numbers are consecutive.

Within a procedure, sync slots are numbered with consecutive non-negative integers starting from 0. Sync slots can only enable local fibers. To support synchronization with other procedures and other nodes, one must be able to reference a sync slot using a global address. From this address, the SU must be able to determine both the frame containing the sync slot and the number of that slot. These addresses may or may not point to the actual storage location of the slot, depending on the implementation, so user code should not attempt to access sync slot contents through these pointers, and should stick to the EARTH operations to manipulate slots.

When a PE in the EU takes a fiber from the Ready Queue, it needs to have access to that fiber's context, including its procedure frame. The RQ pairs the address of a fiber's code with the address of the frame associated with the instance of that fiber. The PE stores this frame address into a register and uses offsets from this register to access variables local to the frame, as in conventional programs. The stack pointer for the PE is also part of the context of a fiber.

5.1.3 Data Types of EVM-A

Section 4.1.3 defined several data types which were fundamental to the EARTH PXM. The following are the equivalent data types as represented in EVM-A. These are used within EARTH instructions, which are executed by fibers. The term "current fiber" refers to the fiber instance currently in the PE executing the EARTH

instruction, and the "current procedure" is the procedure instance corresponding to the current fiber. The current procedure's context is the "current frame."

fid — A frame identifier is a global address pointing to the base of a frame.

ip — An instruction pointer is a regular (local) address to the first instruction of the code for a fiber or procedure. Only a local address is needed because the code is stored in replicated memory, so an *ip* is identical on all nodes.

fibnum — A fiber number is a positive integer which must correspond to one of the fibers in the current procedure.

sptr — A slot pointer is a global address referring to a unique sync slot within one procedure instance. As previously mentioned, the code should not attempt to access a sync slot directly through its *sptr*.

ssnum — Certain EARTH instructions in EVM-A can refer to sync slots by number rather than by global address. In such cases, the sync slot referred to is always in the current procedure, and must be less than the number of sync slots allocated by that procedure.

nodeid — This is a non-negative integer large enough to contain the total number of nodes in the system. If used to specify a node's number, it ranges from 0 to $n-1$ (n is the number of nodes).

T — As in Section 4.1.3, the instructions defined below use T to refer to an arbitrary data type, except that EVM-A restricts T only to scalar objects, local pointers and global addresses (i.e., arrays are not allowed, although pointers to arrays are allowed). Movement of compound objects is handled in EVM-A using an explicit "block move" operator.

T* — For each object of type T, there is a local pointer to that object.

T*G For each object of type T, there is also a global address which refers to that object. The global qualifier is abbreviated "G" for compactness.

5.1.4 EARTH Instructions in EVM-A

One requirement of any EARTH Virtual Machine is that the EARTH instruction set contains at least the core EARTH operations of Section 4.1.3, implemented consistent with the memory model and data type set for this EVM. Refinements and extensions are permissible once the basic requirement is met.

EVM-A extends the basic operation set in several ways:

1. The basic operation set uses generic sync slot references and instruction pointers. EVM-A recognizes that many operations will involve data and sync slots local to the current frame. For instance, a SYNC_WITH_FETCH will usually fetch data to a local location and synchronize a local sync slot. Code can be optimized by avoiding the need to convert to and from global addresses, or by making the SU handle the conversions.

   Therefore, each of the synchronizing operators has two forms, one for local references, and one for remote references. In this context, "local" and "remote" describe locations relative to the current node. When a fiber refers to a location outside of its current frame, the programmer or compiler should conservatively assume that that location is on a remote node, particularly if the programmer or compiler makes aggressive use of parallelism and load balancing. While it is possible, in principle, to use local references to refer to objects in different frames that are on the same node, this should only be done if the programmer or compiler is 100% certain of the locality.

2. The basic synchronizing operators do all synchronization using sync slots. There are many cases where a fiber has only a single data or control dependence. While a sync slot with a reset count of 1 can be used, it is often more convenient for the fiber which satisfies the dependence to synchronize the dependent fiber directly rather than go through a sync slot. This is called *spawning* a fiber in EVM-A. A spawned fiber immediately goes into the Ready Queue of the node where that fiber's frame resides, while the spawning fiber continues execution.

NAME	ARGUMENTS	DESCRIPTION
sync	ssnum	Signal local sync slot
rsync	sptr	Signal remote sync slot
spawn	fibnum	Spawn local fiber
rspawn	fid, ip	Spawn remote fiber
data_sync	T, T*[G], ssnum	SYNC_WITH_DATA, local slot
data_rsync	T, T*[G], sptr	SYNC_WITH_DATA, remote slot
data_spawn	T, T*[G], fibnum	Atomic send/spawn local fiber
data_rspawn	T, T*[G], fid, ip	Atomic send/spawn remote fiber
get_sync	T*G, T*[G], ssnum	SYNC_WITH_FETCH, local slot
get_rsync	T*G, T*[G], sptr	SYNC_WITH_FETCH, remote slot
get_spawn	T*G, T*[G], thnum	Atomic fetch/spawn local fiber
get_rspawn	T*G, T*[G], fid, ip	Atomic fetch/spawn remote fiber
blkmov_sync	void*[G], void*[G], int, ssnum	Block move with local sync
blkmov_rsync	void*[G], void*[G], int, sptr	Block move with remote sync
blkmov_spawn	void*[G], void*[G], int, fibnum	Block move with local spawn
blkmov_rspawn	void*[G], void*[G], int, fid, ip	Block move with remote spawn

Table 5.1: EVM-A Synchronizing Instructions

3. EVM-A provides several functions that are used for converting between local and global addresses, which are not a part of the abstract model.

Table 5.1 lists the synchronizing operators, which perform the SYNC, SYNC_WITH_DATA and SYNC_WITH_FETCH operations. Each basic operator type has four variants, depending on whether the synchronization is local to the frame or remote, and whether a sync slot is used or the fiber is spawned directly. The local variants only require the number of the slot or thread. Remote synchronization requires a global address for the sync slot. Spawning a fiber outside the current frame requires a global address for the fiber's frame and the address of the fiber code.

The first three basic types correspond to the SYNC, SYNC_WITH_DATA, and SYNC_WITH_FETCH operations in the abstract model. The operations beginning with "data_" take a scalar value for their first argument; this scalar is copied to the location referenced by the second argument. Operators in the third group (beginning with "get_") take references to both the source (first argument) and the destination (second argument) of a copy. Some address arguments in the latter two groups may be local or global. These operators are therefore overloaded, and this is indicated by placing the optional "G" (abbreviation for GLOBAL) in square brackets. This

NAME	ARGUMENTS	DESCRIPTION
data_sync/spawn	T, T*, ...	Send data locally
data_rsync/rspawn	T, T*G, ...	Send data remotely
get_sync/spawn	T*G, T*G, ...	Fetch local copy of remote data
blkmov_sync/spawn	void*, void*, ...	Copy block locally
blkmov_sync/spawn	void*G, void*, ...	Fetch local copy of remote block
blkmov_rsync/rspawn	void*, void*G, ...	Send local block to remote dest.

Table 5.2: Common Data Transfer Instructions (EVM-A)

allows different types of transfers to be synchronized. For instance, a data_spawn can take a global address, meaning that the value is sent outside of the current frame but a local fiber is enabled. (The EVM-A implementation must guarantee, at a minimum, that the fiber does not enter the RQ until the remote location has been written.)

The last set of operators (beginning with "blkmov_") handle the special case of moving compound objects (arrays and records), and work with untyped addresses and a size count (number of bytes in the object). Since a block move's source argument is passed "by reference," if it copies a local object to another location, the copying will be concurrent with the continued execution of the fiber, despite the appearance of atomicity. If the fiber modifies the object before the transfer is complete, there will be a race condition between the transfer and the update, and the destination object could end up with incorrect data.

Although the EVM-A data transfer operations allow flexibility in choosing the location of the source, destination, and sync slot or fiber, we expect that most of the operations used will be initiated by the direct producers or consumers of data, and will have the following properties:

1. The sync slot or fiber specified is in the same frame as the destination of the transfer.

2. The source, or destination, or both, are in the current frame.

The operations with these properties are listed in Table 5.2.

Table 5.3 lists the remaining operations of EVM-A, corresponding to the remaining five operators listed in Section 4.1.3. Sync slot initialization only works on local

NAME	ARGUMENTS	DESCRIPTION
init_sync	ssnum, int, int, fibnum	Initialize local sync slot
incr_sync	ssnum, int	Increment local sync slot
incr_rsync	sptr, int	Increment remote sync slot
invoke	nodeid, ip, T, ...	Invoke procedure on specified node
token	ip, T, ...	Invoke procedure, node unspecified
end_fiber	—	Terminate current fiber
end_procedure	—	Terminate current procedure

Table 5.3: EVM-A Additional Instructions

NAME	ARGS	RET.	DESCRIPTION
node_id	—	nodeid	Local node number
num_nodes	—	nodeid	Total number of nodes
to_global	T*	T*G	Global address referring to local address
make_global	nodeid, T*	T*G	Global address on specified node
to_local	T*G	T*	Local address component
owner_of	T*G	nodeid	Node number component
slot_adr	ssnum	sptr	Global address for sync slot (id. by number)
frame_adr	—	fid	Frame identifier for current frame
ip_adr	fibnum	ip	Instruction pointer for fiber (id. by number)

Table 5.4: EVM-A Support Functions

slots. Both local and remote variants of the INCREMENT_SLOT operator are provided. Two types of function invocation are supported, one in which the program chooses the node where the function is invoked, and one in which the EARTH system makes the decision. (In the latter case, the procedure call becomes a "token" which enters a special queue; this is discussed in a later chapter.)

Finally, EVM-A provides several built-in functions and variables, listed in Table 5.4. The two predefined variables identify the local node number and the total node count (runtime constants, not compile-time constants). These are useful to programs which distribute procedures manually (using the invoke operation rather than the token operation). The next four convert between local and global addresses. The final three convert local slots, fibers and frame into global addresses which can be used by remote procedures to refer to these items.

5.2 An EVM Based on Frames

The current implementations of EARTH, presented in Chapters 7 and 8, are based on the EVM-A virtual machine. When EARTH was implemented on its first platform [59, 58], a variant of the EVM-A was chosen because this made code generation easier and more efficient. However, there are several shortcomings with the EVM-A:

1. Frequent conversions between local and global addresses are required. The distinction between local and global addresses is made necessary by machines in which the combined total of all the local non-replicated address spaces is larger than the entire virtual address space of one processor. For instance, a machine with 32-bit addresses, 256 megabytes per node, and 20 nodes would not be able to use a regular (32-bit) pointer for a global address; at least 33 bits would be required. 64-bit processors may solve this particular problem, provided enough bits are provided for virtual addresses, but such machines still require manipulating the virtual memory system (as described in Section 7.1.1) to avoid explicit conversions.

2. As with any pointer-based system, it is easy to write code that generates invalid addresses and then tries to dereference them. This can happen for conventional reasons, such as exceeding array bounds, but can also happen for EARTH-specific reasons such as attempting to synchronize a sync slot after its frame has been terminated and deallocated. Since global addresses have been separated from their relevant contexts (e.g., the frames to which they belong), it is difficult to perform runtime error detection such as frame bounds checking. (Although extensive runtime checking would slow down a purely software-based implementation of EARTH, such checking might be done in parallel in a hardware SU.)

3. If the assumption made in Section 5.1.4, that most data transfers synchronize sync slots in the same frame as the destination of the transfer, is true, then the arguments to such operations have considerable redundancy. Two global addresses (possibly 64 bits each) are used to specify a destination and a sync slot, yet these two addresses are usually quite close together. Removing some of this redundancy would free up register space in the EU, reduce the time to send messages to the SU, and reduce traffic on the network.

The early papers on EARTH [61] specified EARTH instructions that use frame/offset pairs, rather than global addresses, to refer to memory locations not in the current frame. This developed into a virtual machine independent of EVM-A, called EVM-F (an EVM based on *Frames*). Although there are no current implementations of EVM-F, it is being considered for future implementations of EARTH. It is currently an incomplete definition, but is presented here to show that the basic EARTH PXM can be supported by multiple virtual machines. Many features of EVM-F are identical to EVM-A, so only the differences are presented in this section.

5.2.1 The EVM-F Memory Model

The EVM-F relaxes the homogeneity requirements of EVM-A by allowing different processor architectures to reside in the same machine. This would allow an implementation of EARTH based on a cluster of diverse workstations. Furthermore, no part of the address space needs to be replicated on all nodes. The only requirement for compatibility is that all data representations (sizes, bit definitions, endianness, record layouts, etc.) must be the same on all nodes, to allow data structures to be transferred between nodes without costly conversion routines. (Data structures with pointers present a problem if the addresses on two nodes are different, unless the pointers are explicitly converted. But this problem also exists in EVM-A, unless the data structures are stored in the replicated address spaces. EVM-F provides an alternative to replicated address spaces, described below.)

Instead of using global addresses to refer to remote locations, the EVM-F uses a frame/offset pair. Each frame is assigned a unique identifier, which is used in EARTH instructions rather than global addresses. As with global addresses, it should be possible for the SU to extract from the frame identifier the node number and local address base of a frame. Only the SU where the frame resides needs to know its local address; the others only need to know the frame's node number. A memory location for data is referenced using a frame identifier and a byte offsets from the start of the area of the frame reserved for local variables. A sync slot is referenced using a frame identifier and the number of the local slot.

Thus, frames become somewhat analogous to segments in segmented virtual memory, except at a finer level of granularity. With 32-bit pointers, there can be up to 2^{32} globally-unique frames. If deallocated frame identifiers are eventually

recycled, then the addressing space on a 32-bit machine can be exhausted only if more than four billion procedure instances are active *simultaneously.*

As with segments, there is the possibility of checking for violations (exceeding frame bounds, accessing deallocated frames, etc.). With a hardware SU, these checks could be performed concurrently to avoid slowing the rest of the SU. In a pure software SU, these checks would probably be too expensive, but could be switched on for debugging.

In EVM-A, a portion of the address space is replicated on all nodes. In EVM-F, a portion of the *frame* space is replicated. Some frame identifiers refer to frames which are assumed to be on all nodes, though they may be stored locally in different places. As with EVM-A, these can be used for shared data, though, as with EVM-A, writing to one frame does not affect the corresponding frames on other nodes.

Some of these replicated frames have special functions:

1. At least one frame is reserved for static variables which are not bound to a particular procedure instance, i.e., their lifetime is the lifetime of the program.

2. Some replicated frames can be designated *broadcast frames*, meaning they are intended for storing identical copies of shared read-only data on all nodes. If the producer sends data to such a frame, the SU should send copies of the data to all nodes, taking advantage of any broadcast capabilities the network may have.

3. Finally, some frame identifiers are used as global labels for threaded procedures, as explained below.

In addition, frames can be used to identify heap-allocated data structures such as arrays, allowing these structures to be shared among procedures. (This does not include structures of fixed size declared as local variables within a procedure, which are part of the procedure's frame). Special memory-allocation routines are provided that produce frame identifiers rather than addresses.

5.2.2 The EVM-F Thread Model

The code for each procedure is stored on all nodes. The executable codes may vary from node to node if not all nodes are object-code compatible, though all copies are functionally equivalent. A frame identifier from the replicated frame space is used

to refer to this procedure on all nodes. Thus, this frame identifier corresponds to the PP (procedure pointer) in the abstract PXM (see Section 4.1.3). Note that this frame identifier refers to the procedure code rather than a particular instance of that procedure, and thus would be used for function invocation operations rather than synchronizing operations. Therefore, we call this special frame identifier a *procedure identifier.*

A fiber number (fibnum in EVM-A) identifies a particular fiber. A fibnum in conjunction with a procedure identifier specifies the code for one fiber. A fibnum in conjunction with a regular frame identifier specifies a fiber instance.

5.2.3 Data Types of EVM-F

Some of the special data types in EVM-F are changed to reflect the use of frames rather than global addresses. The frame identifier (*fid*) is now an abstract value not connected to any global address. The types *ip* and *sptr* are eliminated, their roles being filled by *fibnum* and *ssnum*, respectively, in conjunction with frame identifiers. Similarly, the GLOBAL qualifier is eliminated, since there are no longer any global addresses. Finally, the *offset* type represents the byte offset into the data area of a frame.

5.2.4 EARTH Instructions in EVM-F

The EVM-F synchronizing operations, listed in Table 5.5, are analogous to the operations in EVM-A. Global addresses are replaced with frame identifiers, offsets, and slot and fiber numbers. As with the EVM-A equivalents, some destinations can be either remote or local (relative to the current frame), so the fid is optional. Again, the operators listed in the table are overloaded and the optional arguments are enclosed in square brackets.

The use of frame identifiers allows more optional arguments to be omitted. Both the frame identifiers in data_rsync are marked as optional. At least one is required, since data_rsync synchronizes a remote slot, by definition. However, if this sync slot is in the same frame as the (remote) destination, then the second fid can be omitted, since it is identical to the first. If the destination is local, then the first fid can be omitted. Only if the destination is remote *and* in a different frame than the sync slot are two frame identifiers needed.

NAME	ARGUMENTS	DESCRIPTION
sync	ssnum	Signal local sync slot
rsync	fid, ssnum	Signal remote sync slot
spawn	fibnum	Spawn local fiber
rspawn	fid, fibnum	Spawn remote fiber
data_sync	T, [fid,] off., ssnum	SYNC_WITH_DATA, local slot
data_rsync	T, [fid,] off., [fid,] ssnum	SYNC_WITH_DATA, remote slot
data_spawn	T, [fid,] off., fibnum	Atomic send/spawn local fiber
data_rspawn	T, [fid,] off., [fid,] fibnum	Atomic send/spawn remote fiber
get_sync	fid, off., [fid,] off., ssnum	SYNC_WITH_FETCH, local slot
get_rsync	fid, off., [fid,] off., [fid,] ssnum	SYNC_WITH_FETCH, remote slot
get_spawn	fid, off., [fid,] off., thnum	Atomic fetch/spawn local fiber
get_rspawn	fid, off., [fid,] off., [fid,] fibnum	Atomic fetch/spawn remote fiber
blkmov_sync	[fid,] off., [fid,] off., int, ssnum	Block move with local sync
blkmov_rsync	[fid,] off., [fid,] off., int, [fid,] ssnum	Block move with remote sync
blkmov_spawn	[fid,] off., [fid,] off., int, fibnum	Block move with local spawn
blkmov_rspawn	[fid,] off., [fid,] off., int, [fid,] fibnum	Block move with remote spawn

Table 5.5: EVM-F Synchronizing Instructions

NAME	ARGUMENTS	DESCRIPTION
data_sync/spawn	T, off., ssnum/fibnum	Send data locally
data_rsync/rspawn	T, fid, off., ssnum/fibnum	Send data remotely
get_sync/spawn	fid, off., off., ssnum/fibnum	Fetch remote data
blkmov_sync/spawn	off., off., int, ssnum/fibnum	Copy block locally
blkmov_sync/spawn	fib, off., off., int, ssnum/fibnum	Fetch remote block
blkmov_rsync/rspawn	off., fid, off., int, ssnum/fibnum	Send local block remotely

Table 5.6: Common Data Transfer Instructions (EVM-F)

As with EVM-A, we expect that most operations will synchronize a slot or fiber within the same frame as the destination, and that either the source or destination will be in the current frame. Therefore, these common cases fall within the list of operations listed in Table 5.6.

The other operations in the EARTH instruction set are the same as those listed in Table 5.3, with the following exceptions:

1. The incr_rsync operator takes a frame identifier and ssnum instead of a sync slot pointer.

2. The invoke and token operators take a frame identifier (acting as a procedure identifier) rather than an instruction pointer.

3. The invoke operator returns a value directly (it is the only function to do so), which is the fid corresponding to the procedure instance.

The return of a frame identifier by **invoke** is a useful feature that can simplify procedure linkage. In a procedure such as **fib** (as presented in the previous chapter), procedures need to send data to the procedures that invoked them. This is simple to do because the invoking procedure can pass global addresses pointing to its variables and sync slots to the invoked procedure. However, once the callee has been invoked and started, it is more difficult to send data *from* caller *to* callee because the caller doesn't know where the callee is located. Some of the EARTH coding examples in the next chapter show how this is done. But if the invoke operation returns the frame identifier right away, the caller knows right away how to send data to the procedures it invokes.

There are several ways to implement this. What they have in common is that the SU can assign a frame identifier for a frame on a remote node before that frame has been allocated. One method is to allocate ranges of frame identifiers for specific node numbers on each SU. Another is to keep, in each SU, a small pool of frame identifiers pre-allocated for each remote node. In either case, the invoking procedure may try to transfer data to or from the frame whose fid it just received before the frame has actually been allocated on the remote node. The implementation must buffer these requests until the frame allocation is finished.

Chapter 6

The Threaded-C Language

The previous chapter defined two possible "virtual machines" implementing the EARTH Program Execution Model on distributed-memory machines. Each defines a set of primitive EARTH operations that are supported by the implementation. These operations serve as an interface between a high-level language and the machine code. The machine code (both ordinary instructions and the translations of EARTH operations) may vary widely, given our desire for portability along the evolutionary path, but compilers can assume these operations exist for all versions supporting a given EVM and target these operations. In principle, a compiler could include EARTH operations in its intermediate representation, which is used to separate the high-level language's structure from the code generation details.

Once a particular EVM has been defined, the next logical step is to develop a high-level language which supports the EVM, so that applications can be written for EARTH. The simplest approach is to present the EARTH constructs directly to the programmer. The programmer thus has control over the partitioning of programs into threads, the distribution of tasks among the processors, and so forth. It is straightforward to translate the program to the EVM layer by mapping the EARTH constructs to their EVM equivalents.

The EARTH group took this approach, developing a language called Threaded-C. This is ANSI-standard C with extensions corresponding to the EARTH operations and other constructs defined in EVM-A. A C compiler, written for the EU processor (and typically off-the-shelf), translates the regular C into native EU instructions, while provisions are made for converting the Threaded-C extensions. The Threaded-C language may be used as a user-level, explicitly-threaded programming language.

Threaded-C is also suitable as a compilation target for another high-level language. Part of the EARTH project team has been working on translating non-threaded languages, such as ordinary C or EARTH-C [52], a parallel language designed for EARTH but which hides threading details from the programmer, to Threaded-C. One important part of this work has been the development of an efficient algorithm to partition a procedure into fibers automatically [111].

Several dialects of Threaded-C have been written. The first version was written specifically for the MANNA multiprocessor [17]. This platform is very flexible and coding can be done at a very low level. Even the on-chip memory-management unit can be manipulated, something normally restricted to the OS kernel. Several features of "EARTH-MANNA Threaded-C" make it unsuitable for larger architectures such as high-end IBM SP-2 machines. In particular, that machine has the problem mentioned in Section 5.2; the aggregate address space is too large to fit in an ordinary address. This led to the introduction of explicit global addresses in both EVM-A and Threaded-C.

The remainder of this chapter is devoted to this version of Threaded-C. It begins with an overview of the essentials of Threaded-C, based on the description of EVM-A in Section 5.1. Since Threaded-C extends standard C with objects and operators corresponding to the elements of EVM-A, the presentation in Section 6.1 assumes familiarity with both C and EVM-A. Section 6.2 presents examples of Threaded-C programs, all of which are used in the experiments described in later chapters. A complete list of Threaded-C keywords, data types and operators is in Appendix B.

6.1 Overview of Threaded-C

Threaded-C constructs correspond very closely to objects and operations in EVM-A. Keywords, data types and operation names specific to Threaded-C are in uppercase to distinguish them from standard C. EARTH operations in Threaded-C are written as function calls, and their names and semantics are almost identical to the operations listed in Tables 5.1, 5.3 and 5.4.

6.1.1 Data Types and Qualifiers

While a rich set of data types are listed for EVM-A in Section 5.1.3, most of these conveniently map to standard C data types. Only the following types and qualifiers are added:

GLOBAL — The **GLOBAL** keyword is a type qualifier, much like the C **const** keyword. It indicates that a pointer should refer to a global address rather than a local address. For instance, the following lines declare **x** to be the global address of an int and **f** to be a function taking a global address (of a float) as its argument:

```
int *GLOBAL x
void f (int *GLOBAL)
```

SLOT — This is the data type of a synchronization slot.

SPTR — This is a shorthand for a global reference to a SLOT. The following two declarations are equivalent:

```
SPTR ss
SLOT * GLOBAL ss
```

THREADED — This is the data type of all threaded procedures.

A global address is associated with a specific node, so an EARTH operator can determine the node ID containing a specific address. One cannot dereference a global address directly. An operation such as ***x** is illegal (where **x** is the int pointer defined above). On the other hand, standard pointer arithmetic is allowed. For instance, (x+k) would be a *global address* pointing to the k^{th} element of an array of integers, assuming **x** is the address of the base. Pointer arithmetic always produces addresses on the same node; data structures cannot span multiple nodes unless components are explicitly linked with global addresses.

Since the internal form of a global address is implementation-dependent, there are special functions corresponding to the conversion functions in Table 5.4. The most common are TO_GLOBAL and TO_LOCAL, which convert between local and global addresses. Programmers should use these macros and not make assumptions

about the internal representations of these pointers.[1] Other functions are listed in Appendix B.4.

6.1.2 Structure of Threaded-C Code

Programs written in Threaded-C look very similar to ordinary C programs, with some small differences.

First, threaded procedures don't return data. The function name is preceded by the keyword THREADED, rather than a data type declarator. Threaded procedures currently cannot be declared static (this might change in the future). Also, the top-level function is called MAIN (upper case) rather than the standard lower-case "main." (This function only executes on one node; no other nodes run fibers until procedures are explicitly invoked on that node.)

Second, the body of a normal C function is defined as a *compound-statement* in the reference manual of Kernighan and Ritchie [69]. A threaded procedure has a more specific structure called a *threaded-procedure-body*. Using the same style as Kernighan and Ritchie, this is defined as follows:

threaded-procedure-body:

{ *declaration-list*$_{opt}$ *fiber-list threaded-procedure-terminator* }

fiber-list:

statement-list

statement-list fiber-delimiter fiber-list

fiber-delimiter:

END_THREAD ( ) ; **THREAD_***integer*:

threaded-procedure-terminator:

END_FUNCTION ( ) ;

RETURN ( ) ;

[1] If the global address space in a particular machine is small enough to fit in a regular pointer, then the EARTH implementation on that machine may use regular pointers for global addresses, in which case the GLOBAL keyword is ignored. However, programmers should always use the GLOBAL keyword to ensure portability.

The *declaration-list*, defined in Kernighan and Ritchie, is normally optional. But if sync slots are used in a threaded procedure body, then the *declaration-list* is required, and the first item declared *must* have the form

SLOT SYNC_SLOTS[*constant-expression*];

This declares the number of sync slots used in the procedure. This must be the first declaration in the procedure body.[2]

The procedure body consists of one or more fibers. The first is the initial fiber (see Section 4.1.1.3), which is automatically enabled when the procedure is invoked. All other fibers must begin with a label of the form **THREAD_***i*: (ending with a colon), where *i* is a positive integer.[3] Space between **THREAD_** and the fiber label *i* is not allowed.

Each fiber but the last ends with a call to the **END_THREAD()** function. If the procedure is to be invoked with either the INVOKE or TOKEN operators (the Threaded-C operators corresponding to the invoke and token operators described in Section 5.1.4), then the last fiber ends with a call to **END_FUNCTION()**. If the procedure is to be invoked with the CALL command (described in Section 6.1.3.2), the last fiber must end by calling **RETURN()**. In either case, the procedure is terminated when the last fiber finishes executing.

Thus, the body of a threaded procedure is partitioned into a set of fibers. Variables in the procedure's parameter list, and variables in the top-level *declaration-list*, are part of the context of the procedure's frame, and are visible to all fibers in the procedure. Variables declared within compound statements inside the procedure body obey normal C scoping rules.

In general, a compound statement (below the top level), such as a loop or conditional, should be contained completely within one fiber. However, it is possible for a fiber delimiter to be in the middle of a compound statement. This might simplify the code structure in some cases. For instance, a for loop in which each iteration

[2]This requirement, and some other seemingly arbitrary language requirements, were imposed to simplify the translators discussed in the next chapter. These are not inherently required by the needs of EARTH and will probably be relaxed in future versions of the language.

[3]The terms "fiber" and "threaded procedure" did not come into use until after Threaded-C was first defined. For this reason, the terms "thread" and "function" are used as keywords in this version of the language.

fetches remote data using a split-phase transaction could put the fiber boundary (corresponding to the split phase) inside the loop. However, like *gotos*, such constructs should be used sparingly, if at all. In the loop example, the processor would jump from one fiber to an earlier fiber when the end of the loop body is reached. This bypasses the normal fiber scheduling mechanism (taking fibers from the Ready Queue), and some implementations may not be able to handle a jump into the middle of a fiber. In such cases, it would be necessary for the compiler to split fibers at these locations.

6.1.3 EARTH Operators in Threaded-C

Most of the EVM-A operators in Tables 5.1, 5.3 and 5.4 have corresponding Threaded-C functions with the same names and argument types, except that the function names are in upper case. However, many of the operators in EVM-A are overloaded, which is not allowed in C. To maintain consistency with C semantics, the argument types of EARTH operations in Threaded-C are limited, and separate typed functions are provided in some cases. In particular,

1. In EVM-A, some operators, such as data_sync, allow an address argument to be local or global. In Threaded-C, all such addresses must be global.

2. The further overloading of operators in EVM-A, in which different data types are allowed (e.g., short or double), is not allowed in Threaded-C. Instead, different operators for each data type are provided, distinguished by extensions: _B for char (byte), _S for short, _L for long, _F for float, _D for double, and _G for global addresses.

Figure 6.1 illustrates the structure of Threaded-C code and the use of EARTH operators in a parallel version of "hello world." The MAIN function invokes, on each node, an instance of the `print_hello` procedure, which prints a message identifying the location (node number) of each instance. The following describes the actions at each statement:

Line 3: The `print_hello` procedure takes a reference to a sync slot as argument (`SPTR`).

```
#include    <stdio.h>

THREADED print_hello (SPTR done)                    /* line  3 */
{
    printf("Hello World from %d!\n", NODE_ID);      /* line  5 */
    RSYNC(done);                                    /* line  6 */
    END_FUNCTION();
}

THREADED MAIN (int argc, char** argv)
{
    SLOT     SYNC_SLOTS[1];                         /* line 12 */
    int      i;

    INIT_SYNC(0, NUM_NODES, NUM_NODES, 1);         /* line 15 */
    for (i = 0; i < NUM_NODES; ++i)
       INVOKE(i, print_hello, SLOT_ADR(0));        /* line 17 */
    END_THREAD();                  /* end of initial fiber */

THREAD_1:                                          /* line 20 */
    RETURN();
}
```

Figure 6.1: Parallel Hello World

Line 5: Sequential functions are called from threaded procedures in the same way they are called from other C functions. `NODE_ID` is a built-in constant identifying the node number.

Line 6: When `print_hello` finishes printing, it sends a sync signal to the sync slot. This is the "remote" form of synchronization, corresponding to the rsync operation in Table 5.1. Note that in this context, "remote" does *not* necessarily mean "on a different node," but merely "in a different frame" (though it also means the former in this example). An SPTR is needed to refer to a sync slot in a different frame, even if that procedure invocation happens to be running on the same processor.

Line 12: Only a single slot is needed, but SYNC_SLOTS is still declared an array.

Line 15: The first argument to INIT_SYNC indicates which slot is being initialized (slot 0). Since we expect a sync signal from each node, we set the initial sync count and reset count to NUM_NODES. The last argument identifies the fiber (THREAD_1) to execute when NUM_NODES syncs have been received.

Line 17: `SLOT_ADR(`i`)` is a built-in macro generating a pointer to sync slot i (type SPTR). This is what gets assigned to **done** in line 3. `SLOT_ADR` corresponds to the slot_adr function in Table 5.4.

Line 20: `THREAD_1` plays the role of a *barrier*. It won't run until all nodes have finished their printing.

If line 17 in Figure 6.1 is replaced with the line

```
TOKEN(print_hello, SLOT_ADR(0));
```

the same number of lines would be printed. However, the node numbers output would probably be different, because the system is making its own decisions about where to run each procedure. It is highly unlikely that each node would run exactly one procedure invocation. Indeed, many of the procedures would probably run on node 0, for these procedures are tiny, and some of them may finish before all the TOKEN commands have been executed. Furthermore, some load balancers (depending on implementation) weigh the cost of sending something to another node against the benefit of sending work to an unloaded node.

There is one restriction on EARTH operators, which is that they cannot be performed within normal sequential functions, with the exception of the POLL() command (described below). This is another artifact of the simplified compiler implementation, and may change in the future.

6.1.3.1 Polling

Implementations of EARTH on nodes without a second processor or hardware to support communication have to perform both the EU and SU functions in the same processor. A problem arises when a node issues a request for data on a remote processor (get_sync or block move) and the remote processor is executing a fiber. For efficiency reasons, the incoming message should not necessarily interrupt the running fiber. Polling is much more efficient than interrupts, so the processor should finish the fiber, and *then* poll the network for incoming messages.

Therefore, on single-processor-node implementations without special hardware support, the EU will poll the network every time it switches to a new fiber. However, if the fiber is long, the processor which issued the request may have to wait a long time for the data it requested, which could cause it to stall if that data is on a

critical path. Some Threaded-C compilers may try to insert polls into the code that it generates if it determines that a fiber may be long. However, the compiler may not always guess correctly, and is likely to be conservative in order not to impede the normal running fibers.

Therefore, Threaded-C provides a POLL() operation, which takes no arguments. The POLL command has no effect on the semantics of the Threaded-C code. It is merely a hint to the compiler (a pragma) suggesting that that location would be a good place to poll the network. Some compilers may choose to make their own decisions about where to put polls. On machines which can automatically address incoming messages without EU intervention (e.g., the MANNA), the POLL command is simply ignored. But programmers may wish to insert them into long fibers for portability.

POLL commands can be used inside of both threaded procedures and sequential functions. They are currently the *only* Threaded-C command which can be used inside a sequential function.

6.1.3.2 The CALL Operation

The TOKEN and INVOKE commands are the normal operators used for invoking threaded procedures. Each gives the name of a threaded procedure and zero or more arguments to that procedure; in addition, the INVOKE command specifies a node number. With each of these, the calling fiber continues execution, and the initial fiber of the specified procedure begins execution at an indefinite time in the future.

Threaded-C also provides a special procedure invocation operator CALL. A CALL command takes the same arguments as a TOKEN command: the procedure name followed by the arguments to the procedure. Like a TOKEN command, a CALL command creates a frame for the indicated procedure, and begins running the initial fiber of that procedure. The differences between CALL and the other two invocation commands are:

1. The specified procedure always runs on the same node as the fiber that executed the CALL.

2. The fiber that executed the CALL is immediately suspended, and control passes to the start of the initial fiber in the called procedure, even if there are fibers waiting in the Ready Queue.

3. When the called procedure terminates, the calling fiber resumes execution at the statement after the CALL. Again, the RQ is bypassed.

Thus, the control flow of a CALL is similar to a standard sequential function call. The difference is that if the called procedure has multiple fibers, then these fibers obey the normal fiber scheduling rules. This means that if there are fibers outside of this procedure in the RQ, then these fibers may run before the called procedure terminates.

In general, the linkage involved in a CALL is different from that in an INVOKE or TOKEN. For instance, when the procedure terminates, the EU must remember to jump back to the statement after the CALL, rather than taking the next fiber off the RQ. Thus, the code may be different for a procedure invoked with CALL. For this reason, procedures which are invoked with CALL must end with RETURN() rather than END_FUNCTION() in order to tell the compiler which type of linkage to use.

6.1.4 Non-Automatic Variables

As previously mentioned, the executable code is replicated on all nodes, so that any procedure can run on any node. A function or threaded procedure has the same local address on all nodes, as described in Section 5.1.1. Automatic variables and parameters, on the other hand, reside in procedure frames, whose addresses are specific to the nodes where they were invoked.

For consistency with C semantics, non-automatic variables have scoping rules similar to procedure and function addresses. These variables include *global-scope* variables, which are variables declared not within any sequential functions or threaded procedures, and static variables declared within functions or procedures. Such variables are replicated over nodes without explicit coherence mechanisms. Therefore if we declare such a variable, the address of this variable will be the same on all nodes. However, each node will have its own independent copy of the variable. Such variables obey normal C scoping rules; static variables declared within a procedure are accessible only within that procedure, and variables not within any function or procedure are either accessible within the module (if static) or anywhere (if external).

Global-scope variables can be useful in some situations. For instance, suppose

a program generates many tokens, each of which may read data from a read-only array (for instance, a lookup table of pre-computed values). Each procedure could fetch individual data it needs from a common source using get_sync operations, or could load the entire table using a block move. However, the former might introduce too much latency, and the latter would be wastefully redundant if each node were to run many instances of the same procedure. Instead, it would be better to make *one* copy of the table on each node, and put each copy in a place where any procedure knows where to find it.

While global-scope variables can be useful, they, like global variables in general, should be "considered harmful" and used only when necessary.

6.2 Threaded-C Examples

This section presents several examples of applications written in Threaded-C. All but the last are used in the experimental studies in the next two chapters. In these cases, the sequential C code is shown for comparison. In most cases, the Threaded-C code borrows some of the functions used in the sequential version, either without modification or with only slight modification. This demonstrates that it is possible, in many cases, to port existing sequential C code to parallel Threaded-C code without having to rewrite everything from scratch.

6.2.1 Fibonacci

The first example computes Fibonacci numbers using the naïve recursive algorithm used throughout Chapter 4. The sequential code is listed in Figure 4.2. While this is not an efficient way to compute Fibonacci, it is a simple illustration of how divide-and-conquer can be implemented on EARTH.

The parallel implementation was discussed in Chapter 4. The linkage of one procedure instance with its children is illustrated in Figure 6.2. The diagram shows each procedure instance as a light box, with two darker boxes representing the two fibers. Box 0 in each procedure represents the initial fiber, which is not explicitly numbered; the other box is fiber 1. Each fiber (except initial fibers) shows the init count and sync count for the sync slot that controls that fiber. (It is assumed here that each fiber is controlled by a single sync slot. This is usually true in practice, but

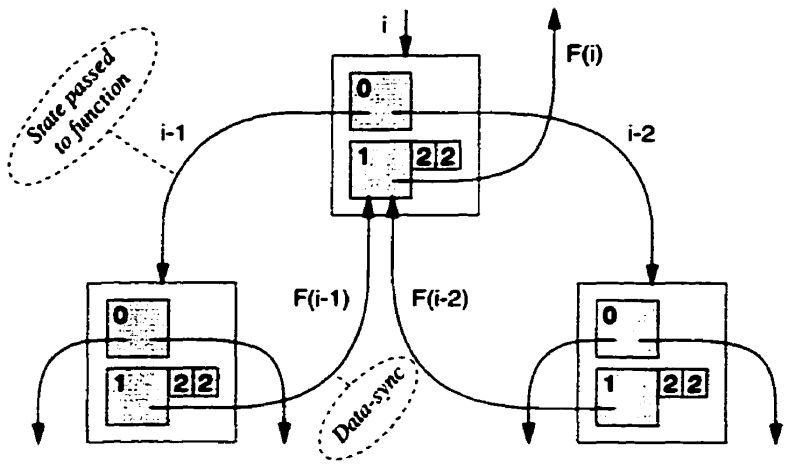


Figure 6.2: Fibonacci Code Structure

it *is* legal to have more than one sync slot control the same fiber, and Section 6.2.3 show an example of this.)

It is hard to determine *a priori* how to distribute the procedure instances evenly across the nodes, especially in this case where the call tree (see Figure 4.3) is not perfectly balanced. In fact, a perfect static distribution of the procedure instances in a parallel Fibonacci (such as in Figure 4.4) would require knowing ahead of time the number of children of each node. Since this information is related to the answer, knowing this at compile time would eliminate the need to run the program!

Therefore, we use tokens to take advantage of EARTH's load balancer. The code for the Fibonacci recursive implementation is shown in Figure 6.3. The resulting code looks similar to the pseudocode in Figure 4.7.

Since threaded procedures don't return values, the `fib` procedure must be called "by reference" in order to return its result to its caller. The variable `result` is a global reference for this purpose. In the recursive invocations, the macro TO_GLOBAL converts the local addresses &r1 and &r2 into global addresses so that the two instances of `fib` can send their results back properly even if they are not executed on the same node as their parent.

```
#include    <stdio.h>
#include    <stdlib.h>

THREADED fib (SPTR done, int n, int* GLOBAL result)
{
    SLOT    SYNC_SLOTS[1];
    int     r1, r2;                    /* Intermediate results */

    INIT_SYNC(0, 2, 2, 1);          /* 2 children => count of 2 */
    if (n < 2) {                /* This is a leaf - no recursion */
        r1 = 1; r2 = 0;                  /* Set up result of 1 */
        SPAWN(1);                   /* Will "return" value of 1 */
    } else {          /* Binary recursion: results go to r1, r2 */
        TOKEN(fib, SLOT_ADR(0), n-1, TO_GLOBAL(&r1));
        TOKEN(fib, SLOT_ADR(0), n-2, TO_GLOBAL(&r2));
    }
    END_THREAD();

THREAD_1:
    DATA_RSYNC_L(r1 + r2, result, done);
    END_FUNCTION();
}

THREADED MAIN(int argc, char** argv)
{
    SLOT    SYNC_SLOTS[1];
    int     N, res;

    N = atoi(argv[1]);              /* No checking for bad args */
    INIT_SYNC(0, 1, 1, 1);
    INVOKE(NODE_ID, fib, SLOT_ADR(0), N, TO_GLOBAL(&res));
    END_THREAD();

THREAD_1:
    printf("fib(%d) = %d\n", N, res);
    RETURN();
}
```

Figure 6.3: Threaded-C Code for Fibonacci

6.2.2 N-Queens

The N-Queens code is also a divide-and-conquer program, but, unlike Fibonacci, it is not contrived. The N-Queens code counts the number of ways in which n queens can be placed on an $n \times n$ chessboard so that no queen can attack any other queen (under normal chess rules). The code uses a recursive divide-and-conquer strategy.

A queen is considered to be in a valid position if it cannot attack any other queen in the chessboard. A partial solution is represented by a 1-dimensional array.

```
#include <stdlib.h>
#include <stdio.h>
#define MAX_QUEENS 24

int safe (int board[], int row, int col)
{       /* Return 1 if new queen can go in (row,col) on board */
  int rowchk, colchk;

  for (rowchk = 0; rowchk < row; rowchk++) {
    colchk = board[rowchk];
    if ((col == colchk) || (row - rowchk == col - colchk) ||
        (row - rowchk == colchk - col))          /* Conflict? */
      return 0;
  }
  return 1;                                    /* No conflicts */
}

int sequeens(int n, int row, int start_col, int board[])
{
  int col, num;

  if (row >= n) return 1;  /* Board full => this is 1 solution */
  for (col = start_col; col < n; col++) {
    if (safe(board, row, col)) {     /* Queen fits in this col */
      board[row] = col;
      num = sequeens(n, row+1, 0, board);
      return (num + sequeens(n, row, col+1, board));
    }
  }
  return 0;    /* Reached right end of board and no cols match */
}

main (int argc, char *argv[])
{
  int n, result, board[MAX_QUEENS];

  n = atoi(argv[1]);                    /* No check for bad args */
  printf("queens (%d) running on %d processors\n", n, 1);
  result = sequeens(n, 0, 0, board);
  printf("Number of solutions: %d\n", result);
}
```

Figure 6.4: Sequential C Code for N-Queens

Each element of this array corresponds to a row of the chess board and contains the column in which a queen is positioned in that row.

Sequential code is shown in Figure 6.4. As with the Fibonacci program, code to check for correct command-line arguments has been omitted for brevity. A search procedure is called with a row number **row**, a column number **start_col**, and a

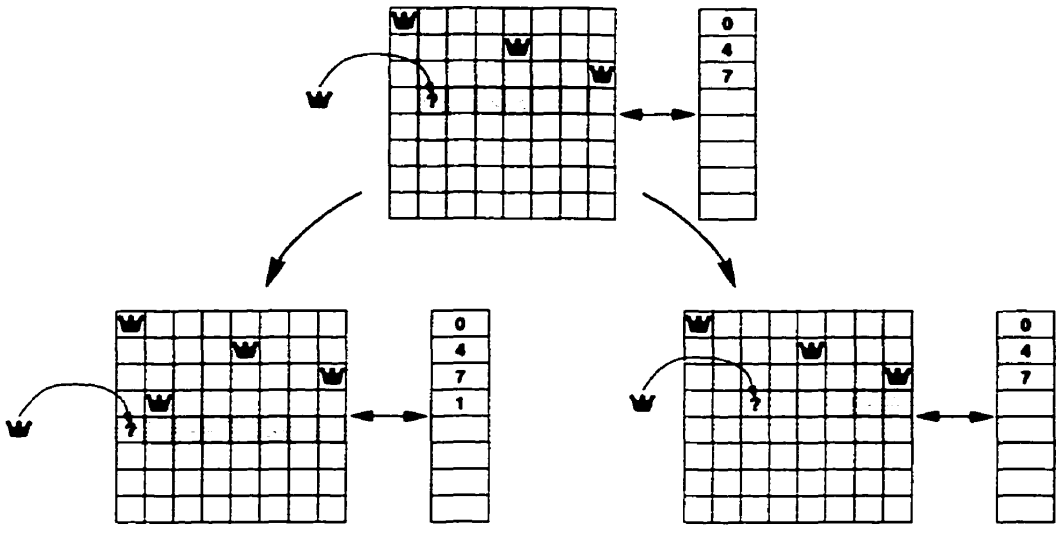


Figure 6.5: Recursion in N-Queens

partial solution array, in which rows 0 through `row`–1 have been filled with queens in valid positions. The procedure returns the number of complete solutions in which rows 0 through `row`–1 match the initial configuration, and in which the queen in row `row` is somewhere between columns `start_col` and $n-1$ (inclusive). It works by trying the queen in all positions from column `start_col` to the right side of the board. If it finds a valid position, it splits the search into two sub-searches, as shown in Figure 6.5:

- Add the new queen to the chessboard and start searching the next row.
- Keep trying positions to the right of the current position.

The top-level procedure calls the search procedure with an empty board.

The call structure is similar to our Fibonacci example, suggesting that a code structure similar to Figure 6.2 is appropriate. But there is a problem. The state of the computation is represented by a one-dimensional array. In a sequential implementation, this array can be continually updated in place, but parallel procedure instances would interfere with each other. Furthermore, procedure instances on different nodes can't share the same array, since EVM-A assumes a distributed memory machine. Instead, we need to replicate the state whenever we start a new procedure. However, since arrays can't be passed "by value" in C, the partial solution can't be directly passed to the next level in the parameter list.

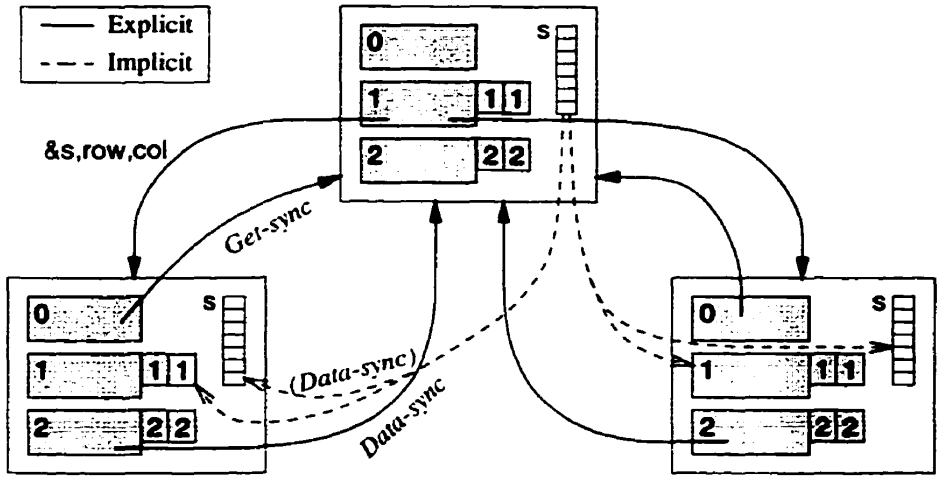


Figure 6.6: Data Transfers and Synchronization in the N-Queens Solution

The solution is for a procedure to fetch a copy of its parent's state. Since this may involve fetching from a different node, a split-phase transaction is used. We need to split the initial fiber in Figure 6.2, which performs the recursion, into two fibers: one to initiate the fetch and one to perform the recursion once the fetch has completed. The caller passes a global address which points to its copy of the partial solution. The callee then *fetches* the block with a block-move. Since the destination is local, this block move is similar to get_sync operators (which copy remote data to local memory in a split-phase transaction), except that more than one datum is moved. Since we are syncing a local sync slot (0), we use BLKMOV_SYNC, whose last argument specifies a local slot number. In our implementation, the initial fiber issues the block move request, and fiber THREAD_1 processes the data. This fiber then invokes the new search procedures (using TOKEN commands), and the results must synchronize THREAD_2. The structure is illustrated in Figure 6.6, where dashed lines indicate data transfers that are performed automatically by the EARTH runtime system (as a consequence of the EARTH operations used in the code).

Figure 6.7 lists the recursive search routine. The threaded code uses the sequential function **safe** (see Figure 6.4) to verify the validity of a partial solution. The MAIN routine, which makes the initial call to **queens**, is listed in Figure 6.8.

The recursive routine in Figure 6.7 fully traverses the call tree. The number of procedure instances grows exponentially as n increases. The number of instances can

```
THREADED queens(int *GLOBAL result,        /* Where to send result */
                SPTR done,                 /* Slot to sync when sent */
                int n,                              /* Size of board */
                int row,                /* Current row (fixed above) */
                int start_col,             /* Check from here to end */
                int *GLOBAL previous)       /* Caller's board state */
{
        SLOT    SYNC_SLOTS[2];
        int     col,                                /* Column index */
                own_board[MAX_QUEENS],      /* Local copy of board */
                sols_this_col,   /* # solutions with this position */
                sols_other_cols;       /* # solutions to right side */

  INIT_SYNC(0, 1, 1, 1);
  INIT_SYNC(1, 2, 2, 2);
  BLKMOV_SYNC(previous, TO_GLOBAL(own_board), row*sizeof(int), 0);
  sols_this_col = sols_other_cols = 0;      /* Preset sub-solutions */
  END_THREAD();

THREAD_1:                  /* Activated when block move is complete */
  for (col = start_col; col < n; col++) {    /* Here to right end */
    if (safe(own_board, row, col)) {
      own_board[row] = col;
      if (row+1 == n) {        /* Board full?  This is a solution! */
        sols_this_col = 1;                /* Set the partial result */
        SYNC(1);                 /* Sync directly instead of TOKEN */
      } else {                  /* How many solns with this partial? */
        TOKEN(queens, TO_GLOBAL(&sols_this_col), SLOT_ADR(1),
              n, row+1, 0, TO_GLOBAL(own_board));
      }
      if (col+1 == n) {         /* Last column?  No more solutions */
        SYNC(1);                 /* Sync directly instead of TOKEN */
      } else {                  /* How many solutions to the right? */
        TOKEN(queens, TO_GLOBAL(&sols_other_cols), SLOT_ADR(1),
              n, row, col+1, TO_GLOBAL(own_board));
      }
      break;
    }
  }
  if (col == n)          /* No safe position found?  Then return 0 */
    SPAWN(2);       /* Sub-solns already 0; start final thread now */
  END_THREAD();

THREAD_2:             /* Activated when both sub-solutions computed */
  DATA_RSYNC_L(sols_this_col + sols_other_cols, result, done);
  END_FUNCTION();
}
```

Figure 6.7: Threaded-C Code for N-Queens (Recursive Procedure)

```
THREADED MAIN (int argc, char *argv[])
{
  SLOT SYNC_SLOTS[1];
  int n, result, place;

  n = atoi(argv[1]);
  INIT_SYNC(0, 1, 1, 1);
  INVOKE(0, queens, TO_GLOBAL(&result), SLOT_ADR(0),
         n, 0, 0, TO_GLOBAL(&place));
  END_THREAD();

THREAD_1:
  printf("Number of solutions: %d\n", result);
  RETURN();
}
```

Figure 6.8: Threaded-C Code for N-Queens (MAIN Procedure)

```
if (safe(own_board, row, col)) {
  own_board[row] = col;
  if (row+1 == n) {       /* Board full? This is a solution! */
    sols_this_col = 1;              /* Set the partial result */
    SYNC(1);                /* Sync directly instead of TOKEN */
  } else if (row >= THRESHOLD) {
    sols_this_col = sequeens(n, row+1, 0, own_board);
    SYNC(1);
  } else {               /* How many solns with this partial? */
    TOKEN(queens, TO_GLOBAL(&sols_this_col), SLOT_ADR(1),
          n, row+1, 0, TO_GLOBAL(own_board));
  }
  if (col+1 == n) {        /* Last column? No more solutions */
    SYNC(1);                /* Sync directly instead of TOKEN */
  } else if (row >= THRESHOLD) {
    sols_other_cols = sequeens(n, row, col+1, own_board);
    SYNC(1);
  } else {                /* How many solutions to the right? */
    TOKEN(queens, TO_GLOBAL(&sols_other_cols), SLOT_ADR(1),
          n, row, col+1, TO_GLOBAL(own_board));
  }
  break;
}
```

Figure 6.9: Threaded-C Code for N-Queens (Throttled Version)

be reduced by "throttling" the growth of parallelism. If the body of the for loop in fiber THREAD_1 of Figure 6.7 is replaced by the code in Figure 6.9, then recursion can be halted after going down THRESHOLD levels, after which the sequential

```
#define real double

void MM(const real *A, const real *B, real *C, int size)
{
  int i, j, k;
  real *Cij, dAik;
  const real *Aik = A, *Bkj;

  for (i = 0; i < size; i++) {
    for (k = 0; k < size; k++) {
       Cij = &C[i*size];
       Bkj = &B[k*size];
       dAik = *Aik;
       for (j = 0; j < size; j++)
         *Cij++ += dAik * *Bkj++;
       Aik++;
    }
  }
}
```

Figure 6.10: Sequential C Code for Matrix Multiply (Block Multiply)

recursive routine in Figure 6.4 is called. Both the throttled and unthrottled versions of N-Queens are studied in the experimental chapters.

Other improvements are possible. For instance, if for a given partially-filled board, there are k positions in the next row in which a queen can be placed, then both the sequential and parallel programs can call (or invoke) all k sub-problems from within the same for loop, rather than forking as soon as one legal position is found. For the parallel code, this would require allocating enough space to hold k separate return values. Since k varies from one instance to the next, one would have to use the INCR_SYNC operator to change the sync count dynamically. Section 4.1.3.2 describes one correct way to do this.

6.2.3 Matrix Multiply

Dense matrix multiplication is a highly-regular application, in terms of both data movement and control structures. A simple sequential version is shown in Figures 6.10–6.12. This program uses *blocking* to improve the cache hit rate, by keeping the working data set small enough to fit in the L1 cache. Figure 6.10 is a function which multiplies two square matrices of a given dimension, and adds the resulting product to a third matrix. Figure 6.11 uses the first function to multiply two larger

```
#define real double

void multiply(const real *A, const real *B, real *C, int size, int block)
{
  int blocks = size/block;
  real *tmpA, *tmpB, *tmpC;
  int ii, jj, kk, i, j, k;

  tmpA = (real *) malloc(block * block * sizeof(real));
  tmpB = (real *) malloc(block * block * sizeof(real));
  tmpC = (real *) malloc(block * block * sizeof(real));
  for (i = 0; i < blocks; i++) {
    for (j = 0; j < blocks; j++) {
      for (ii = 0; ii < block; ii++)
        for (jj = 0; jj < block; jj++)
          tmpC[ii*block+jj] = C[(i*block+ii)*size+j*block+jj];
      for (k = 0; k < blocks; k++) {
        for (ii = 0; ii < block; ii++)
          for (kk = 0; kk < block; kk++)
            tmpA[ii*block+kk] = A[(i*block+ii)*size+k*block+kk];
        for (kk = 0; kk < block; kk++)
          for (jj = 0; jj < block; jj++)
            tmpB[kk*block+jj] = B[(k*block+kk)*size+j*block+jj];
        MM(tmpA, tmpB, tmpC, block);
      }
      for (ii=0; ii<block; ii++)
        for (jj = 0; jj < block; jj++)
          C[(i*block+ii)*size+j*block+jj] = tmpC[ii*block+jj];
    }
  }
  free(tmpA); free(tmpB); free(tmpC);
}
```

Figure 6.11: Sequential C Code for Matrix Multiply (Top-Level Multiply)

square matrices. The `multiply` function copies blocks from the large matrix to temporary blocks so that contiguous memory is used; otherwise, a large power of 2 for the problem size would lead to cache line conflicts. On the platforms tested, this blocked multiplier runs 20–100% faster than using `MM` to multiply the entire matrix in one call.

The main function in Figure 6.12 allocates and initializes the matrix, and computes a "magic number," or checksum, over the result. The matrix is initialized in place using a function (declared but not shown) which takes several seed values. The `magic_number` function (also not shown) computes and returns the checksum. Again, command-line arguments are not checked for correctness; also, the blocks

```
#include <stdlib.h>
#include <stdio.h>
#define real double

extern real *initmatrix(int, real, real, real);    /* Int is size */
extern double magic_number(const real *, int);

main (int argc, char *argv[])
{
  int size, block;
  real *A, *B, *C;
  double magic;

  sscanf(argv[1], "%d", &size);        /* Dimension of big matrix */
  sscanf(argv[2], "%d", &block);           /* Dimension of block */
  printf("Starting computation of %d x %d result\n", size, size);
  A = initmatrix(size, 1.0, -2.02, 3.0);
  B = initmatrix(size, -5.5, .82, 10000.0);
  C = (real *) calloc(size * size, sizeof(real));   /* Assume all */
  multiply(A, B, C, size, block);              /* zero bits = 0.0 */
  magic = magic_number(C, size);
  printf("Magic # is %f\n", magic);
}
```

Figure 6.12: Sequential C Code for Matrix Multiply (Main Routine)

returned by `malloc` are not checked for validity.

The parallel version of matrix multiply uses Cannon's algorithm [20]. In this algorithm, the A, B and C matrices are blocked and distributed among the nodes in a grid fashion. The blocks of the result matrix C remain in place, while the blocks for multiplicands A and B are shifted after each block multiplication. In order for the blocks of A and B to be in the right place at the right time, they must begin in a staggered state.

Figure 6.13 shows the initial state of the blocks if the main matrix dimension is 4 times the block dimension. In each phase of the computation, corresponding blocks of A and B align at each grid point to contribute a partial sum to the block C kept at that point. All 16 block multiplications can run concurrently, after which all blocks of A are passed to the left and all blocks of B are passed upward. After 4 such computation-communication stages, the distributed C blocks contain the product of A and B.

The Threaded-C implementation of Cannon's algorithm is listed in Figures 6.14–6.17. Figure 6.14 contains declarations of data types and external functions. The

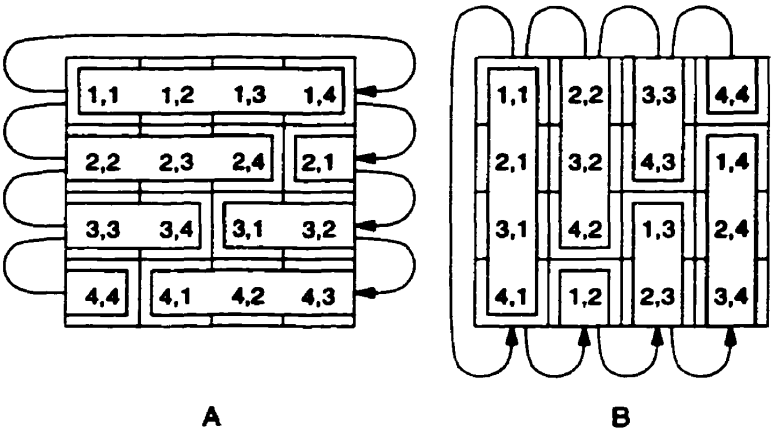


Figure 6.13: Block Rotation in Cannon's Algorithm

```
#include <stdlib.h>
#include <stdio.h>
#define real double

void MM(const real *, const real *, real *, int);
void initmatrix(int, real, real, real, int, int, real *);
double magic_number(const real *, int, int, int);

typedef struct ptr_and_sync {
  real *GLOBAL ptr;
  SPTR slot;
} ptr_sync;

#define SYNC_NOMEM      0     /* Malloc failed (out of memory?) */
#define SYNC_INIT       1     /* All procedures have been invoked */
#define SYNC_READY      2     /* Perform one communicate/compute stage */
#define SYNC_READY2     3
#define SYNC_MAGIC      4     /* Compute magic number */
```

Figure 6.14: Threaded-C Code for Matrix Multiply (Declarations)

block multiplier is identical to the sequential code in Figure 6.10. The matrix initialization function and magic number generator are modified somewhat to account for the fact that the matrices are distributed and aren't initialized in one place. Both now take additional arguments to indicate the position of the block within the larger matrix, because the initial values and checksum are dependent on this information. The initializer also now writes to a block already allocated, rather than allocating a block itself. The `ptr_sync` data type has fields pointing (via global addresses) to

a block and to a sync slot. These are combined into one struct because they are often sent bundled in a single block-move. Finally, the end of Figure 6.14 contains symbolic definitions for sync slots, used by both threaded procedures.

Each block of C is computed by one instance of procedure `block`. It is perhaps best to understand this kind of threaded program by first considering the steady state, which in this case is fiber THREAD_2 of `block` (see Figure 6.16). This fiber, which is executed k times per procedure instance (where k is the number of blocks per row or column of the whole matrix, e.g., 4 in Figure 6.13), does the following:

1. Send the block of A to the neighbor on its left.
2. Send the block of B to the neighbor above.
3. Multiply the blocks (the *same* blocks that were just sent) and add the product to the resident block of C.
4. Inform the neighbor to the right and the neighbor below that they are free to send new blocks.
5. Inform itself that it is free to start the next iteration.

This Threaded-C program achieves good performance (as will be shown in the next chapter) because the communication (steps 1–2) are initiated before the computation starts, allowing communication and computation to be overlapped. However, this kind of overlap can be dangerous if not implemented correctly, because one of the block-moves in step 1 or 2 can overwrite a block in one of its neighbors before the neighbor has finished using the previous data, which would lead to erroneous results. The following features of this Threaded-C code prevent this from happening:

1. When memory is allocated for A and B, each block is large enough to hold *two* blocks. The block-moves send blocks to alternating halves. An odd iteration of THREAD_2 uses the lower halves of the buffers holding A and B, while sending those same halves to the *upper* halves of the buffers in its neighbors.
2. Two sync slots (symbols SYNC_READY and SYNC_READY2) are used to control alternate executions of THREAD_2. This is essential because otherwise, sync signals corresponding to different iterations could arrive at the same sync slot at the same time, leading to premature enabling of a fiber before the

data is actually ready. Note that the sync slot semantics allow more than one sync slot to enable the same fiber.

3. The sync signals sent downward and rightward (step 4) aren't sent until the computation (step 3) has been completed.

4. Each procedure instance sends a sync to itself, ensuring that a fiber won't become enabled while the same fiber in the same procedure instance is currently running.

The remaining parts of Figures 6.15–6.17 are there to set up this steady state and to clean up when it is over. There are k^2 instances of `block`. The first instance, corresponding to the upper left block in the matrix, is invoked by MAIN, and each instance invokes the next instance. INVOKE is used to place the distribution under program control, and nodes are assigned cyclically, ensuring the most even distribution possible. Each instance of `block` invokes its neighbor to the right; the rightmost block in each row invokes the leftmost block in the row below.

For THREAD_2 to work, each procedure instance must have a global reference to:

1. Its left neighbor's buffer for A blocks;

2. Its upper neighbor's buffer for B blocks;

3. The sync slots of all four neighbors.

Linking a procedure to its left neighbor is easy in most cases, since it is *invoked* by that neighbor, who can send it the appropriate global addresses. A special provision must be made for the leftmost block in each row, since its "left" neighbor is at the right end of the row. Linking a procedure to its right neighbor's sync slots can be done by passing to that neighbor (in the argument list) a reference to a global address, which the invoked neighbor then fills with a reference to its own sync slots; this is done by fiber THREAD_1. Again, connecting a right-end block to its "right" neighbor at the other end is handled as a special case.

What is more difficult is connecting to neighbors above and below, since there may be many instances in between. A straightforward way to do this is for each instance to send references to its own B block and sync slots to a central table, allocated by **MAIN**. This occurs at the end of the initial fiber. When **MAIN** receives

a pair of references from each of the k^2 instances, it (in fiber THREAD_2) sends a sync signal to the rightmost block in each row, which starts a wave of synchronizations propagating to the left (THREAD_1 in `block`). Each instance of that fiber fetches the references from the appropriate entries in the central table, using a BLKMOV_SYNC and a GET_SYNC_G.

Once these last two fetches have completed, a procedure instance has all data it needs to begin the first block multiplication. From the preceding description of the steady state, one can see that each instance will receive 5 sync signals per iteration, which means that the counts for sync slots SYNC_READY and SYNC_READY2 should be 5. However, the *initial* count for the former should be 2 rather than 5. This is because for the first iteration, the buffers already have their data, and only the two fetches initiated by THREAD_1 need to be completed.

The remaining code mainly has to do with error detection and the computation of the magic number. Both use the original chain of communication from a procedure to its caller, which eventually propagates back to `MAIN`. This sequential chain is used, rather than a more advanced parallel communication, because the error handling (THREAD_4) is an exceptional occurrence, and the magic number computation (THREAD_3) is only for checking the correctness of the result. Both cause "clean" termination, i.e., by the time `MAIN` executes the RETURN command, all other procedures have been terminated.

In a real application using matrix multiply, it will probably be useful to multiply matrices many times in one run. Once the function linkage has been performed (THREAD_1 and the initial fiber of `block`), the procedure instances can retain their connections. The code can be easily modified so that completing the multiply leads to the distribution of new matrices rather than termination of the procedures.

This program also illustrates the use of conventional (`malloc`) function calls. Each procedure instance's buffers for A and B are allocated from the node's heap. They are made accessible to other procedure instances via EARTH operations by using TO_GLOBAL to convert the address returned by `malloc` into a global address.

Two more small points need to be made about this code. First, the use of the first argument to `block` to pass two separate values is merely an artifact of the MANNA implementation (see Section 7.1), which allows only ten integer arguments. Second, this program does not work if there is only one block in the matrix, but a slight modification to the code can accommodate this special case.

```
THREADED block (long                    block_and_size, /* Dimensions */
                int                     blockcnt,       /* Position */
                ptr_sync *GLOBAL        main_table,     /* Put B, slot here */
                SPTR                    slot_main,      /* Main slots */
                real *GLOBAL            A_left,         /* Send A here */
                SPTR                    slot_left,      /*  and sync here */
                SPTR *GLOBAL            slot_link,      /* Put own slot here */
                ptr_sync *GLOBAL        A_wrap,         /* Rightmost A here */
                SPTR                    slot_wrap,      /* Leftmost slot */
                double *GLOBAL          magic_left)     /* -> Left magic # */
{
  SLOT          SYNC_SLOTS[SYNC_MAGIC + 1];
  int           blocks, size, blocksize, row, col, count, odd,
                offset_new, offset_old, slot_num;
  real          *A, *B, *C;
  double        magic_num, local_magic;
  ptr_sync      A_ptr, B_ptr;
  SPTR          slot_right, slot_down;

  blocks = block_and_size & 0xffff; size = block_and_size >> 16;
  blocksize = size * size;
  A = (real *) malloc(blocksize * 2 * sizeof(real));
  B = (real *) malloc(blocksize * 2 * sizeof(real));
  C = (real *) calloc(blocksize, sizeof(real)); /* Init C to all 0s */
  if (A == NULL || B == NULL || C == NULL) {
    SPAWN(4);                                   /* Out of memory */
  } else {
    int nextblock = blockcnt + 1;
    row = blockcnt / blocks; col = blockcnt % blocks;
    INIT_SYNC(SYNC_NOMEM, 1, 1, 4); INIT_SYNC(SYNC_MAGIC, 1, 1, 3);
    if (col == 0) {
      INIT_SYNC(SYNC_INIT, 3, 1, 1);
      A_wrap = TO_GLOBAL(&A_ptr); slot_wrap = SLOT_ADR(0);
    } else {
      INIT_SYNC(SYNC_INIT, 1, 1, 1);
      A_ptr.ptr = A_left; A_ptr.slot = slot_left;
    }
    if (nextblock < blocks * blocks)
      INVOKE((NODE_ID + 1) % NUM_NODES, block, block_and_size, nextblock,
             main_table, slot_main, TO_GLOBAL(A), SLOT_ADR(0),
             TO_GLOBAL(&slot_right), A_wrap, slot_wrap, TO_GLOBAL(&magic_num));
    INIT_SYNC(SYNC_READY, 2, 5, 2); INIT_SYNC(SYNC_READY2, 5, 5, 2);
    B_ptr.ptr = TO_GLOBAL(B); B_ptr.slot = SLOT_ADR(0);
    initmatrix(size, 1.0, -2.02, 3.0, row*size, ((row+col)%blocks)*size, A);
    initmatrix(size, -5.5, .82, 10000.0, ((row+col)%blocks)*size, col*size, B);
    BLKMOV_RSYNC(TO_GLOBAL(&B_ptr), main_table + blockcnt,
                 sizeof(ptr_sync), slot_main + SYNC_INIT);
  } END_THREAD();               /* Initialize A and B in staggered form */
```

Figure 6.15: Threaded-C Code for Matrix Multiply (Block Procedure Initial Thread)

```
THREAD_1:                          /* Spawned after global table ready */
  if (col != 0)
    DATA_RSYNC_G(SLOT_ADR(0), slot_link, slot_left + SYNC_INIT);
  if (col == blocks - 1) {
    DATA_RSYNC_G(TO_GLOBAL(A), &A_wrap->ptr, slot_wrap + SYNC_INIT);
    DATA_RSYNC_G(SLOT_ADR(0), &A_wrap->slot, slot_wrap + SYNC_INIT);
    slot_right = slot_wrap;
    magic_num = 0.0;               /* Init needed if lower right corner */
  }
  BLKMOV_SYNC(main_table + (blockcnt - blocks + (row==0 ? blocks*blocks : 0)),
              TO_GLOBAL(&B_ptr), sizeof(ptr_sync), SYNC_READY);
  count = 0;
  GET_SYNC_G(&(main_table[row == (blocks-1) ? col : blockcnt+blocks].slot),
             TO_GLOBAL(&slot_down), SYNC_READY);
  END_THREAD();

THREAD_2:                              /* Multiply local blocks and add to C */
  odd = count & 1;
  if (odd) {
    offset_new = 0; offset_old = blocksize; slot_num = SYNC_READY;
  } else {
    offset_old = 0; offset_new = blocksize; slot_num = SYNC_READY2;
  }
  BLKMOV_RSYNC(TO_GLOBAL(A + offset_old), A_ptr.ptr + offset_new,
               blocksize * sizeof(real), A_ptr.slot + slot_num);
  BLKMOV_RSYNC(TO_GLOBAL(B + offset_old), B_ptr.ptr + offset_new,
               blocksize * sizeof(real), B_ptr.slot + slot_num);
  MM(A + offset_old, B + offset_old, C, size);
  if (++count == blocks)           /* Multiply is finished */
    RSYNC(slot_main + SYNC_READY);
  else
    SYNC(slot_num);          /* Enable next iteration of same location */
  RSYNC(slot_right + slot_num); /* Tell owners of next A & B it's OK to send */
  RSYNC(slot_down + slot_num);
  END_THREAD();

THREAD_3:                          /* Compute local part of magic # and pass */
  local_magic = magic_number(C, size, row * size, col * size);
  DATA_RSYNC_D(local_magic + magic_num, magic_left, slot_left + SYNC_MAGIC);
  SPAWN(5);
  END_THREAD();

THREAD_4:                          /* Spawned when someone's malloc fails */
  RSYNC(slot_left + SYNC_NOMEM);
  SPAWN(5);
  END_THREAD();
THREAD_5:
  END_FUNCTION();
}
```

Figure 6.16: Threaded-C Code for Matrix Multiply (Block Procedure Continued)

```
THREADED MAIN (int argc, char *argv[])
{
  SLOT          SYNC_SLOTS[SYNC_MAGIC + 1];
  int           size, blocksize, blocks_per_row, i, total_blocks;
  ptr_sync      *main_table;
  double        magic_num = 0.0;

  sscanf(argv[1], "%d", &size);
  sscanf(argv[2], "%d", &blocksize);
  blocks_per_row = size / blocksize;
  total_blocks = blocks_per_row * blocks_per_row;
  main_table = (ptr_sync *) malloc(total_blocks * sizeof(ptr_sync));
  INIT_SYNC(SYNC_NOMEM, 1, 1, 1);
  INIT_SYNC(SYNC_INIT, total_blocks, total_blocks, 2);
  INIT_SYNC(SYNC_READY, total_blocks, total_blocks, 3);
  INIT_SYNC(SYNC_MAGIC, 1, 1, 4);
  if (main_table == NULL) {
    SPAWN(4);
  } else {
    INVOKE(NUM_NODES>1 ? 1 : 0, block, (blocksize << 16) | blocks_per_row, 0,
           TO_GLOBAL(main_table), SLOT_ADR(0), NULL, SLOT_ADR(0), NULL,
           NULL, NULL, TO_GLOBAL(&magic_num));
    printf("Starting computation of %d x %d result\n", size, size);
  }
  END_THREAD();

THREAD_1:                          /* Spawned when someone's malloc fails */
  printf("Not enough memory\n");
  SPAWN(4);
  END_THREAD();

THREAD_2:                          /* Spawned after all blocks initialized */
  for (i = blocks_per_row - 1; i < total_blocks; i += blocks_per_row)
    RSYNC(main_table[i].slot + SYNC_INIT); /* Sync right end in each row */
  END_THREAD();

THREAD_3:                          /* Spawned after multiply finished */
  RSYNC(main_table[total_blocks - 1].slot + SYNC_MAGIC); /* Get magic # */
  END_THREAD();

THREAD_4:                          /* Spawned after magic number computed */
  printf("Magic # is %f\n", magic_num);
  RETURN();
}
```

Figure 6.17: Threaded-C Code for Matrix Multiply (MAIN Procedure)

6.2.4 Mutual Exclusion in Threaded-C

The last example of Threaded-C shows one way that new primitives can be made available to the programmer even if they are not currently supported in Threaded-C. Currently, Threaded-C does not provide direct support for the implementation of mutual exclusion based on atomic operations.[4] However, there is a straightforward implementation of mutual exclusion using the currently available primitives in Threaded-C. This implementation assumes that the procedure containing the mutual exclusion computation consists of a single fiber, and that the shared variables used in the mutual exclusive computation are not modified by any other procedure.

The mutual exclusion code presented in this section also assumes that the Threaded-C implementation adheres to the following assumptions:

1. Fibers are non-preemptive.

2. Only a single fiber can run on a node at a time.

Observe that while condition 1 is inherent to the EARTH model, condition 2 might not be valid in future implementations (in SMP clusters for example). However, it is true for all current implementations of EARTH systems.

We illustrate the use of mutual exclusion assuming a "black board" multiprocessing organization in which each individual node performs its computation and occasionally reports its intermediate results to a shared data structure. The reporting node receives an answer with the best value reported so far. If a node reports the best value, it will receive its own value back. The mutual exclusion is necessary to ensure that the updating of the shared value is an atomic operation.

We choose to use static variables to store the shared memory values that must persist throughout the computation. The drawback of this implementation is that the address of this variable needs to be passed as an argument to all procedures that interact with the corresponding mutual exclusion region. Alternative implementations could use dynamic allocation in the heap of the node that is in charge of the mutual exclusive computation or variables in global scope.

The mutual exclusion is based in the fact that a single node, called the MUTEX_NODE in our code, is chosen to process all the mutual exclusive computation.

[4] However, atomic operations on shared variables are supported in EARTH-C [52].

```
#define  MUTEX_NODE  4                                    /* Line 1 */

THREADED MAIN (int argc, char** argv)
{
   SLOT    SYNC_SLOTS[3];
   int     i;
   int     final;
   int     *GLOBAL shared_value;

   INIT_SYNC(0, 1, 1, 1);
   INIT_SYNC(1, NUM_NODES, NUM_NODES, 2);
   INIT_SYNC(2, 1, 1, 3);
   INVOKE(MUTEX_NODE, initialize_shared_value, 0,     /* Line 13 */
          TO_GLOBAL(&shared_value), SLOT_ADR(0));
   END_THREAD();

THREAD_1:
  for(i = 0; i < NUM_NODES; ++i)
    INVOKE(i, produce_value, MUTEX_NODE, shared_value, SLOT_ADR(1));
  END_THREAD();

THREAD_2:
  INVOKE(MUTEX_NODE, print_shared_value, shared_value, SLOT_ADR(2));
  END_THREAD();

THREAD_3:
  RETURN();
}
```

Figure 6.18: Threaded-C Code for Mutual Exclusion (MAIN Procedure)

Because we assume that at most one fiber can be active in each node, mutual exclusion is guaranteed. All the nodes that need to interact through mutual exclusion must invoke the corresponding fiber in the mutex node.

Figures 6.18–6.20 present the code that implements a mutex synchronization mechanism for a black board architecture using standard Threaded-C primitives. The first figure shows the MAIN procedure, which illustrates use of the mutex procedures (Figures 6.19–6.20):

Line 1: This specifies that node 4 is in charge of the mutual exclusion region processing.

Line 13: This invokes a fiber to initialize the shared value in the node that is the bookkeeper of the shared value (mutex node). We have to pass to the procedure the address of our pointer that will store the global address of the

```
THREADED produce_value(int mutex_node, int *GLOBAL shared_value,
                       SPTR produced)
{
  SLOT   SYNC_SLOTS[1];
  int    local_value;

  INIT_SYNC(0,1,1,1);
  local_value = NODE_ID;
  INVOKE(mutex_node, mutex_update, local_value, shared_value,
         TO_GLOBAL(&local_value), SLOT_ADR(0));
  END_THREAD();

THREAD_1:
  RSYNC(produced);
  END_FUNCTION();
}
```

Figure 6.19: Threaded-C Code for Mutual Exclusion (Produce Value)

shared variable. The initializing procedure will write to our pointer. Therefore, we can pass this global address to all the procedures that use the mutual exclusion region.

Fiber THREAD_1: This invokes the procedure that produces new values in each node in the system, this procedure receives the number of the mutex node, and a sync slot to be synchronized at the end of the computation. Remember that shared_value contains now the global address of the shared value.

Fiber THREAD_2: This is spawned when all the nodes have reported (through synchronization of slot 1) that they have finished performing their computations. We now ask the mutex node to print the final shared value.

Fiber THREAD_3: This ensures the computation does not terminate before the final value is printed.

The procedure **produce_value** (Figure 6.19) illustrates how a new local value is reported to the node that keeps the shared value. We are assuming that we also want a reply with the current shared value before we proceed with the computation. When fiber THREAD_1 is spawned, the variable local_value has the most recent shared value reported by the mutex node. We could now proceed with local computation using this new value, though in this example, we simply report to the node that invoked this procedure.

```
THREADED mutex_update(int new_value, int GLOBAL *shared_value,
                      int GLOBAL *reply_address, SPTR reply_slot)
{
  printf("Value %d was reported. ",new_value);
  if(*shared_value < new_value)
    *shared_value = new_value;
  printf("Mutex node replied with %d.\n", *(int *)TO_LOCAL(shared_value));
  DATA_RSYNC_L(*shared_value, reply_address, reply_slot);
  END_FUNCTION();
}

THREADED initialize_shared_value(int initial_value,
                                 int GLOBAL **shared_value,
                                 SPTR initialized)
{
  static my_shared;

  my_shared = initial_value;
  DATA_RSYNC_G(TO_GLOBAL(&my_shared), shared_value, initialized);
  END_FUNCTION();
}

THREADED print_shared_value(int *GLOBAL shared_value, SPTR printed)
{
  printf("Final maximum value: %d.\n", *(int *)TO_LOCAL(shared_value));
  RSYNC(printed);
  END_FUNCTION();
}
```

Figure 6.20: Threaded-C Code for Mutual Exclusion (Other Procedures)

The **mutex_update** procedure (Figure 6.20) contains the computation to be done in the mutually-exclusive region. In this simple example we just maintain the maximum value reported by all the nodes in the shared variable. We report the new maximum value to the node that invoked this procedure.

The **initialize_shared_value** procedure (Figure 6.20) must allocate all the static variables necessary for the computation in the mutual exclusion region. The address of the allocated variable is returned to the main procedure using the DATA_RSYNC_G primitive. These addresses will be passed to all the nodes that interact through the mutual exclusion region.

Chapter 7

Implementation of EARTH on Off-the-Shelf Multiprocessors

The first step in the evolution of EARTH toward a full-custom implementation is its emulation on an existing multiprocessor. Most such machines have commodity microprocessors with minimal support for parallelism, and no special support for the kinds of operations needed by EARTH. Therefore, EARTH operations must be emulated in software. This chapter describes several systems implemented in this manner.

To implement EARTH on an off-the-shelf multiprocessor, we begin with the following:

1. A collection of nodes, each with one or more processors, and each with its own memory (if a distributed-memory or distributed shared-memory machine).

2. A communications network. EARTH operations involve their own special messages, which are usually quite short, so it is better if the EARTH implementation can get access to low-level networking commands rather than try to use a higher-level communications layer. However, implementations have been done both ways.

3. A uniprocessor compiler for the processors. Since writing an effective compiler for modern processors takes considerable effort, EARTH can be implemented far more quickly by wrapping translators around existing compilers. Unfortunately, compilers written expressly for conventional multiprocessing

are not very useful because their programming models are not compatible with EARTH's.

From this, a complete implementation of EARTH should provide:

1. A *compiler* for translating high-level code based on EARTH (such as Threaded-C) into an executable form which includes the EARTH operations. Our compiler is known as the *EARTH Threaded-C Compiler.*

2. Software for emulating the SU tasks and other EARTH operations (the *EARTH Runtime System* or RTS). This is generally fixed and runs alongside the compiled applications code.

3. Utilities for downloading and starting the executable);

The bulk of our research has involved a multiprocessor called the MANNA. The first section describes this machine and two implementations of EARTH (based on EVM-A and Threaded-C). These are called EARTH-MANNA-D and EARTH-MANNA-S, though they are so similar that the label EARTH-MANNA is used when discussing their common elements. This section also presents experimental results showing the performance, on both systems, of individual features of the system and of a set of benchmarks. The second section presents a simulator for the MANNA called SEMi, used for performing further experiments, including increasing the number of processors and improving processor performance. Experimental results for these implementations are given as well. The final section briefly lists implementations of EARTH on other multiprocessors.

7.1 Implementation of EARTH-MANNA

The MANNA 2.0 (Massively parallel Architecture for Non-numerical and Numerical Applications) was developed at GMD-FIRST in Berlin, Germany in the early 90s [17]. A MANNA node, shown in Figure 7.1, contains two 50-MHz Intel i860XP RISC processors [66], each with 16 Kbytes of on-chip data cache and 16K of instruction cache.[1] The two processors share 32 Mbytes of DRAM on a common bus, and

[1]The i860XP is part of a small family of processors. In this writing, the label "i860" is used when describing properties common to all members of the family, while "i860XP" refers specifically to the chip used in the MANNA.

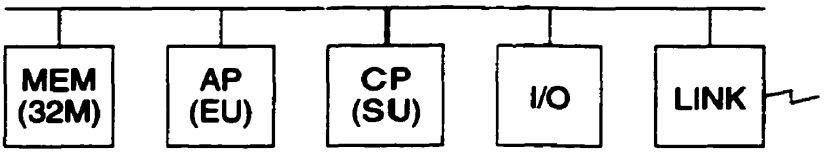


Figure 7.1: One MANNA Node

stay coherent with this memory and each other using bus snooping and the MESI protocol. The bus also runs at 50MHz, and memory access is fast compared to the processor; a cache miss takes 12 cycles (240ns) to load the line (a load followed immediately by a use of the loaded register imposes a 9-cycle penalty.) There is no L2 cache. The bus also connects to serial I/O devices and, on node 0, an Ethernet port for outside communication only.

Each node has a custom-designed *link chip* providing an interface between the interconnection network and the memory bus through memory-mapped I/O addresses. The link chip can transfer a byte to and from the network every bus cycle, at the same time if necessary, giving it a total unidirectional bandwidth of 50 megabytes per second (100 MB/s bidirectional). The bus interface to the link chip is 64 bits wide, and the link itself has internal buffering, making it possible to transfer blocks to and from the network interface in bursts. The link chip has 9 cycles (180ns) internal delay in each direction.

MANNA computers with more than 2 nodes are connected using custom-designed 16 × 16 packet-switched crossbars. Each input port can accept one data byte per 20ns cycle, and the input is buffered by a FIFO. The bandwidth of the crossbar is 800 MB/s if all 16 inputs are in use and each transmits to a different output port. The latency through the crossbar of a new packet is between 10 and 25 cycles, provided the requested output node is not in use. For machines with more than 16 nodes, the network can be organized as a hierarchy of crossbars. The most common configuration available, the one used for most of our experiments, is a 20-node MANNA consisting of two crossbars with 10 nodes apiece and two bidirectional links between them.

The MANNA is a standalone system intended for dedicated single-user parallel applications. There is a small operating system for this machine, but the EARTH system doesn't use it. Instead, system functions are linked with the executable.

Executables are downloaded from a host machine which also acts as a file server to the MANNA. The MANNA system includes a commercial uniprocessor compiler (written by the Portland Group, Inc.) which can exploit some of the more complex features of the i860, such as explicitly-pipelined floating point instructions and a switchable VLIW mode known as Dual Instruction Mode. This compiler is at the heart of our Threaded-C compiler.

7.1.1 Dual-Processor-Node Version of EARTH-MANNA

The two-processor structure of MANNA closely matches the EARTH architecture model, permitting an efficient emulation of EARTH on the MANNA system. This is called EARTH-MANNA-D, and was the first working implementation of EARTH. One processor (called the *Application Processor* or AP in the MANNA) is assigned the task of emulating the Execution Unit, the other (the *Communication Processor* or CP) runs the RTS, i.e., emulates the Synchronization Unit. The CP regularly polls the link chip for incoming messages, and sends outgoing requests to this link. The Event Queue and Ready Queue are stored in memory shared by the two CPUs.

The SU's tasks are well-defined (see Section 4.2.5) and fixed; they don't change from one application to the next. Therefore, the RTS code is the same no matter what runs on the AP. While most of the RTS runs on the CP, some of the tasks belonging to the SU (according to the architecture model) are actually performed by the AP. Frame allocation/initialization is one example. This simplifies code generation.

To repeat from Section 4.2.5, the functions of the RTS are:

1. EU and network interfacing,
2. Event decoding,
3. Sync slot management,
4. Data transfers,
5. Fiber scheduling,
6. Function invocation and load balancing.

Most of the RTS is a straightforward implementation of EVM-A and the semantics of the EARTH operations. Sync slots are stored in the frame (actually, they are declared as local variables in Threaded-C) at a known offset from the base address of the frame. The queues between the EU and SU are stored in memory. They are implemented as linked lists, and enqueueing and dequeueing are made atomic by changing single pointers, eliminating the need for locked access to the list.

Fibers are placed in the RQ in the order in which they are enabled. When the EU finishes a fiber, it reads the next (FID, IP) pair from the RQ, copies the FID to the register normally used for the frame pointer (normally r3 in the i860), and jumps to the instruction address. If the RQ is empty, the EU continues to check the queue periodically (there is nothing else for it to do). A procedure invocation is represented by a special record in the RQ. This record has a pointer to an invocation handler instead of an IP, and a pointer to an argument list instead of a frame identifier. The special handler copies the arguments to the local registers and then jumps to the initial fiber, whose address is included in the argument list. The initial fiber handles the frame allocation for that procedure. Deallocated frames are kept on special free lists for efficient reuse.

Because there is no operating system in the EARTH-MANNA, the RTS has control over the Memory Management Unit and the page tables of the virtual memory system. EARTH-MANNA takes advantage of this to make translation between global and local addresses automatic. The 32 megabytes of physical memory are located at the end of the physical address space. The virtual address space is divided into 128 32-megabyte blocks, numbered 0–127. In each node, two virtual blocks are mapped to the same physical block: the block with the same number as the node, and block 127. The latter is used for the replicated address space (the code, the stack, and the "global scope variables" of Threaded-C are located there). The former is used for dynamically-allocated memory including procedure frames. The heap boundaries are set so that the two areas avoid using the same physical space. Figure 7.2 illustrates the memory mapping for node 2. This figure also shows that some physical addresses in the bottom 256 Mbytes are reserved for the I/O and link chips. This means EARTH-MANNA can have no more than 120 nodes with this mapping scheme.

One important task of the SU is choosing where to run procedures that are invoked with the TOKEN operator, since that operator doesn't specify a node.

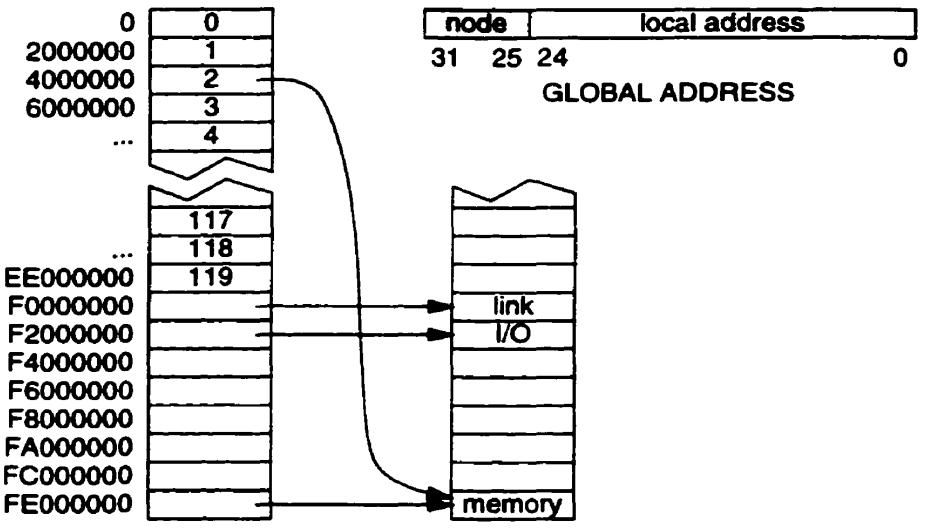


Figure 7.2: Memory Mapping for Node 2

This is an opportunity to improve runtime efficiency by balancing the workload evenly among the nodes. Once a procedure has started execution on a node, it is nearly impossible to move it or its frame, because other procedures may have global addresses pointing to the frame. *Before* the frame begins execution, however, the SU tries to move the TOKEN operator to a lightly-loaded node. Our method of *load balancing* is derived from the method of token management used in the ADAM dataflow architecture [82, 84].

Each SU has a queue of outstanding TOKEN operators, called the *Token Queue* (TQ). When the EU executes a TOKEN operation, the SU forms a "token" encapsulating the procedure call with its arguments, and puts it on top of the local TQ. When the RQ becomes empty, the SU removes a token from the *top* of the TQ and invokes the procedure locally. Note that putting locally-generated tokens on top of the queue and then removing tokens from the top results in depth-first traversal of the call graph. This generally leads to better control of functional parallelism, i.e., it diminishes the likelihood of parallelism explosion which can exhaust the memory resources of a node.[2]

If both the RQ and TQ are empty, the SU sends a message to a neighboring processor requesting a token, in effect performing *work stealing*. The neighboring processor, if it has tokens in its queue, takes one from the *bottom* of its queue and

[2]A similar technique was first used in the Concert system [49].

sends it back to the requestor. In this manner, breadth-first traversal of the call graph is implemented across processors, hopefully resulting in a better spreading of tasks. If the neighboring processor does not have any tokens to satisfy a token request, the neighbor's neighbor is queried, and so on. Either a token will be found, or the request cannot be fulfilled. Idle processors periodically query their neighbors for work.

Our subsequent experiments show that the quality of the load balancer can have a significant impact on the performance of divide-and-conquer programs such as N-Queens. However, there is a tradeoff, for the performance benefits of making an optimal choice where to run a token must be balanced against the costs of getting the information necessary to make that decision. Load balancing has been explored more thoroughly in some of the other EARTH implementations [19].

7.1.2 Single-Processor-Node Version of EARTH-MANNA

Since most multiprocessor systems have only a single CPU per node, an obvious question was whether the EARTH model can be implemented with competitive performance and efficiency on such machines, and what key issues and challenges must be addressed. To answer these questions, we did another implementation of the EARTH model on MANNA, called EARTH-MANNA-S, which uses only one of the two CPUs to perform the functions of both the EU and SU.

With only a single CPU to execute both the program code and the multithreading support code, it is necessary to find an efficient way to switch from one to the other. For performing synchronizing operations, initiating external requests, and even context switching this is not too much of a problem, for these occur where the EARTH operations are performed. Thus, it is only a question of replacing the EARTH operations with inline code to carry out the operations directly, rather than inline code to send the request to the EQ. For some simple operations, doing them inline in the EU may even take less of the EU's time than sending the request to the SU. Other aspects of the implementation are identical to the dual-CPU version, such as address mapping.

The biggest challenge with single-CPU nodes is external messages, such as remote function invocations, token requests, and SYNC_WITH_FETCH requests initiated remotely. Asynchronous reception of such messages has to be merged into the flow

of control dynamically, at run-time. (Others besides EARTH have encountered similar difficulties [108].) On most single-CPU machines, one of two mechanisms is used to handle incoming messages. Either there are explicit polls in the application code (added by the user or compiler), or incoming messages generate interrupts in the CPU. However, using either polling or interrupts exclusively can be inefficient, especially when message traffic is high and its irregular pattern is not statically analyzable. If polling is used, but the message frequency is much lower than the polling frequency, the polling overhead will be incurred many times for each message. If the polling frequency is too *low*, the CPU may take too long to respond to remote requests which may be on the critical path of the computation. If, on the other hand, interrupts are used and the message frequency is high, the interrupt costs can be significant, even if the interrupt handler is able to service all outstanding messages when it is invoked.

Our single-CPU version of EARTH-MANNA is implemented two ways. In the first implementation, the link is polled whenever the EU reads a new fiber from the RQ. This is a good time to poll the link since the EU already is switching contexts at this time, so there are no extra costs for saving registers as would be the case if the poll occurred in the middle of a fiber. However, the frequency of network events can vary dramatically from program to program and even during one program's execution, which means the response time to remote messages can be unpredictable. An alternative implementation augments this scheme with a *polling watchdog* [83], which interrupts the EU if it has gone too long without polling the link. Although the polling watchdog is not a standard part of the MANNA, it would be simple to build in hardware and can, in fact, be emulated using the second CPU of the MANNA.

7.1.3 Compiling for EARTH-MANNA

In order to minimize the design effort required, the code generator uses the PGI C compiler to do most of the actual compilation and code generation. There are code generators for both the dual-processor and single-processor versions, but these are almost identical, with differences only in the RTS libraries, some include files, and some of the inline code substitutions. Code is generated for EARTH-MANNA in stages, as shown in Figure 7.3:

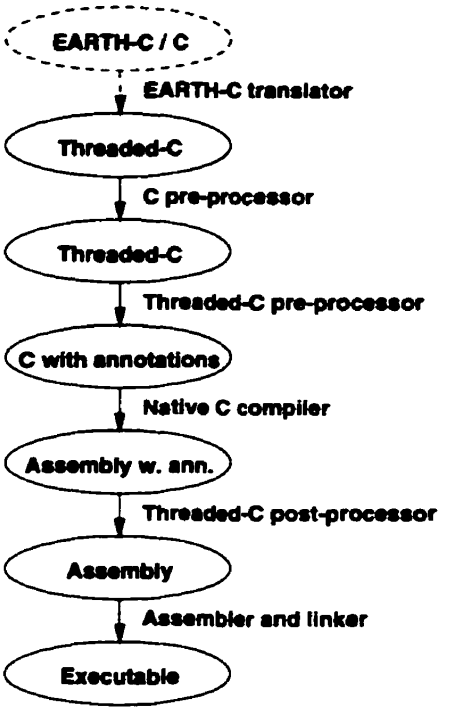


Figure 7.3: Compiling for EARTH-MANNA

1. The source code is either Threaded-C, or another high-level language which is translated into Threaded-C.

2. Threaded-C code is passed through a regular C pre-processor to expand macros, delete comments, etc. Also, some EARTH operations defined in Threaded-C are not implemented in the RTS, but simply are macros that call other EARTH operations. These substitutions are made here, using standard C include files.

3. The next stage uses a custom *Threaded-C pre-processor*. The main task here is to prepare the C code for processing by the standard C compiler. Threaded-C constructs must be converted into ordinary C constructs so that they can be recognized by the compiler. EARTH operations are converted into function calls. Fibers are converted into individual cases of a switch statement at the top level of a procedure. This makes it necessary to insert extra C statements to "fool" the compiler into not performing certain optimizations, such as common subexpression elimination, which would otherwise move code

from one fiber to another. Also, pragmas are inserted at various places in the code, including fiber boundaries, to "annotate" the code for the benefit of the post-processor.

4. The source code is now legitimate C code (albeit with annotations) and can be sent to the regular C compiler. The PGI compiler makes whatever optimizations it can within a fiber, including minimizing register use. The output is assembly code. Some of the assembly code consists of non-existent instructions produced by the pragmas.

5. The assembly code is read and interpreted by a *post-processor*, which recognizes and removes the illegal instructions and other false code stemming from the Threaded-C pre-processor, and replaces the function calls representing EARTH operations with inline code (e.g., to insert a request into the Event Queue in the case of the dual-processor version).

6. The final, pure assembly code is linked with other modules in the program and with the RTS.

One of the most important tasks of the post-processor is generating spill code. When the Threaded-C pre-processor converts the fibers into switch statement cases, the C compiler will naturally assume sequential execution when it performs def-use and liveness analysis. It doesn't understand that when the EU reaches the end of a fiber, the next fiber in the RQ may be in a completely different procedure, which might overwrite the live registers. The EU needs to save whatever registers are live in the current frame. The post-processor analyzes the code to determines exactly which registers are "live" at the beginning and end of each fiber, and inserts code for reading or writing these registers at the beginning or end, respectively, of a fiber. The number of registers needing to be saved between fibers is actually quite small, usually only 2 or 3 registers per fiber [85]. Optimizing in this way helps tremendously to produce faster code.

7.1.4 Performance of EARTH-MANNA

This section presents the main results of extensive performance studies involving both implementations of EARTH on the MANNA. The first part measures and compares the performance of individual EARTH operations using simple programs

designed to isolate specific costs. This is followed by a study involving application benchmarks written in Threaded-C and compiled for EARTH-MANNA. The following are our major observations:

1. The EARTH operations can be dispatched and executed reasonably quickly. The latencies and bandwidth achieved with EARTH-MANNA compare favorably with other parallel systems (Section 7.1.4.1).

2. Multithreading operations can be added without imposing large overheads on the code, provided the granularity of the code is not too fine. This is confirmed by comparing multithreaded code running on a single node to sequential code (Section 7.1.4.3).

3. This performance scales well as one moves from single processors to small-scale parallel systems. All benchmarks achieved near-linear speedups in the range of processor counts available on the MANNA (Section 7.1.4.4). (Larger configurations are considered in Section 7.2.2.)

From this we can conclude that the goal of effective parallel processing using off-the-shelf technology is achieved with the EARTH PXM and the MANNA implementation.

7.1.4.1 Performance of Individual Operations

The case has been made that effective use of large-scale parallel machines requires exploitation of fine-grain parallelism [22, 54]. Unfortunately, most conventional parallel systems discourage fine-grain code by imposing large overheads that can dominate such code. In many systems, multithreading is only supported at the OS level, and programmers must endure large OS overheads. In other machines, the OS guards access to the communications network, again imposing its extra costs on the programmer. If fine-grain parallelism is to be exploited well, it is crucial to reduce the costs of small operations, such as individual synchronization operations, small data transfers and procedure invocations.

The following tests measure the following characteristics of EARTH-MANNA:

1. Simple communication latency and bandwidth,

2. Latency and throughput of EARTH operations,

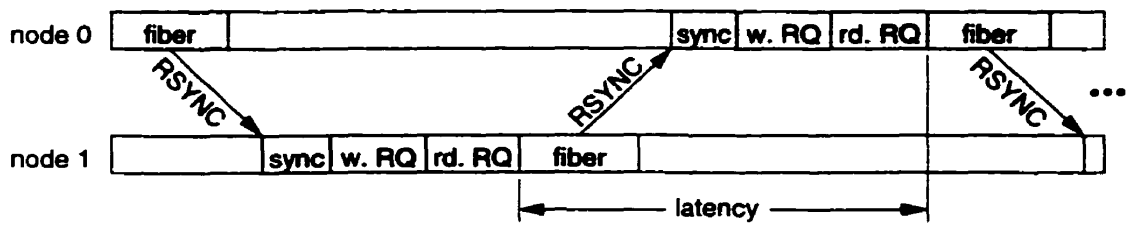


Figure 7.4: Measuring Latency on EARTH-MANNA

3. EU overheads of EARTH operations.

The first test is a simple measurement of raw latency and bandwidth. Two tests were performed on a 20-node MANNA using two nodes connected to the same crossbar (so that a message from one to the other passes through only one crossbar). Latency is measured by invoking a simple procedure on both nodes. Each procedure has a small fiber, which sends a sync signal (using the RSYNC operation) to a sync slot on the other node, whose IP field refers to the same fiber in the other procedure instance. The slot in each instance has a reset count of 1, so that the sync signal immediately enables the fiber. The nodes have no other work during this time, and are therefore idle when not running these fibers.

This is illustrated in Figure 7.4. Latency is measured by running the "pingpong" back and forth many times and dividing the total time by the number of iterations. As the figure shows, this measures not only the transmission time of the sync signal, but also the time it takes for the sync to have its effect, i.e., for the remote sync count to be updated, the fiber enabled, and the fiber read from the RQ. Thus, it is a practical measure of latency as seen by the program. This latency also includes the code at the start of the fiber, but this only consists of loading the remote sync slot address in preparation for the RSYNC operation; the address should be in cache and hence cost only 1–2 cycles. The end of the fiber tests the loop count, but this is overlapped with the transmission of the sync signal over the network, and hence does not contribute to the latency.

The latency costs for both the dual-processor implementation (separate processors for EU and SU) and the single-processor implementation (one processor for both) are listed in Table 7.1. To put this in perspective, these latencies can be compared with the measurements from a survey of parallel machines [28] in which messages were sent between neighboring processors (minimizing network delay). The

Parameter	Dual-processor	Single-processor
Latency (ns)	4091	2450
Latency (cycles)	204.5	122.5
Bandwidth (MB/s)	42.0	28.8
Bandwidth (% of peak)	83.9	57.5

Table 7.1: Latency and Bandwidth on EARTH-MANNA

only machine in the survey with a lower latency than the EARTH-MANNA-S is the shared-memory Convex SPP1200 (2.2μs). This machine runs at 120MHz, so its latency is 264 processor cycles, more than both versions of EARTH-MANNA. Even the slower EARTH-MANNA-D is surpassed only by the Convex and the shared-memory Cray T3D (3μs), but the Cray runs at 150MHz. The fastest distributed-memory machine in the survey has a latency of 10μs.

Note that this test measures latency that immediately affects the other node. Sync signals to slots with higher counts take less time. Also note that most of this 2–4μs can be gainfully utilized if other enabled fibers are waiting in the Ready Queues.

Bandwidth is measured with a similar pingpong technique, but the sync signals going from node 1 to node 0 are replaced with block moves (BLKMOV_RSYNC) of 1 million bytes apiece. Thus, the overhead of sending the acknowledgment sync signal back is insignificant to the total calculation. Table 7.1 shows high bandwidth utilization by both implementations. EARTH-MANNA-D beats EARTH-MANNA-S because the extra processor can work full-time on the block transmission. Most of the rather small loss (16%) is due to packetization overhead; large blocks are split into manageable packets, but each packet has a header. Furthermore, a new packet reaching the crossbar may incur a variable stall (200–500ns) due to the crossbar's arbitration scheme. Even so, both systems achieve a much higher fraction of their theoretical bandwidth than the low-latency systems in the survey (37% for the SPP1200 and 43% for the T3D)). The only systems with comparable bandwidth utilization rates have much higher latencies, such as the Meiko CS2, with a bandwidth of 43 MB/s out a possible 50 (almost identical to EARTH-MANNA) but a latency of 83μs.

Operation	Dual-processor nodes				Single-processor nodes			
	Sequential		Pipelined		Sequential		Pipelined	
	Loc.	Rem.	Loc.	Rem.	Loc.	Rem.	Loc.	Rem.
(r)sync	2327	3982	841	994	1000	2290	380	668
(r)spawn	2252	4266	N/A	N/A	920	2500	N/A	N/A
get_sync	2824	6968	1137	1880	1440	4666	700	1502
data_(r)sync	2767	6667	1060	1814	1280	4340	560	1200
invoke (1 arg)	5011	9011	3188	2794	2300	5360	1611	2165
invoke (5 args)	6217	10240	3879	2984	2460	5640	1768	2231
invoke (9 args)	6826	10727	4260	3504	3060	6500	2368	3165
invoke (18 args)	8192	12552	5529	4456	3220	7620	2528	3537

Table 7.2: EARTH Operation Latencies (nsec.) on EARTH-MANNA

These simple tests were followed by a more comprehensive set of experiments measuring the latencies of specific EARTH operations. Measurements were made using the same technique of repeating the operation many times and dividing the total time by the loop count. Latencies were measured for four basic synchronization operations: sending a sync signal, spawning a fiber directly, data read (SYNC_WITH_FETCH or split-phase transaction), and data write (SYNC_WITH_DATA). The data transfers were of single words. Measurements were also made for procedure invocations with four different argument counts. Like the previous experiments, these tests used only two nodes. However, these tests were made on a 2-node "MiniMANNA" in which the link chips of the two nodes are connected to each other directly, rather than to a crossbar. This removes the uncertainty of the crossbar delay.

Table 7.2 lists the latencies measured. For each operation, there are four values. Both local and remote versions of each operation are performed. Local operations stay entirely on node 0, while remote operations are executed on the EU of node 0 but involve some interaction with node 1. For example, "remote invoke" means invoking the procedure on node 1 using the INVOKE operation.

For both local and remote versions of each operation, there are two measurements, one for *sequential* latency and one for *pipelined* latency. Sequential latency is similar to the pingpong latency experiment; the code is designed so that the EU which executes the operation stalls until the operation finishes. Pipelined latency measures throughput; the EU executes the operation repeatly as quickly as it can.

The pipelined latency figures show the ability of the EARTH implementation to overlap long-latency operations with useful work (in this case, other EARTH ops). (Spawning is not pipelined because the immediate enabling of fibers would interfere with the pipelining of the SPAWN operations, making the timing meaningless.)

The results in Table 7.2 show relatively low latencies for EARTH operations. More importantly, they demonstrate the ability of the multithreading system to pipeline EARTH operations for greater throughput. Synchronizing operations (the most frequent EARTH operations as seen in our benchmark programs) have a pipeline throughput typically 3–4 times higher than pure sequential latency for remote operations, and 2–3 times higher for local operations. The pipeline depth is somewhat greater when there are separate processors for EU and SU, because there can be more "stages" in the pipeline. Nevertheless, even the EARTH-MANNA-S exhibits some pipelining for local operations, mainly because the RTS effectively combines the SU components of several EARTH operations into one unit, amortizing the costs of switching between the EU and SU contexts.

The tables show EARTH-MANNA-S consistently outperforming EARTH-MANNA-D, but this is partly due to the nature of the experiment, in which the EU doesn't do any computations except for the EARTH operations. The next experiment measures the costs to the local EU of various operations, meaning the amount of time taken by the EU to execute each operation. For EARTH-MANNA-D, this is the cost of forming a request message and writing it to the EQ in memory; for EARTH-MANNA-S, this is the cost of stopping and performing the entire operation (if local) or forming a request message and writing it to the link chip (if remote).

The results in Table 7.3 show that for remote ops, EARTH-MANNA-D can take advantage of a separate processor by having that processor send messages to the network, freeing the EU for other tasks. Overall, if most operations are local to the node, EARTH-MANNA-S should run faster. But this outcome should not be used to reject dual-processor configurations entirely. The i860XP is not designed for efficient processor-to-processor transfer of short messages, hence the software-based queues are cumbersome. A processor with better support for direct interprocessor communications should produce a dual-processor node that runs faster than a single-processor node in all operations. (The next chapter shows the simulation of a custom hardware SU producing significantly reduced latencies in transferring requests between EU and SU.) Furthermore, a separate SU processor has other benefits: its

Operation	Dual-processor nodes		Single-processor nodes	
	Local	Remote	Local	Remote
(r)sync	504	504	300	588
(r)spawn	721	580	323	640
end_fiber	530	N/A	441	N/A
incr_(r)sync	561	554	300	620
data_(r)sync	580	606	480	660
get_sync	580	620	620	700
invoke (1 arg)	760	620	479	806
end_procedure (1 arg)	794	N/A	760	N/A
invoke (5 args)	1039	907	599	936
end_procedure (5 args)	1203	N/A	800	N/A
invoke (9 args)	1223	1210	960	1406
end_procedure (9 args)	1372	N/A	1040	N/A
invoke (18 args)	1766	1512	1099	1670
end_procedure (18 args)	1728	N/A	1060	N/A

Table 7.3: EU Costs (nsec.) of EARTH Operation on EARTH-MANNA

ability to monitor the network independently improves performance for applications with widely-varying fiber lengths, as shown in the next section. Also, stress tests have shown that an independent SU can handle large increases in network traffic without significantly slowing down programs [58, 59].

7.1.4.2 Benchmarks Used

The remaining studies use real application programs written in Threaded-C and compiled for both versions of EARTH-MANNA. Table 7.4 lists the sequential running time of each benchmark under T_{seq}. This denotes the execution time of an optimized sequential version of the program running on one i860XP processor of the MANNA, i.e., without the execution of multithreading instructions (and their overheads). The code is compiled using the same PGI sequential C compiler as the core of the Threaded-C code generator.

Several programs representative of traditional scientific applications are tested to demonstrate that EARTH can handle large data-parallel applications. These all use static work distribution directed by the program (using the invoke operator) rather

Benchmark	Input	T_{seq} (sec.)	Description
FFT	2^{16}	0.866	Regular; frequent data moves
Fibonacci	30	0.969	Recursive; high overheads
Matrix multiply	512×512	36.6	Regular, data-parallel
N-Queens-P	12 queens	17.2	Fully para. recursive enumeration
N-Queens-T	12 queens	"	Partially sequentialized
Paraffins	$N = 23$	3.69	Recursive enumeration
Povray	shapes $(256)^2$	69.4	Task-parallel
Protein folding	$3 \times 3 \times 3$	7.43	Recursive search
SLT-2D	80×80	2.60	Regular, data-parallel
Tomcatv	$N = 257$	48.6	Regular, data-parallel, barrier
TSP	10 cities	38.2	Recursive search

Table 7.4: Benchmarks and Sequential Performance

than the automatic load-balancer. Fast Fourier Transform (*FFT*) is a regular application which requires frequent exchanges of data between distant points. Our FFT is a decimation-in-time algorithm written in a dataflow-like producer-consumer style in which elements involved in butterfly exchanges also exchange pointers to their neighbors, setting up the next butterfly (whose interval is twice the interval of the previous butterfly). *Matrix multiply* multiplies two 512×512 matrices using the code presented in Section 6.2.3, with a block size of 32. The time measurements, for both sequential and parallel code, excludes the time needed to initialize the matrix and compute the checksum. *SLT-2D* uses a semi-Lagrangian time discretization scheme to simulate a global atmospheric model in two dimensions; these experiments use 80×80 meshes. In *Tomcatv,* a floating-point benchmark from the SPEC89 and SPEC92 suites, each iteration updates a pair of 257×257 meshes, using independent nearest-neighbor calculations and calculations with horizontal loop-carry dependencies. Iterations continue until the maximum change drops below a threshold; this involves a global reduction and barrier (discussed in Section 4.3.1). Separate rows synchronize with each other and the top-level function using a dataflow-like paradigm.

The *Fibonacci* program is the code used for illustration in Chapters 4 and 6. While this is a contrived application, it is a useful benchmark for parallel systems because very little computation is done within the function body. Almost all the code is involved with function linkage. Since many of the overheads in parallel

programs involve interactions between different functions, this benchmark gives a good upper bound on the overheads encountered by a parallel system.

The *N-Queens* problem, a familiar benchmark that typifies searching problems, was presented in Section 6.2.2. Two versions are used in this study. *N-Queens-P* is the maximally parallel version presented in Section 6.2.2. *N-Queens-T* is a modified form of N-Queens-P in which the parallelism is "throttled" algorithmically, as shown in Figure 6.9. When the number of queens successfully placed on the board reaches a threshold (4 in this case), the program switches from parallel execution to sequential execution. Comparing the performance of these two benchmarks allows us to explore the tradeoff between expressing maximal parallelism in a program and restraining parallelism through program modifications. The latter is generally more difficult for the programmer (especially if good performance across a large number of machine sizes is desired) but generally yields better speedups by reducing overheads.

Paraffins is one of the four "Salishan problems" from the 1988 Salishan High-Speed Computing Conference, regarded as being challenging to parallelize [93]. This application enumerates all distinct isomers of each paraffin (molecule of the form C_nH_{2n+2}) of size up to a given maximum (23 in our experiments). The total number grows exponentially with this maximum. A paraffin is equivalent to an unrooted 4-ary tree; thus the problem is essentially the same as the problem of detecting isomorphisms in labeled free trees. The program generates lists of paraffins in a pointer data structure and returns an array filled with the number of distinct paraffins of each size up to and including the maximum. Parallelism is exploited by invoking functions on all the processors to compute the radicals (basic reaction units to form larger molecules) and then forming tokenized functions which compute the paraffins for the required sizes.[3]

The Persistence Of Vision Ray Tracing (*Povray*) program reads in a text file that describes the objects and lighting in a scene and generates a three-dimensional 256×256-pixel image using the ray-tracing rendering technique. This program has at its core a doubly-nested loop which iterates over each pixel. The Threaded-C implementation performs this loop in parallel. As the amount of work needed for each pixel can vary widely, this implementation takes advantage of EARTH's dynamic load-balancing mechanism.

The *Protein folding* chemistry application finds all possible polymers of a $3 \times 3 \times 3$

[3]The algorithm is explained in greater detail in the programmer's Master's thesis [30].

cube, where a polymer is defined as a chain of monomers and each monomer may represent a number of amino acids. It is assumed that each monomer is situated on a lattice point (i.e., a point on a 3-D grid). Each possible folding of the polymer can then be described as some path along the set of lattice points. Two kinds of parallelism are exploited in this program. The first is the loop-level parallelism exploited for all possible start paths provided in the original data-set using a dynamic token-driven mechanism. Another is the token-level medium-grained parallelism exploited at each lattice point for finding possible foldings using tail-recursive and token mechanisms.

For the Traveling Salesman Problem (*TSP*), a graph with 10 vertices is used. Branch-and-bound is used to exploit two types of parallelism in this program. The first is the loop-level parallelism exploited for all possible paths described in the original path-set, using EARTH's dynamic load-balancing mechanism to get good processor utilization. Another form of parallelism is the token-level medium-grained parallelism exploited for all arrived cities to find a minimal-cost Hamiltonian tour using a threshold-controlling mechanism.

7.1.4.3 Single-Node Performance

To measure the cost of multithreading in the EARTH-MANNA platform (and thus the EARTH model), the following experiments run the compiled multithreaded (Threaded-C) code on one processor, and compare this to the speed of the sequential code in Table 7.4. The ratio is the *Uni-Node Support Efficiency*, or "USE factor," of the platform, defined as

$$USE = T_{seq}/T_1 \tag{7.1}$$

where T_{seq}, as previously defined, is the best *sequential* running time, and T_1 denotes the execution time of a program written for the EVM (i.e., in Threaded-C) running on one processor node with multithreading support. Measuring this parameter guages how much processing power is lost due to multithreading overheads, while excluding losses which may be due to load imbalances among multiple nodes.

When running EARTH code on EARTH-MANNA, one may choose how many nodes to run. Of course, one may produce code which checks the number of processors assigned to the task and then simply branches to a sequential version of the program if only one node is used. However, these programs don't do this. They utilize multithreading operations as if they were run on multiple nodes; thus, the

Benchmark	USE factor (%)	
	Dual-processor	Single-processor
FFT	59.8	75.6
Fibonacci	7.55	13.9
Matrix multiply	99.9	100.3
N-Queens-P	52.5	67.0
N-Queens-T	98.8	99.3
Paraffins	91.4	99.4
Povray	94.0	100.0
Protein folding	95.0	98.8
SLT-2D	88.5	99.9
Tomcatv	95.0	100.0
TSP	98.9	99.6

Table 7.5: Uni-Node Support Efficiencies on EARTH-MANNA

version running a single node would execute the same number of multithreading operations no matter how many nodes are used.[4] A USE value close to unity for such programs can be attributed to many factors:

1. There are enough parallel threads to hide the latency of multithreading operations, i.e., there are enough fibers for execution while multithreading operations in other fibers are performed by the SU;

2. The total overhead for performing multithreading operations as seen by the EU is minimal (this can occur if the parallelism is sufficiently coarse-grained to keep the *number* of multithreading operations to a minimum);

3. The support for multithreading operations is not intrusive on sequential code.

The eleven benchmarks running on one processing node of the EARTH-MANNA platform produce the USE factors listed in Table 7.5. The results show that in most cases, executing multithreading instructions in a uniprocessor does not degrade performance very much. In some cases, they do not degrade at all![5] One could suggest

[4]Some programs deviate from this slightly. In Povray, for instance, a single block of data is initialized on one node and then copied to every node; each copy takes one EARTH operation. However, this is insignificant compared to the total number of multithreading operations executed in that program.

[5]This does not mean there are no multithreading overheads. Modern compilation is so complex that even minor code changes can lead to unexpectedly large changes in code performance due to

that the Threaded-C versions of the programs were well-written to minimize the use of multithreading operations, but what it also indicates is that the multithreading support does not have a huge negative impact on sequential code within a thread as in machines which require the interleaving of threads.

Only the Fibonacci benchmark does poorly. Its low USE factor is to be expected. The overheads of recursion far outweigh the actual computation performed even in the sequential code (one test per instance, and three adds per non-leaf instance). But function linkage is efficient in sequential code, especially if the compiler is aware of how little context there is in each function call (reducing the number of registers saved each call). The synchronizing operations of EARTH have to be more flexible than simple stack frame linkage, and consequently are far more expensive. When the dominant part of the code increases in cost, the USE factor suffers.

The fully-parallel version of N-Queens does moderately well, though on EARTH-MANNA-D it runs only half as fast as the sequential code. The throttled version achieves almost full performance. A count of procedure instances shows that the fully-parallel version generates over 1.6 million threaded procedure instances, though it takes less than twice as much time as the throttled code, which generates fewer than 10,000 such instances. (Of course, the use of a dequeue for the Token Queue, as described in Section 7.1.1, ensures that in both cases, not many of these instances are active at the same time.)

A comparison of the two N-Queens cases gives a glimpse of the tradeoff between expressing parallelism at a fine grain versus investing extra effort into programming to "coarsen" the threads. The throttled code represents how programmers can boost performance by controlling parallelism directly in the code, if they are willing to spend more time with the code (see Section 1.2.4). In this case, the modification is trivial and yields large improvements, so the outcome clearly justifies the extra effort. But the results show that letting parallelism "run wild" doesn't drag down performance too much.[6] This suggests that in cases where coarse-grain parallelism is more difficult to express without sacrificing scalability, the fine-grain alternative may run almost as well.

---

optimization anomalies. When the first Threaded-C version of Tomcatv was tested, it had a use factor of 1.07 [59]. Later analysis showed this was due to high register pressure in the sequential version (Tomcatv is notorious for this) which was relieved when the code was threaded. Subsequent rewriting of the sequential code reduced the USE factor to its current value.

[6]The next chapter shows that a hardware SU reduces the difference between the two considerably.

One final observation to make is that the implementation with single-processor nodes does consistently better than the dual-processor-node system. This could be predicted from Table 7.3, because all operations are local to the node (there is only one). That table also suggests that on multiple nodes, applications with many remote operations may run faster on EARTH-MANNA-D. This is verified in the next section.

7.1.4.4 Parallel Performance

The final experiments with the actual MANNA hardware involved running the multithreaded benchmarks from Table 7.5 on multiple processors and measuring the gain in speed over one processor. Two different measurements can be made. The *relative speedup* on k nodes is the speedup relative to the same multithreaded version running on a single node:

$$R_k = T_1/T_k \tag{7.2}$$

If the USE factor for this program is poor, then R_k can be a misleading measure of the multiprocessor's speed, though one that is often used [13]. An ideal linear speedup ($R_k = k$) would indicate that the work is distributed evenly. However, if multithreading the program has high overhead costs, the benefit would be much smaller. That is why we also report the *absolute speedup*, which compares the speed to the *sequential* version:

$$A_k = T_{seq}/T_k \tag{7.3}$$

Relative and absolute speedup are related by the USE factor:

$$A_k = USE \times R_k \tag{7.4}$$

Figures 7.5–7.10 show both absolute and relative speedups for the variants of EARTH-MANNA discussed in this chapter. The first two give relative and absolute speedups, respectively, for the MANNA with dual-processor nodes (EARTH-MANNA-D). The next two show the same for the single-processor-node version (EARTH-MANNA-S). The last two are also for EARTH-MANNA-S, but use the second CPU to emulate the watchdog timer (see Section 7.1.2).

The large amount of data necessitated using a 3-dimensional representation because many speedup curves would be superimposed in a 2-dimensional graph. One can best understand each curve by looking at the right endpoint, from which one

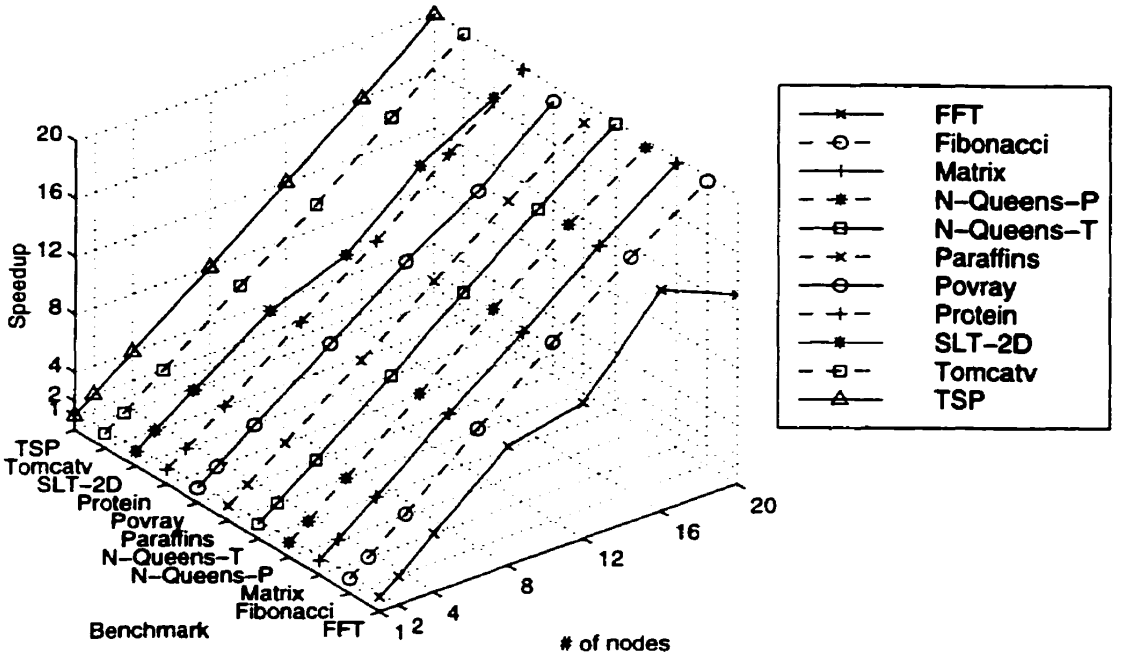


Figure 7.5: Relative Speedups on EARTH-MANNA-D

can read the speedup for 20 nodes. From there, the shape of the curve gives a good impression of the performance over the rest of the node counts.

For instance, it can be seen in Figure 7.5 that most of the benchmarks have relative speedups very close to 20 on 20 nodes on EARTH-MANNA-D; only SLT-2D and FFT has much performance loss at 20 nodes ($R_{20} = 16.7$). Also, the curves are almost completely linear, again with the exception of SLT-2D and FFT.[7] When multithreading overheads are factored into the picture (Figure 7.6), speeds for some of the benchmarks decline, especially for the worst-case Fibonacci example. Nevertheless, 7 of the 11 benchmarks still have speedups greater than 17 on 20 nodes. This means that for these benchmarks, each EU processor in a 20-node configuration is utilized 85% as much as a single processor running sequential code.[8]

[7]The dip at 12 nodes with SLT-2D is due to the static task distribution; the 80-row array is divided by rows, but 12 does not divide 80 evenly. The FFT code uses round-robin distribution of the elements, which works much better with powers of 2 because that causes many butterflies to be completely local (both elements in the exchange are on the same node).

[8]Of course, this ignores the second CPU, which doesn't contribute to this utilization figure. However, the use of a processor equal to the EU for SU functions is somewhat wasteful. Most of its capabilities, including built-in floating point, go unused. A small, cheap integer-only processor should work just as well.

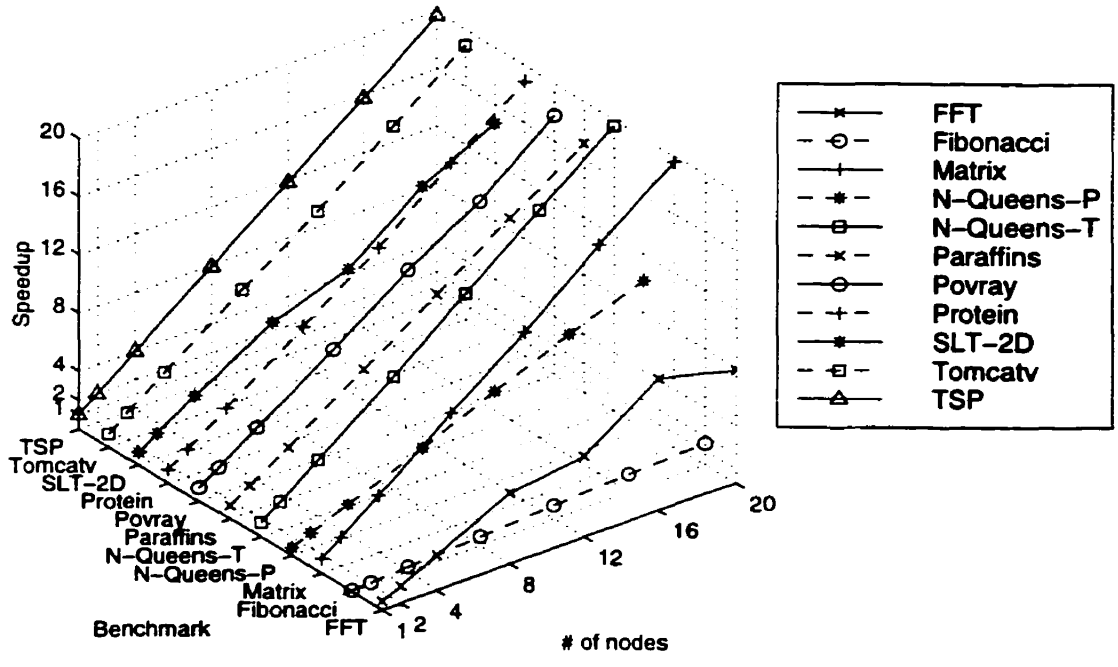


Figure 7.6: Absolute Speedups on EARTH-MANNA-D

Figures 7.7 and 7.8 show declines in performance for most of the benchmarks on single-processor-node machines with larger numbers of processors, even though the USE factors reported in Table 7.5 were better for MANNA-SPN across the board. While a single processor on a node may be sufficient for handling multithreading operations locally, it can slow down multiprocessors in several ways:

1. If there is a lot of interprocessor traffic (e.g., many small messages due to a large number of invocations, or some large messages due to large block moves), the traffic may be a heavy burden on the single processor, which must stop to read all of it, rather than letting the other processor respond to the messages.

2. If fibers are long, requests from other nodes may go unfulfilled for quite a while. The EARTH-MANNA-S RTS only checks for incoming messages when one fiber terminates and another is about to begin. Remote requests that are on critical paths could stall the entire computation.

The second problem, at least, can be ameliorated somewhat with a watchdog timer. The timer is supposed to monitor the link chip and interrupt the EU if an

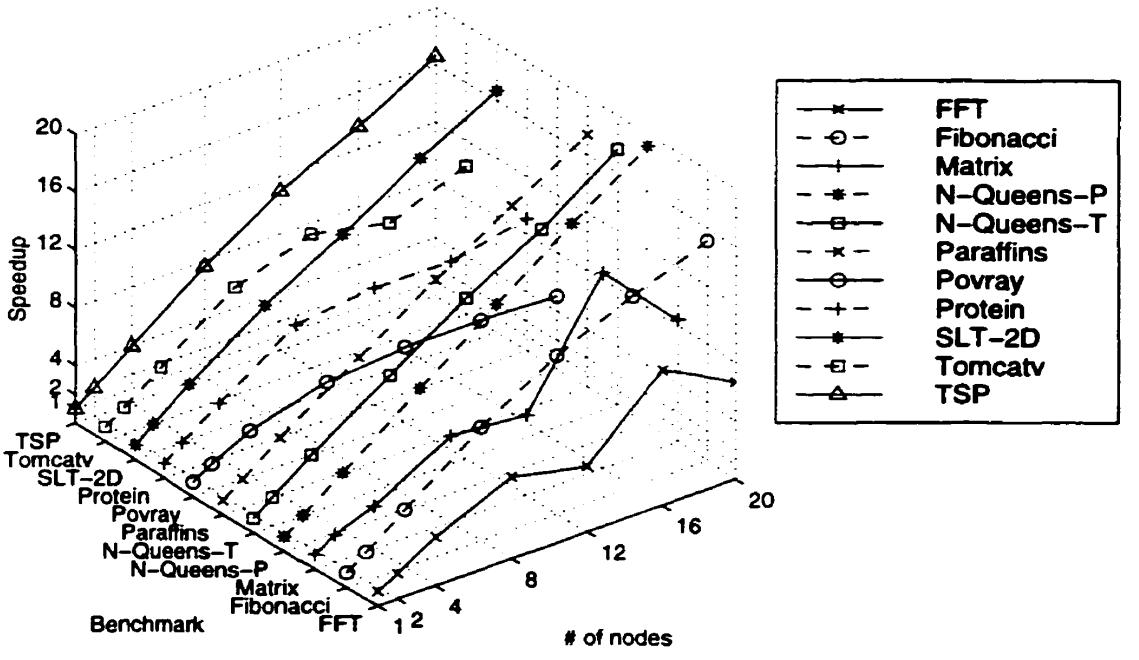


Figure 7.7: Relative Speedups on EARTH-MANNA-S

incoming message has waited at the link chip for more than a specified threshold ($50\mu s$ in these experiments). Lacking the internal hardware needed for this function, the EARTH-MANNA-S implementation does the next best thing by using the second CPU, now unused, to monitor the link and generate the interrupt. This is more coarse-grained than an ideal timer, for the CPU can't monitor the link continuously without tying up the system bus. Nevertheless, it gives a good approximation to what a real watchdog timer can do.[9] Figures 7.9 and 7.10 show the benefits of faster response to messages, particularly for the coarse-grained applications Povray, Protein and Tomcatv.

[9]See the original polling watchdog paper [83] for more details.

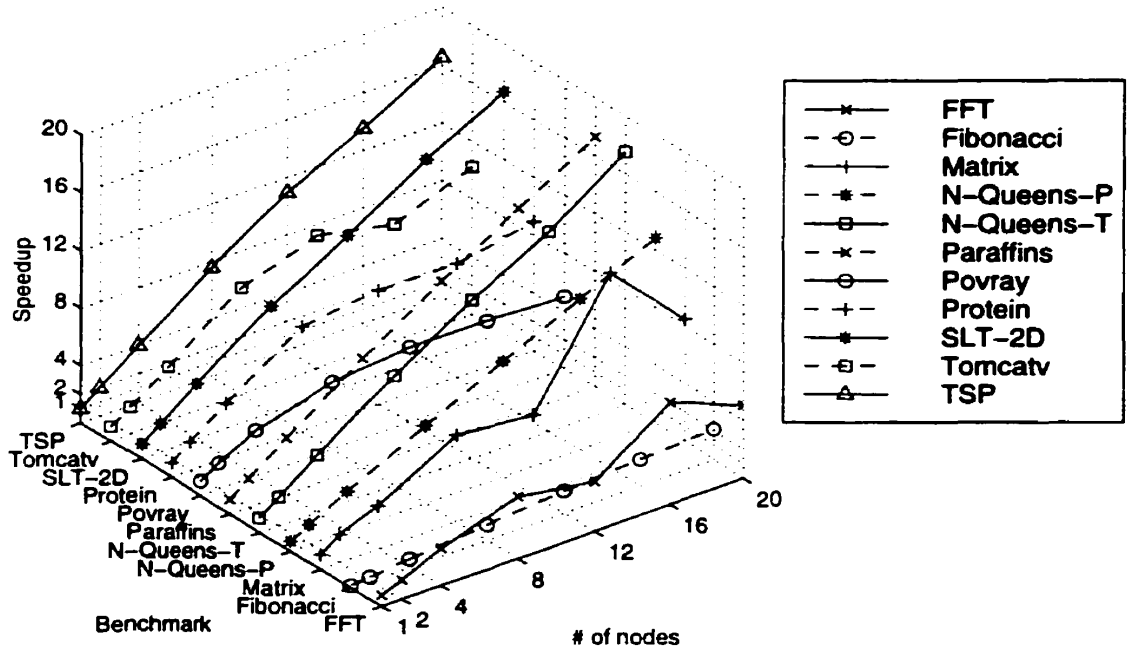


Figure 7.8: Absolute Speedups on EARTH-MANNA-S

7.2 Simulation of Alternate EARTH-MANNA Computers

This dissertation has proposed a series of design extensions to an off-the-shelf multiprocessor representing an evolutionary progression toward more customized machines executing EARTH programs. To demonstrate the benefits of each step in the series without actually building the custom hardware requires a simulator which can model each proposed machine. For this purpose, we have developed an *accurate* simulator for the MANNA hardware called SEMi (Simulator for EARTH, MANNA and the i860) [58, 112, 115]. SEMi is mainly intended for measuring the benefits of adding new modules to the basic MANNA configuration (e.g., a hardware SU), and these experiments are discussed in the next chapter.

However, we can also make modifications to SEMi's default MANNA configuration that don't involve adding new hardware. In this section, we use SEMi to simulate two extensions to the MANNA hardware that do not exist, but still represent conventional multiprocessors at the beginning of the evolutionary path. These

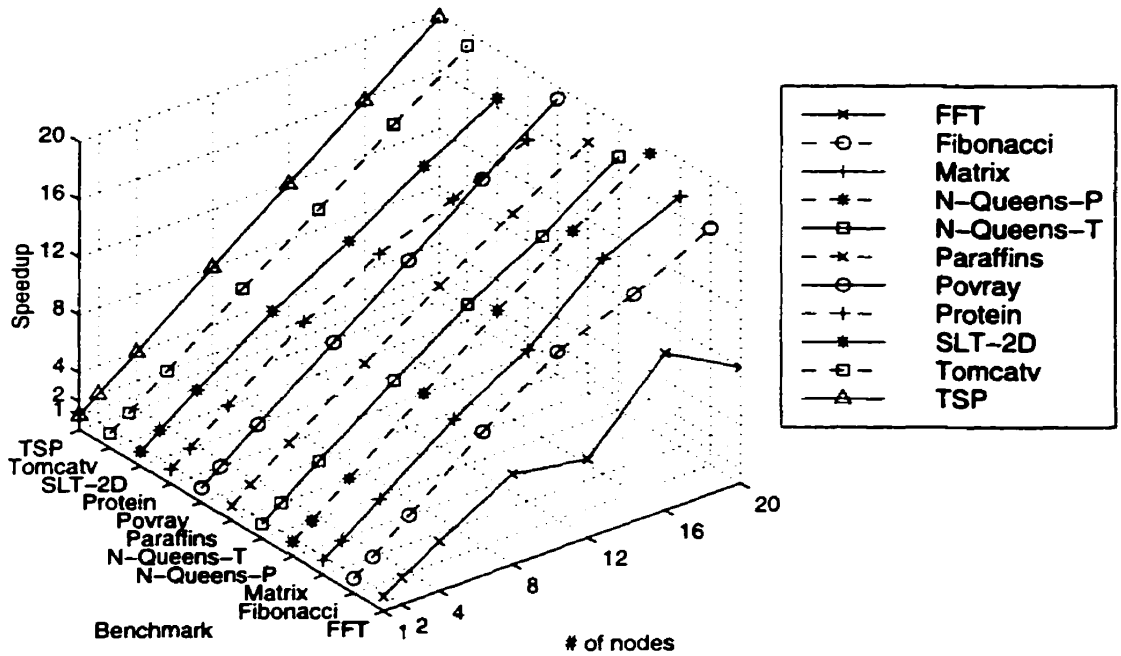


Figure 7.9: Relative Speedups on EARTH-MANNA-S with Polling Watchdog

machines run the same applications code as the EARTH-MANNA machines covered in the previous section. The first modification extends the existing MANNA machine from 20 nodes to 120 nodes. With this, the speedup curves from Section 7.1.4 can be extended to see if the EARTH model is scalable into the range of large-scale machines. The second modifies the performance parameters of the i860 to reflect some of the improvements in processor design made since the i860 was made. This is to test whether the good results obtained for EARTH on the MANNA may be applicable to other multiprocessors or are dependent on some specific performance characteristics of the i860.

The first part of this section describes SEMi, and the modified MANNA machines simulated using SEMi. The second part presents the experimental results.

7.2.1 The SEMi Simulation Testbed

SEMi (Simulator for EARTH, MANNA and the i860) is a functional simulator for i860-based uniprocessors and multiprocessors. It was originally inspired by DLXsim [53], and uses a similar interface. SEMi currently has modules for simulating the

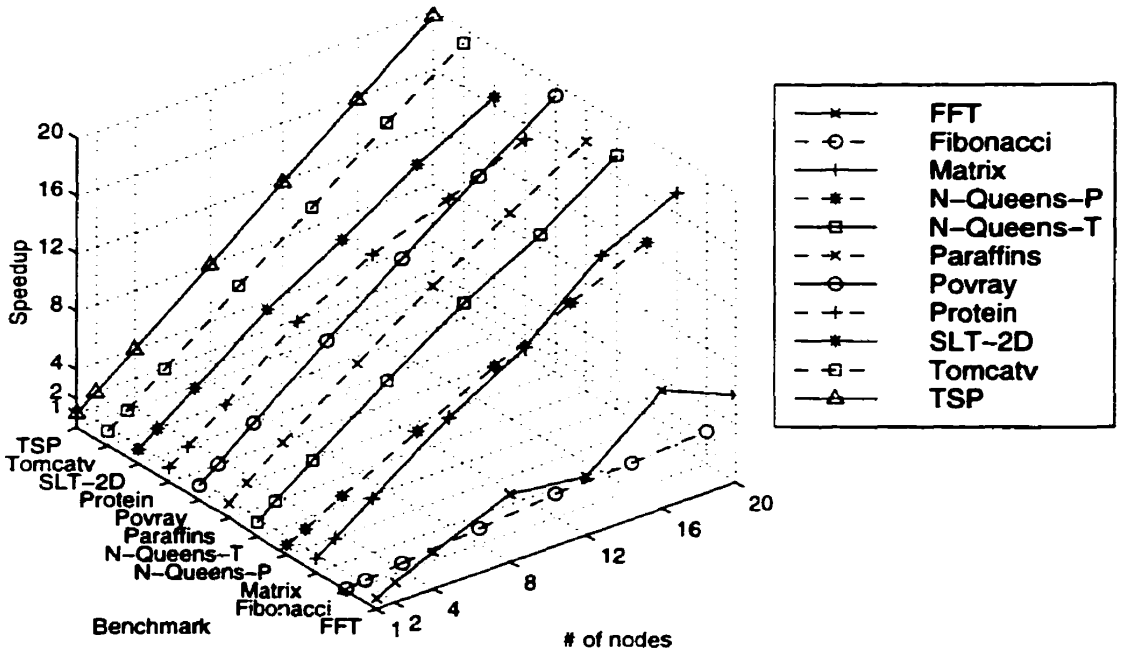


Figure 7.10: Absolute Speedups on EARTH-MANNA-S with Polling Watchdog

i860XP processor, the MANNA bus and memory system, the I/O and link chips, and the crossbars, i.e., all components of the MANNA, as well as extensions to the base MANNA (described in the next chapter).

SEMi is primarily a "functional" simulator, meant to product correct program results and a cycle count equivalent to a real MANNA. Therefore, SEMi only simulates the details needed to model the interactions between components that affect the program state or timing. The real behavior of individual circuits are not simulated (this simulation is not intended for synthesis). For instance, SEMi models the fact that instructions are fetched 2 cycles before they are executed. This is necessary for the correct modeling of delay slots and bus interactions. The order of operations in a pipeline affects the latter because bus cycles can be generated from both the fetch stage (I-cache miss) and the cache load stage (D-cache miss). However, the write-back stage isn't modeled since it doesn't affect the timing behavior (due to bypass logic).

The benefit of this approach is that SEMi is far faster than a logic simulation. A SEMi simulation of a MANNA uniprocessor takes about 300 processor cycles for each cycle in the simulated machine. If clock frequencies are normalized, a

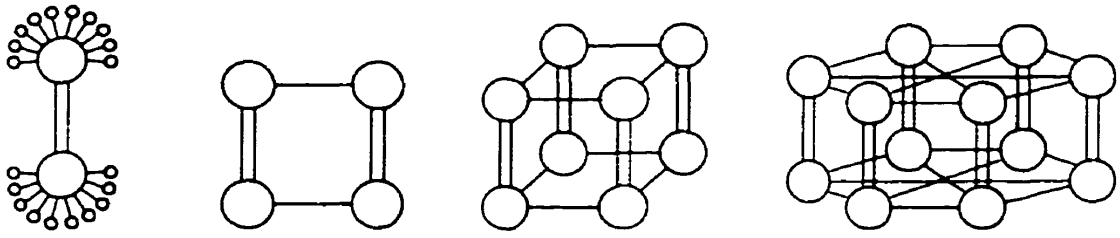
dual-processor node runs about 500 times slower. As nodes are added, the ratio grows slightly more than linearly, as extra crossbars are added and the expanding simulation state overwhelms the cache. Nevertheless, even a large simulated machine with n nodes ($n > 100$) runs no worse than $2000n$ times slower. Thus, we are able to simulate realistic problem sizes in which T_{seq} is around one minute.[10]

One criticism commonly leveled at simulators is that they are inaccurate and only reflect the wishes of their programmers. SEMi was designed for *accuracy* to avoid this objection. SEMi carefully models actual processor behavior in terms of cycle counts, and accounts for resource hazards, load stalls, and bus delays. SEMi has been run side-by-side with a real MANNA to measure its accuracy, in terms of the number of clock cycles reported, and its timing figures are typically within 0.2% of the timing reported by a real MANNA for sequential uniprocessor code, and within 5% for dual-CPU and multiprocessor runs.[11] This proven accuracy of our simulator makes us confident that if we make small modifications to the MANNA architecture, the timing measurements will be close to what we could expect if such modifications were made to a real MANNA.

One of the options of SEMi selects a topology from a file of predefined configurations. An arbitrary network of $n \times n$ routers, based on the MANNA's crossbar chip, can be defined. To extend the speedup curves of the EARTH benchmarks, we created several extensions of the standard topology. Figure 7.11(a) shows the 20-node topology which is standard in the MANNA. Two crossbars connect to ten nodes apiece, with two bidirectional connections running between the crossbars. If a node sends a packet to a destination that is not attached to the same crossbar, the route through the network is deterministic and determined by a static routing table, even if there are multiple paths to that destination. Figure 7.11(b) shows a 40-node network formed by tying two 20-node MANNA's together. (This topology has actually been constructed by GMD although it is never maintained very long and always subsequently broken back into 20-node machines.) Figures (c) and (d) show further extensions to 80 and 120 nodes, respectively. (120 nodes is the upper limit of EARTH-MANNA without a substantial rewrite of the RTS and compiler,

[10] A near-linear application speedup mostly offsets the growth in simulation time as more nodes are added.

[11] The greater error for multiple-CPU simulations is due in part to the greater complexity and unpredictability of these systems. Also, some non-determinism is to be expected from the real machine. However, repeated experiments on the real machine with a given benchmark and input show very little variance in running times.



a) 20 nodes b) 40 nodes c) 80 nodes d) 120 nodes

Figure 7.11: Large-Scale Topologies

for reasons given in Section 7.1.1.) All routing tables are deterministic and designed to be deadlock-free. (In parts (b)–(d), the 10 nodes connected to each crossbar have been omitted for clarity.)

Other options to SEMi change various parameters of the system (e.g., cache sizes, memory delays) or make small changes to the behavior of the processor, which by default emulates the i860XP as closely as possible. Although the i860 was a big step forward in RISC processing at the time it was released, some design compromises had to be made due to the lower level of integration at that time. Among these:

- There is no scoreboard logic for the floating point unit. Instead, there is a simple lock, which blocks access to the unit until a scalar FP op is finished, even if succeeding FP ops are independent.
- The L1 on-chip cache is blocking. If the CPU misses the L1 cache and starts loading a cache line, subsequent accesses to the cache are blocked until the load is finished, even if they would hit the cache.
- There is a simple form of dynamically-switchable VLIW processing called Dual Instruction Mode, but no other form of multiple-instruction issue.

Changes selected in the SEMi command line may alter the number of clock cycles needed to execute a given program, but normally do not change the functional behavior of the processor. Generally, a sequential program compiled for MANNA runs the same on both the base configuration and a modified configuration. A parallel program may run differently if the program is inherently non-deterministic, since the options may affect the relative order of two events on different processors.

For our last set of experiments we want to configure the MANNA and its underlying i860 processors to look more like the systems used today. The i860XP dates

back to the early 90s and only runs at 50MHz. To bring the i860-based experiments a bit closer to the processors of today, we configured SEMi to simulate a MANNA with the following processor and system changes:

- Processors have increased in speed faster than memory chips and buses. To account for this effect, we have slowed down the memory system and bus, but we have also added an L2 cache. The performance parameters of the new memory and cache are taken from the PowerMANNA, a successor to the MANNA also built by GMD. The PowerMANNA is based on a 200MHz PowerPC 620 processor instead of the i860.

- The limitations of the i860 have been removed where possible. The options make the cache non-blocking, add hazard detection logic to the FP unit, and allow multiple-instruction issue (though only *in-order* issuing is allowed at this time).

7.2.2 Performance of Larger EARTH-MANNA Systems

The first use of SEMi to enhance the existing MANNA architecture is extending the speedup curves of Section 7.1.4.4 to see how scalable are the current programs. Five of the benchmarks were chosen for further study. To measure how well EARTH scales for smaller problems, we ran simulations for smaller problem sizes for each of the benchmarks, as well as the problem sizes used in the real MANNA studies. Table 7.6 lists the benchmarks used, the problem sizes for each, the sequential running time, and the USE factors for EARTH-MANNA-D and EARTH-MANNA-S under these simulations.

Speedup curves for the five benchmarks are shown in Figures 7.12 through 7.24. The data from these experiments are shown in separate graphs for each benchmark/implementation combination, with each graph displaying relative and absolute speedups for all input sizes. This allows the impact of problem size on scalability to be viewed. Results for EARTH-MANNA-S are shown both without a polling watchdog (Figures 7.17–7.21) and with a watchdog (Figures 7.22–7.24), except that watchdog results are not shown for Fibonacci and N-Queens-P. Since the EARTH-MANNA-S polls the network at the end of each fiber, and the maximum fiber length in these two programs is far less than the watchdog timeout interval, the watchdog

Benchmark	Input	T_{seq} (sec)	USE factor (%)	
			Dual-processor	Single-processor
Fibonacci	15	0.000831	8.6	15.7
	20	0.00801	7.7	14.1
	25	0.0875	7.6	13.9
	30	0.969	7.6	13.9
N-Queens-P	8	0.0223	39.9	51.7
	10	0.541	46.8	56.1
	12	17.3	53.9	65.6
N-Queens-T	8	0.0223	68.5	78.5
	10	0.541	93.1	95.3
	12	17.3	99.1	99.3
Paraffins	18	0.0394	82.1	97.6
	20	0.228	85.4	101.4
	23	3.69	84.7	100.6
Tomcatv	33	0.721	89.3	92.2
	65	2.94	91.4	93.7
	129	12.0	93.2	95.6
	257	48.7	93.7	96.5

Table 7.6: Uni-Node Support Efficiencies on SEMi Simulation of EARTH-MANNA

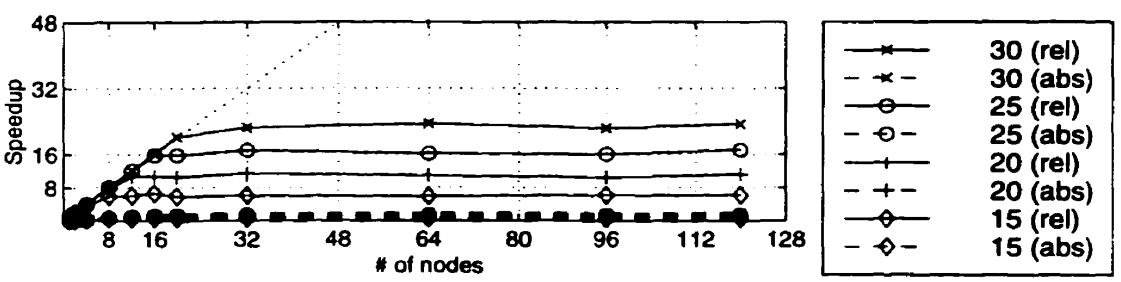


Figure 7.12: Speedups on EARTH-MANNA-D for Fibonacci

never generates an interrupt and the results are identical.[12]

Comparing these results with the timings from the real MANNA in Section 7.1.4 leads to several observations.

[12]This is not always true for the implementation on the real MANNA, due to the inexact nature of the timing as explained at the end of Section 7.1.4.4.

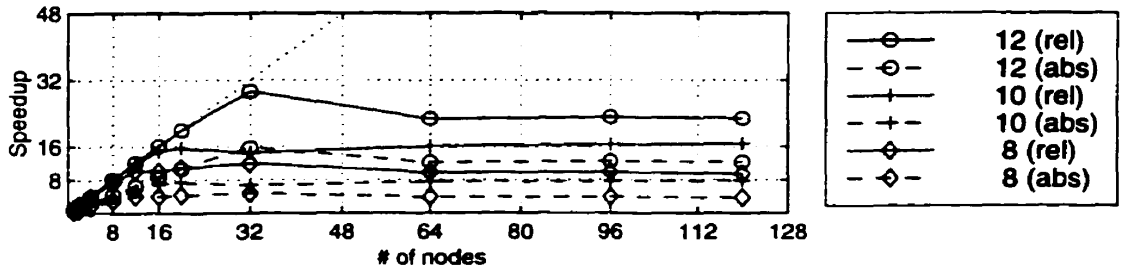


Figure 7.13: Speedups on EARTH-MANNA-D for N-Queens-P

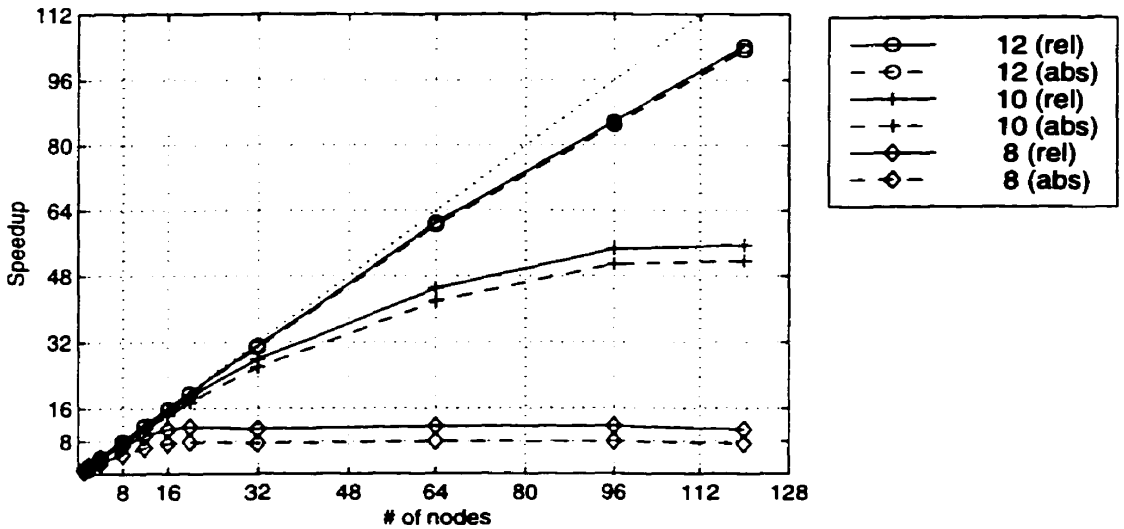


Figure 7.14: Speedups on EARTH-MANNA-D for N-Queens-T

Load Balancing: First, some of the speedups that looked so promising in Figures 7.5–7.10 fail to extend much beyond 20 nodes, particularly for the fine-grained applications (Fibonacci and N-Queens-P). This appears to be due to poor load balancing. This was determined by measuring the total time spent by all processors in different parts of the program. This breakdown is shown for N-Queens-P (10 queens) running on EARTH-MANNA-D (Figure 7.25) and EARTH-MANNA-S (Figure 7.26).

Five components are shown: the three fibers (fibers 1, 2, and the initial fiber), the sequential function safe( ), which is called from fiber 1, and idle time. Any time when an EU is not actively running a fiber (e.g., it reads the RQ when nothing

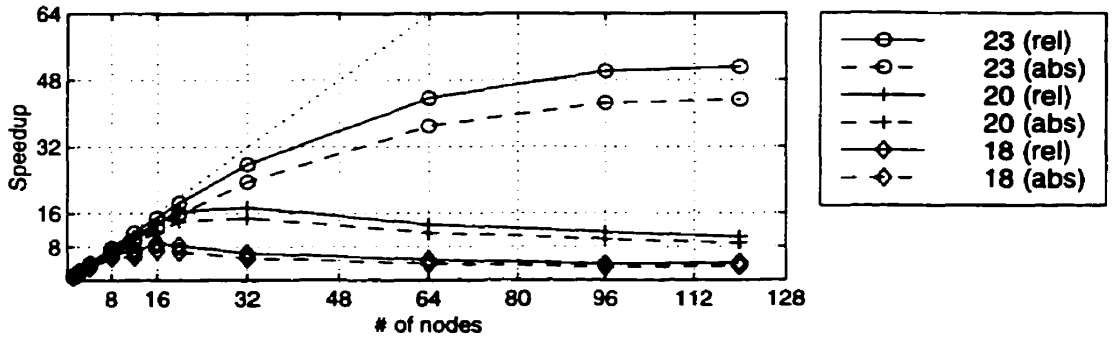


Figure 7.15: Speedups on EARTH-MANNA-D for Paraffins

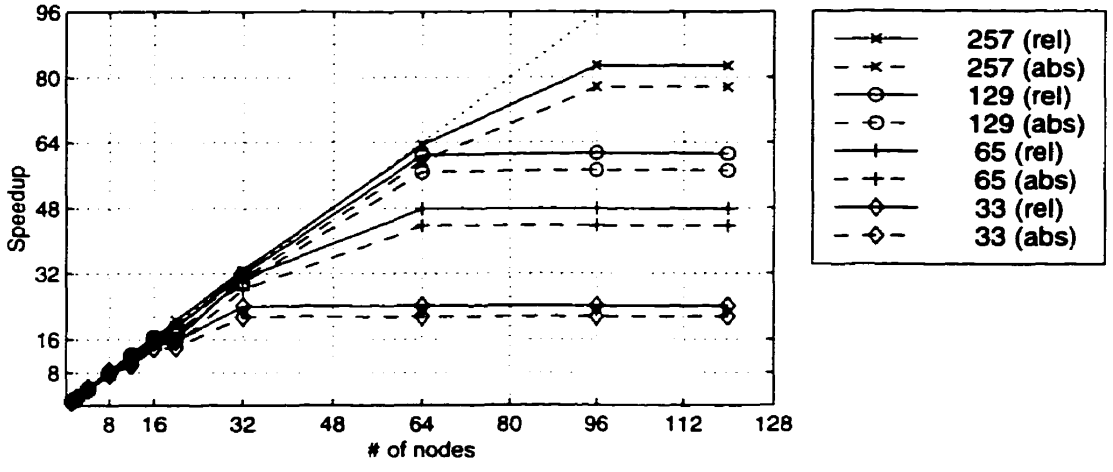


Figure 7.16: Speedups on EARTH-MANNA-D for Tomcatv

is there) is counted as idle time. Each graph shows the time totals for each node count in the experiments. For comparison, an analogous breakdown is given for the sequential code at the left side of each graph. In this case, since there are no fibers in the sequential code, the times shown for these three components are for the sections of the recursive function which correspond to the three fibers.

The graphs show that the safe( ) function (which is identical in the sequential and threaded codes) is used about the same amount of time in all runs. Each of the three fibers takes longer in the parallel code than the corresponding code fragment in the sequential code. This graphically shows the source of the USE factors observed for this benchmark. Once this overhead is paid, the total time spent in the fibers

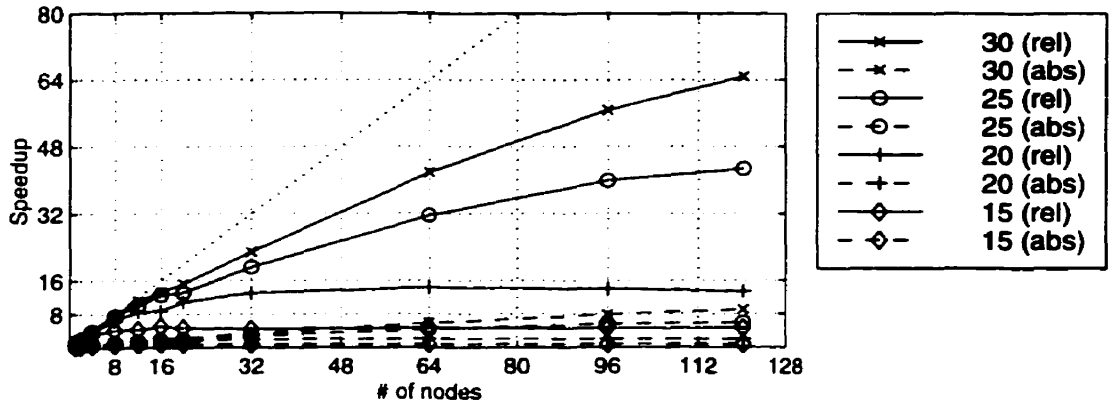


Figure 7.17: Speedups on EARTH-MANNA-S for Fibonacci

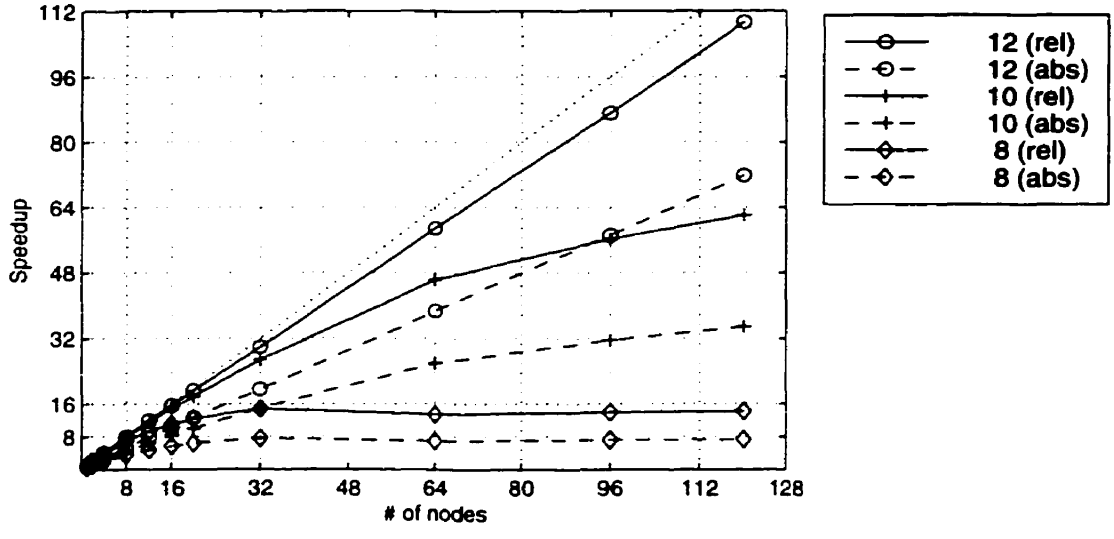


Figure 7.18: Speedups on EARTH-MANNA-S for N-Queens-P

stays fairly constant for all node counts. If these were the only costs, then the code should enjoy a linear relative speedup. Thus, the only cause of the poor speedups is the idle time, which grows rapidly, especially for dual-processor nodes.

This suggests there are shortcomings in the load balancer. The runtime system was originally developed for real MANNA machines, which have at most 40 nodes. These simulations used the same RTS, without any attempt to tune it for a larger number of nodes. More research will be needed to see if the load balancer can be

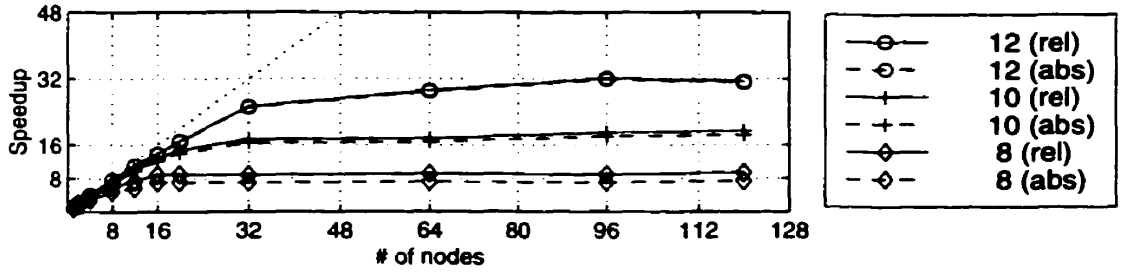


Figure 7.19: Speedups on EARTH-MANNA-S for N-Queens-T

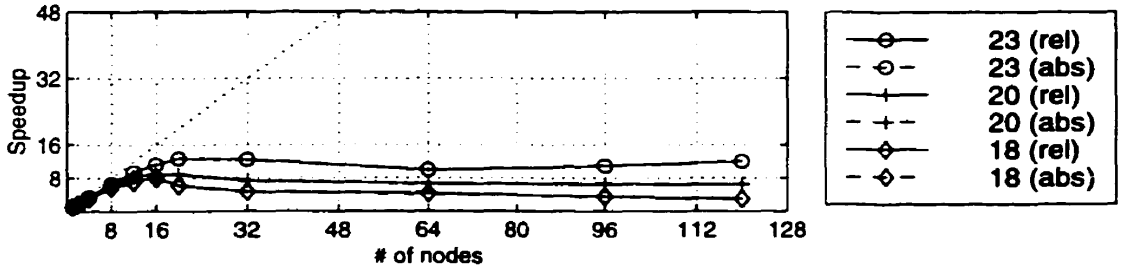


Figure 7.20: Speedups on EARTH-MANNA-S for Paraffins

optimized for larger node counts, and whether this tuning can be done automatically at runtime or should be accomplished by using different load balancers for different configurations.

The much better performance of N-Queens-T shows the benefits of reducing the

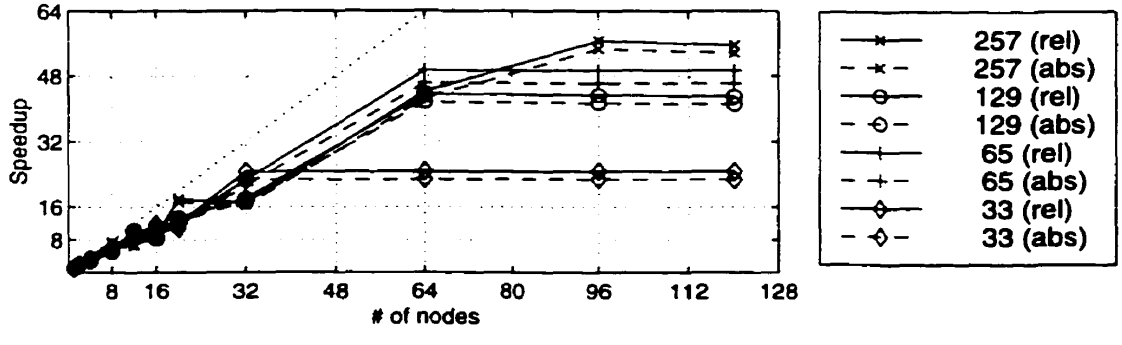


Figure 7.21: Speedups on EARTH-MANNA-S for Tomcatv

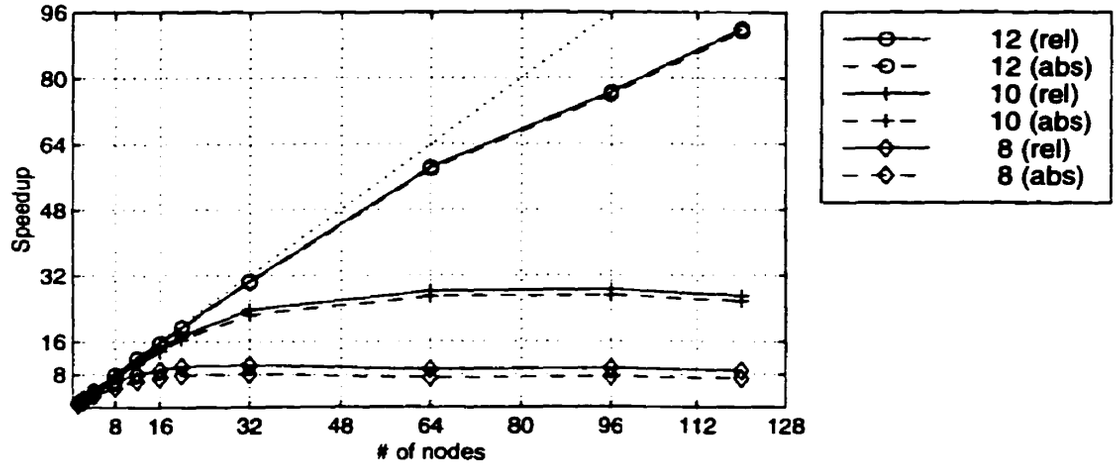


Figure 7.22: Speedups on EARTH-MANNA-S/Watchdog for N-Queens-T

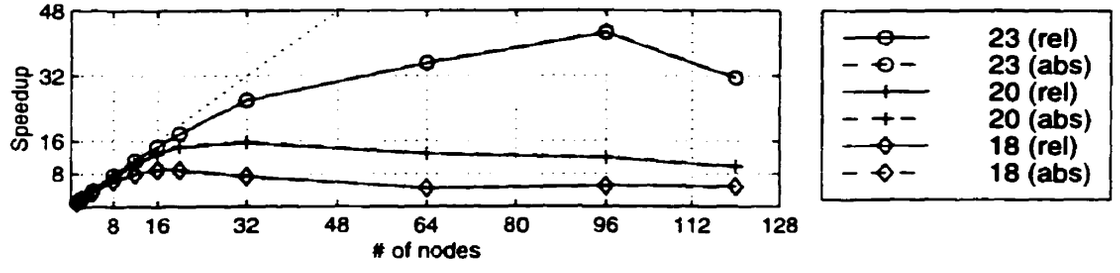


Figure 7.23: Speedups on EARTH-MANNA-S/Watchdog for Paraffins

number of fibers executed at runtime. On the other hand, the Paraffins application is even more coarse-grained, and does only moderately well for the largest problem size. Some effort was spent by the programmer to keep the number of threads low, in order to get a high USE factor by reducing multithreading overheads. The code was not modified for this set of experiments, so as the number of nodes increases, the number of procedure instances per node correspondingly drops. Since the work-stealing algorithm can't predict the running time of a token before the procedure begins execution, it does not work well when there are only a few tokens per node, because there are fewer opportunities for load imbalances to be corrected.

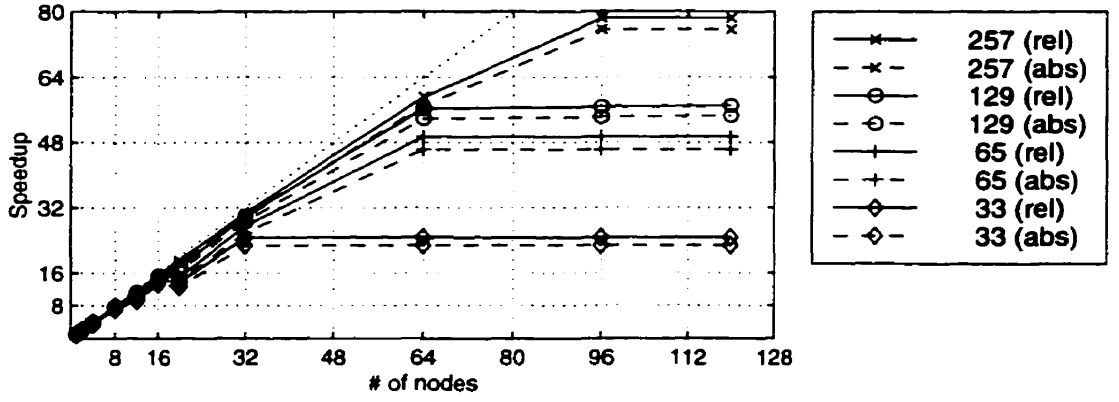


Figure 7.24: Speedups on EARTH-MANNA-S/Watchdog for Tomcatv

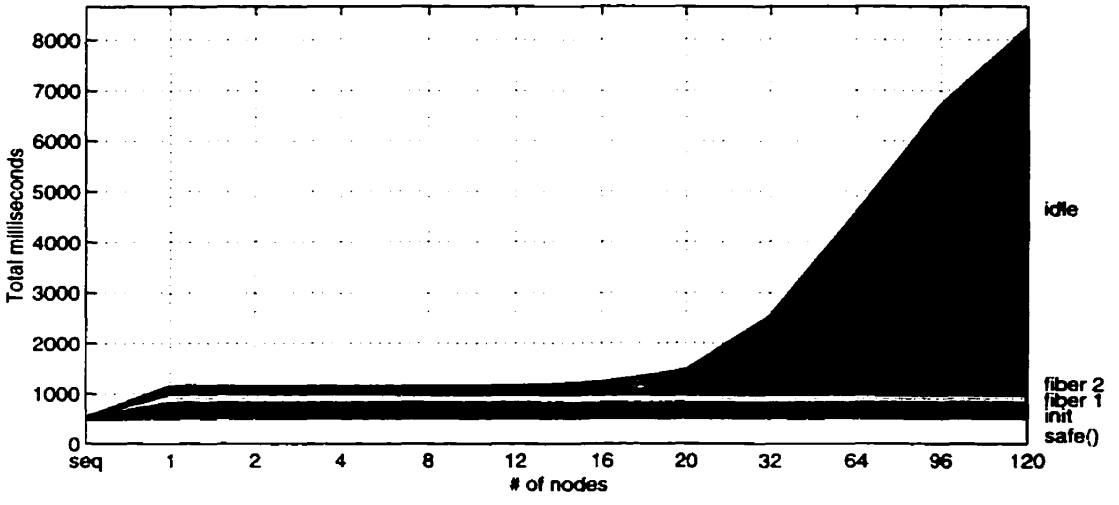


Figure 7.25: Breakdown of EU Use on EARTH-MANNA-D for N-Queens-P (10)

Benefits of Multithreading: The most coarse-grained application in this set is Tomcatv, in which the $N \times N$ matrix is divided into $N-2$ procedures, one for each row (excluding the top and bottom). For each row, three fibers are run in sequence for each of the 100 iterations in the program. The load is the same in each row, so the rows are distributed among the nodes statically and evenly. This places an upper bound on the speedup, and leads to uneven loads if the number of rows is not an even multiple of the number of nodes. For instance, Tomcatv with $N = 257$ (i.e., 255 active rows) should get a speedup of at most 85 (255/3) on 120 nodes, since at

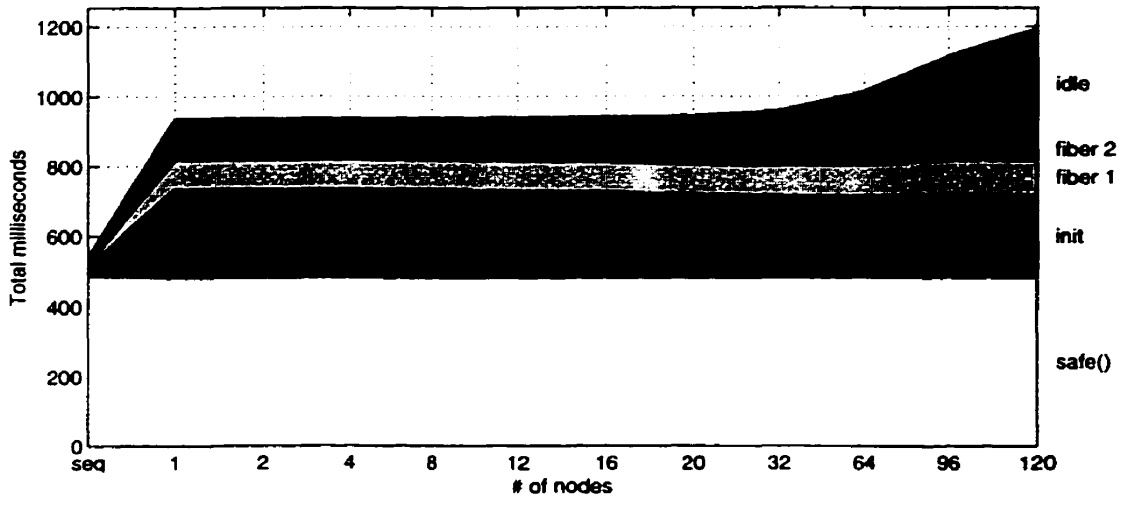


Figure 7.26: Breakdown of EU Use on EARTH-MANNA-S for N-Queens-P (10)

least some of the nodes must have three rows instead of two.

The curves in Figure 7.16 show Tomcatv achieving close to its theoretical speedups for the larger problem sizes. The results for the two smaller sizes ($(N = 33 and 65)$) show the benefits of multithreading by demonstrating what happens when there is only one thread. When the number of nodes equals the number of rows, there is only one procedure per node. When a fiber sends a block of data to another fiber and then terminates, there is no work to be done while the block is being moved, and no work can become enabled until the block move is finished. Thus, there is excess idle time, and Tomcatv achieves only about 3/4 of its theoretical relative speedup for these two sizes.

Dual-Processor Nodes Versus Single-Processor Nodes: It was shown previously that EARTH-MANNA-S generally has better USE factors; the overhead of communicating between processors outweighs the costs of performing the operations on one processor. Comparing Figures 7.17–7.18 with Figures 7.12–7.13 show that for the fine-grain applications, the single-processor implementation gives better speedups for larger node counts. Once again, this may be due to better performance of the load balancer, and tuning of the balancer on each implementation might change this advantage. However, the trend seen in Section 7.1.4, in which

programs with longer threads suffer on EARTH-MANNA-S due to long network response delays, is seen even more clearly as the node count gets large. In fact, in this case the fully-parallel version of N-Queens outperforms the throttled version! As in the earlier experiments, adding a watchdog timer improves things considerably, as seen in Figures 7.22–7.24.

7.2.3 Performance of Updated EARTH-MANNA Systems

The final experiments performed with simulated off-the-shelf hardware involved a simulated "modernized" i860-based multiprocessor. USE factors and speedup curves were obtained for the 13 benchmark/platform combinations, using a simulated system with processor and memory parameters improved to levels corresponding to the PowerPC 620-based PowerMANNA, as described at the end of Section 7.2.1. While the improvements in speed varied from application to application, the relative performance of the various implementations, and the speedup curves, were not significantly different from the results in the previous section, except for a few anomalies. Thus, our main conclusions from the previous sections still hold true, and the EARTH multithreading model will benefit modern processors just as well as the i860. The data from these experiments are therefore placed in Appendix D.

7.3 EARTH on Other Multiprocessors

The MANNA was the first off-the-shelf platform on which EARTH was implemented. The first working version of EARTH-MANNA-D, including the runtime system described at the beginning of the chapter and the compiler outlined in Section 7.3, was completed in four months by a single designer. Since then, the EARTH team has successfully implemented EARTH on other off-the-shelf platforms. Although these systems are not the focus of this dissertation, they are discussed in this section to demonstrate the generality of the EARTH model and to draw conclusions from the results obtained.

7.3.1 EARTH-SP-2

The first non-MANNA platform running EARTH was the IBM SP-2 multiprocessor. This platform is a little more "off-the-shelf" than the MANNA, which was a research

machine specifically intended for multithreading and whose production was limited. The SP-2 is widely used in large research laboratories and is thus a good benchmark for testing the portability of the EARTH PXM. Furthermore, IBM generously assisted us by providing proprietary low-level information on the communication layer in their system, allowing us to make interprocessor communication far more efficient.

There are two major differences between the SP-2 implementation and the original MANNA version. First, changes to Threaded-C were mandatory. In EARTH-MANNA it is possible to reference the entire global address space with 32 bits (the largest MANNA implemented having only 1.28 Gbytes). Therefore, the original implementation of Threaded-C on EARTH-MANNA stored global addresses in regular pointers, and there was no GLOBAL keyword. However, a larger SP-2 machine has more memory than will fit into the address space of a single processor (4Gbytes), so regular pointers won't be big enough to accommodate global addresses.

Therefore, explicit support for global addresses had to be added to Threaded-C. The new type specifies (internally) both a processor-node number and an address within that node. It is necessary to distinguish between global and local addresses explicitly, for the EU may only access *local* addresses; the EU must send global addresses to the SU for processing. This distinction was achieved by introducing a GLOBAL type qualifier for pointers and by providing explicit conversion operations between local and global pointers. (Support for GLOBAL handles has been added to EARTH-MANNA, but this is only for code compatibility; the EARTH-MANNA compilers ignore the keywords and use the original MMU-based translation scheme.)

The other change was to the methods used for compilation. Although the EARTH-MANNA compilers have a processor-specific C compiler at their core, the pre- and post-processors in Section 7.3 are finely tuned both to this compiler and the i860 instruction set. To reduce further the development time of the implementation on a given platform, it was decided to make the Threaded-C compiler more generic by expanding the source transformation of the pre-processor and eliminating the post-processor entirely. Any processor-specific post-compilation tuning of the object code must be done at runtime by the runtime system, which is still tuned to the specific platform.

7.3.2 EARTH-Beowulf and Other Networks of Workstations

The SP-2 is built of tightly-coupled off-the-shelf uniprocessors. An even more generic approach to building inexpensive multiprocessors is the *Network Of Workstations* (NOW) approach, exemplified by the Beowulf system [15]. Beowulf-class systems are formed quickly and easily by connecting ordinary microcomputers (e.g., Pentium-based PCs) together with fast Ethernet (100BASE-T) switches. These systems typically run a standard operating system such as Linux.

Because the Beowulf is gaining in popularity as a low-cost alternative to tightly-coupled multiprocessors, EARTH was ported to this system to make it available for experimental use. (The Beowulf implementation was first publically demonstrated at CalTech in January, 1998, with 60 nodes running the Povray application.) The compiler is the same as the portable compiler used for the SP-2. Like the SP-2 implementation, EARTH-Beowulf uses inlined code for EARTH operations as in EARTH-MANNA-S. The user code and the runtime system (i.e., the EU and SU) run as separate OS processes, but the EU runs as a single process, and all switching between fibers is the responsibility of the runtime system. Finally, the internode communication uses TCP/IP, since the low-level communication primitives of the SP-2 are unavailable here.

The Beowulf system is the most generic version of EARTH available, though this portability comes at the price of performance. It was recently ported to a PowerMANNA (the PowerPC 620-based successor to the MANNA), requiring only a few hours of effort. (This system was first demonstrated, again using Povray, at Supercomputing '98.) The TCP/IP layer on the PowerMANNA uses the MANNA link chips and crossbars, rather than a standard Ethernet system, and is thus much faster, but it still incurs the OS overheads associated with TCP/IP. (There are plans to make a more finely-tuned implementation specifically for the PowerMANNA, similar to the MANNA implementations.)

7.3.3 Clusters of SMP Workstations

Tightly-coupled multiprocessors and networks of single-processor workstations represent the endpoints on a line representing the distribution of processors and memory. Some researchers have suggested the benefits of systems between these two

extremes, in which small multiprocessors with local shared memory are interconnected, but two different clusters either do not share memory or have much higher latencies for shared-memory access than two processors in the same cluster. The EARTH Architecture Model already accounts for this possibility; Section 4.2.1 allows the EU to have multiple processors.

EARTH has been ported to a network of 16 workstations at the University of Delaware [80]. Each machine has 4 250MHz UltraSPARC-II processors, shared memory connected to the processors through a local crossbar, and a high-speed Myrinet network interface. The implementation of EARTH is based on the generic EARTH-Beowulf system with TCP/IP, but the runtime system has been modified to handle multiple EU processors. Each CPU is a separate single-processor node, but the load balancer favors local processors when distributing tokens, on the assumption that the invoked procedure is likely to access data stored on the originating processor. The Delaware team is currently investigating an implementation more like EARTH-MANNA-D, with separate EU and SU, except that the EU consists of three of the four processors.

7.3.4 Observations

This chapter has presented several implementations of the EARTH EVM-A on off-the-shelf multiprocessors. The main focus of this dissertation is the MANNA-based implementations, and a repeating of the experiments on all the additional platforms discussed in this section would lead to an overwhelming amount of data to present. Rather than present all the data, this chapter will conclude with a general empirical comparison of the systems and some conclusions which can be drawn.

One important lesson is the importance of removing as many overheads as possible from the synchronization and communication operations. Unfortunately, these are the areas in which traditional multiprocessing systems often place the *most* overheads. Of all the EARTH implementations, the MANNA platform is the fastest, if clock speeds are normalized. All essential multithreading operations (thread generation, thread synchronization, and passing of data between threads) are performed at the user level, eliminating the overheads of interacting with the operating system. The systems discussed in this section all require OS intervention to communicate

between processors, and performance varies inversely with the amount of intervention needed. The generic system, which uses separate processes for EU and SU, and uses TCP/IP for communications, does the worst, especially for fine-grained programs, while the SP-2 does much better, thanks to the low-level access that the EARTH RTS has. From this it can be inferred that OS-level threading would be even less appropriate for an EARTH-like system.

Of course, one reason for requiring OS intervention for certain operations is to keep access to certain essential resources under the protection of the kernel, especially in multiuser environments. A dual-processor system such as EARTH-MANNA-D provides an opportunity for such a protection scheme without sacrificing performance. In EARTH-MANNA-D, only the Communication Processor (emulating the SU) needs to access the network. The SU can therefore run as a privileged process while the EU only runs user code, interacting with the SU only through the EQ and RQ (which also can have access restricted for greater protection).

Another observation is that it is difficult to develop a single system which works optimally for all applications. The experimental results in Sections 7.1.4 and 7.2.2 show that some implementations work better for fine-grain parallelism, but worse for coarse-grain programs. Since different levels of granularity are appropriate for different applications, it is important to find a system that works reasonably well for most applications rather than working best for only a few.

Our experiences with "generic" implementations of EARTH that rely entirely on native compilers for code generation show that portable versions of EARTH are possible, and can be brought up quickly on new multiprocessors so long as they support the standards on which the portable implementations are based. This is much more in line with the idea of EARTH as a programming model rather than an architecture. However, such generality comes at the price of performance. For best performance, both the runtime system and the compiler need to be tuned to the specific characteristics of the architecture. Higher-level abstractions such as EVM-A and Threaded-C are sufficient to ensure portability of *applications* across platforms, provided they are well-planned. Portable systems, however, may be ideal for one-of-a-kind systems where the effort of a fine-tuned implementation cannot be supported.

Our final observation is that parallel computers based entirely on off-the-shelf processors can go only so far without specialized hardware support. Such parallel

machines offer a quick and low-cost means to get started in general-purpose parallel programming, but they have limitations. While all implementations of EARTH were able to exploit modest amounts of parallelism in all applications, none were able to extend this to a larger number of procesors (a hundred or more) on enough applications to be considered a general solution to the problem of attaining such levels of parallelism.

Therefore, it is time to start moving along the evolutionary path toward machines specifically built for multithreading and parallelism. The next chapter takes up the issue of specialized hardware support for machines still primarily based on commodity processors. Experiments there show that such hardware goes a long way toward addressing the performance concerns raised here.

Chapter 8

Toward a Custom EARTH Implementation

The previous chapter demonstrated that a viable EARTH system could be emulated on an off-the-shelf multiprocessor and achieve good absolute and relative speedups on many applications.

Nevertheless, our experimentations with the EARTH-MANNA system, the most efficient of the EARTH implementations so far, showed that there are limitations to this approach. The system works quite well when threads are coarse and synchronizations are not too frequent, for the overheads of fiber switching and interacting with the Event and Ready Queues take a small fraction of the total computation time. But as threads become smaller and more numerous, these overheads start to dominate and drag down total performance.

One result of this is that programmers have to spend extra time thinking about how to coarsen their programs without sacrificing parallelism (or rely on compilers, such as the EARTH-C translator [52], to do this for them). When members of the EARTH team were writing applications in Threaded-C (some of which are used in this study), some found that a large part of their effort was spent in such coarsening. For divide-and-conquer programs this may involve simply writing both a parallel and a sequential version of the same routine, and switching from parallel recursion to sequential execution after descending enough levels in the call tree (as in the N-queens program in our study). In other cases, coarsening might be more complicated, requiring algorithmic changes to achieve the same results with fewer threads, or figuring out ways to combine finer threads into larger units to amortize

synchronization and communication costs.

In either case, this effort goes against our stated goal of programmability. In our comparison of hypothetical architectures in Section 1.2.4 (see Figure 1.1), we said that architecture A was the ideal vehicle for quick implementations of parallel programs with acceptable performance. Yet for some applications, our EARTH-MANNA programmers followed a curve more like B due to the extra effort of coarsening. Furthermore, such efforts are even more difficult to program into a high-level translator that is targeting Threaded-C code.

Reducing the number of threads to improve speedup for a small number of nodes may end up harming the program's scalability on larger machines. Coarsening a program reduces the supply of tokens available for load balancing, while enlarging the machine increases the need for tokens. Furthermore, as the number of tokens per node decreases, the variability of token execution times becomes a bigger factor in the success of the load balancer, as there will be fewer tokens to "average out" the running times. The programmer may compensate for these problems, by making the number of tokens generated proportional to the node count (NUM_NODES in Threaded-C, for example), and by trying to make sure the threads have roughly equal running times. But these added burdens merely exacerbate the programmability problem.

Our goal, therefore, is to lessen the need for programming effort (and translator/compiler complexity) by making the machine better able to support fine-grain parallelism. The architectural model for EARTH describes the SU as a separate unit with specialized functions. The first section in this chapter demonstrates that a module specialized to the SU's tasks can support smaller grain sizes effectively, leading to improvements in both absolute and relative speedups. The second section looks at further benefits in combining and partially integrating the EU and SU cores, which is the next step in the evolutionary path. The final section considers some of the ramifications these findings may have for future high-end processor design.

8.1 An External Synchronization Unit

Implementing the SU functions in a custom unit would produce a machine with many advantages over the multiprocessor platforms in the previous chapter. Overall, a specialized SU would be able to perform EARTH operations much faster than

a general-purpose processor executing RTS code. A node based on a hardware SU would retain the advantages of a dual-processor system, such as removing the burden of network interaction from the main processor, while eliminating the speed disadvantages that made some of the benchmarks run slower on the dual-CPU nodes than on the single-CPU nodes.

First, communication between the EU and SU would be faster. Typical processors are not designed to act as simple slave devices on a bus. It is difficult for one processor to communicate with another without incurring a lot of overhead. The software queues between the AP and CP in EARTH-MANNA are about the best that can be done with such a system.

A hardware SU would perform its specialized tasks much faster than a general-purpose processor executing RTS code. Even a simple operation such as a synchronization operation involves fetching a sync count, decrementing, testing for zero and writing back, which can take more than a dozen instructions in the i860. Logic built for this purpose, with sync slots stored in a special cache, should be able to do this in a few cycles at most. This would decrease the latency of EARTH operations, which would reduce idle time in cases where these operations are on a critical path.

Furthermore, a hardware SU could have several semi-independent modules operating in parallel, since it isn't constrained by the semantics of a single sequential stream. For instance, one part could be receiving messages from the EU, another part polling the link chip, and a third module making load balancing decisions, all at the same time. This would further decrease the latency of EARTH operations, for interactions on one interface would not be stalled while the CPU is working on the other interface.

Both the faster execution of SU functions and their execution in parallel would increase the throughput of the SU, making it better able to handle higher workloads. Preliminary experiments with an EARTH implementation on 4-CPU workstations suggest that, for fine-grain applications, a single CPU emulating an SU can become overloaded if it tries to support the other three CPUs at the same time. A hardware SU would be better able to handle a multiple-PE Execution Unit.

The faster and parallel execution of SU functions would allow greater complexity in the tasks performed. The operations on an emulated SU must be kept simple to avoid overloading the CPU. Thus, some components are implemented in the most straightforward way. The Ready Queue, for instance, is a simple FIFO queue in the

off-the-shelf EARTH emulators. However, the EARTH model allows events in the RQ to be reordered, and this could be done to optimize for locality, for instance. Load balancing is another area for improvement. The load balancer on EARTH-MANNA has to avoid using too much network bandwidth in distributing workload information, since traffic between the SU and link competes with the EU for bus access. Furthermore, the RTS can't spend much time processing token requests or making an optimal decision where to send tokens. Consequently, the RTS load balancer often makes suboptimal choices, and some nodes take a long time to receive any work. Our speed-up curves in Section 7.2 often show performance leveling off as the number of nodes increases, mostly reflecting the worsening effectiveness of the load balancer.

Finally, if produced in sufficient volume, a hardware SU would be much cheaper than a state-of-the-art microprocessor of comparable performance. Major sections of the typical microprocessor are irrelevant to the SU and could be eliminated, such as the floating point logic and instruction cache. Other components of the modern superscalar processor, such as the reorder buffer and register renaming logic, would not be needed by a hardware SU because it would not be trying to extract maximum parallelism from a piece of sequential code.

This section proposes an SU interface and top-level design, and performs SEMi simulations to measure the benefits of this design compared to the multiprocessor versions. The SU proposed here is designed specifically for the MANNA system. However, with minor modifications it should be equally suitable for a node based on another processor, or can be made more portable by having an interface for a common bus standard such as PCI.

8.1.1 An SU-Based EARTH Node

The EARTH architecture in Figure 4.11 shows separate modules for the EU, SU, and queues (EQ and RQ), and separate paths to memory. However, a MANNA-based SU would need to read and write program data on the same bus as the EU, because the i860XP's cache coherence is based on bus snooping. However, if the SU hardware and memory system can support dual ports, a separate bus can be used for SU-specific traffic, as shown in Figure 8.1. This traffic would include all link traffic (since the EU no longer needs to access the link) and any memory which is

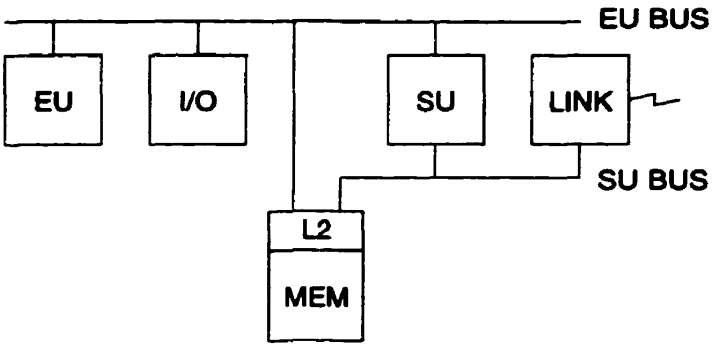


Figure 8.1: Node with Hardware SU (Separate Bus)

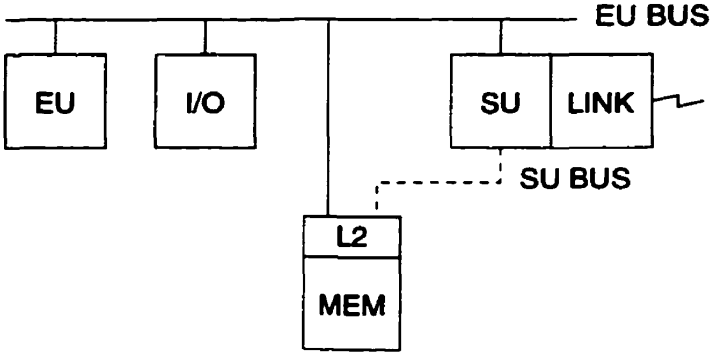


Figure 8.2: Node with Hardware SU (Integrated Link)

specific to the SU (since coherence with the EU's cache is not needed). The latter includes sync slots (which are not be manipulated directly by the EU) and any extra buffering needed because it doesn't fit in the SU chip.

An alternative design is to integrate the link and SU into a single chip. A major part of the link chip's logic is occupied by a general interface to the MANNA's 64-bit data bus and 32-bit address bus, including special control and status registers used by the processors. This logic is no longer needed if a regular processor is not using the link. It should be possible to integrate the link's FIFO queues and crossbar interface into the SU, especially since the SU in Figure 8.1 requires its own driver for the general-purpose link chip, and this driver could be eliminated as part of the integration. The alternative configuration is shown in Figure 8.2. Here the SU bus is shown merely as an option because, given a sufficiently large cache in the SU, there should be so little need for accessing the main memory from the SU that using

the EU bus should not hurt performance significantly.

In these configurations, the EQ and RQ are included in the SU. Although there are off-the-shelf FIFO units that could be used for the queues, they are not expected to be very large, and integrating them into the SU allows random access to the queues by the SU, making possible the reordering of ready fibers, as previously described. Furthermore, direct contact between the EU and SU allows for auxiliary information to be passed from the EU to the SU through the address lines, as explained in the next section.

8.1.2 The SU Interface

How do the EU and SU communicate in a node such as Figure 8.2? The link chip in the MANNA currently allows data written to the link's FIFO to be tagged by writing to different addresses in the link chip's address space. This idea can be extended to create different addresses for each of the EARTH operations. It would not be necessary to send an operator tag on the data bus, meaning that EARTH operations could be sent over the bus more quickly.

Use of tagged addresses can be extended even further by using some of the address bits to send small pieces of data. There are not many EARTH operations, so they should require only a small number of bits. On the other hand, many EARTH operations have operands which in most cases are very small integers. Local synchronizing operators typically select a small sync slot or fiber number (less than 10 in all our benchmarks). Block moves of structs and arrays of fixed size often have small byte counts which are compile-time constants, and procedure invocations have short argument lists of fixed length.

In this implementation, the address space of the SU covers a region of the memory address space reachable using a single immediate offset from a base address. The i860 has 16-bit immediate offsets which may be used in load and store instructions. The EU has a base address which is permanently stored in a register set aside for addressing the SU (or which can be regenerated in a single instruction). From this base, a range of 2^{16} addresses can be specified in a single memory access using a constant offset.

This 16 bits can be divided into a basic tag field and a number field for passing short integers, e.g., small sync slot numbers. There can be two variants of each

operation, a "compressed form" using the number field and a "long" form using the data bus to pass this information. The long form is used if the value that would go in the number field (e.g., the sync slot number) is too large, or unknown at compile time.

Most EARTH operations have more than one operand (particularly when the long form is used) and consequently will require writing to the SU more than once. Thus, each EARTH operation requires a particular sequence of reads or writes. The first access in a sequence identifies the operation; from there the SU knows the identity and purpose of each subsequent field read or written so that the transfers can occur without any additional tagging. Some operations may allow tagging of some additional fields to permit more options, and may allow additional address fields to be used to transmit small integers.

With synchronizing operations that specify local slot or fiber numbers, such as SYNC or SPAWN, the slot and fiber numbers are only meaningful in the local frame context. Therefore, the SU must know the current frame when it receives these events from the EU. The SU keeps a register with the current frame identifier so that it can produce the proper sync slot locations and instruction pointers. This register is updated whenever a new fiber is removed from the RQ.

Remote sync and spawn always require using the data bus, because they must pass global addresses to the SU (sync slots or instruction pointers). Remote spawn passes both an IP and an FID to the SU. They may be passed one at a time or together in a single double-word store (the system bus includes bits to indicate the data size, which can be seen by the SU). The latter may be useful if register pairs are used to hold (FID,ID) pairs, since register pairs may be used in double-word store instructions.

Most of these operators are executed by writing to addresses in the SU address space. Two operators, however, use load instructions rather than stores. The NEXT_FIBER operation reads an entry from the RQ and returns the (fid, ip) pair. Either the two addresses can be read simultaneously (as a double-word), or the IP is read first, in which case the next load reads the frame address. The operation also copies the pair read from the RQ into the SU's context register, so that slot and fiber numbers from the new fiber are matched to the right frame. The END_PROCEDURE operation also fetches the next fiber from the RQ. It also tells the SU to terminate the current procedure and deallocate the current frame.

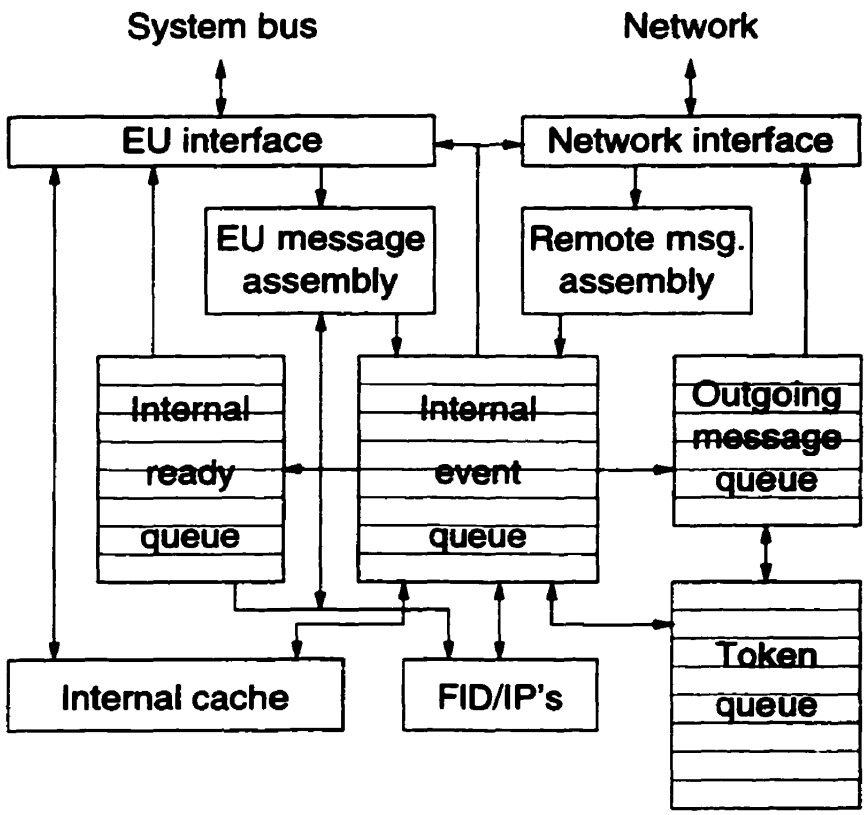


Figure 8.3: Synchronization Unit Block Diagram

Finally, some addresses are reserved for configuration. The simulated SU in the next section doesn't need this, for it has been tuned to the MANNA and i860. However, the SU can be made more portable if the system can configure it by writing to certain memory-mapped registers to set system parameters. For instance, if the SU is used with a Sparc-based system, the data fields should be reduced by three bits because Sparc processors only have 13-bit immediate offsets in load/store instructions.

8.1.3 SU Design and Simulation

While an actual hardware design is beyond the scope of this dissertation, a basic top-level design was done in order to guide the construction of the simulator. A block diagram is shown in Figure 8.3.

The SU has the following storage areas:

- The core of the SU is the **internal Event Queue**, which is a pool of uncompleted events waiting to be finished or forwarded to another node. There may be times when many events are generated at the same time, which will fill the queue faster than the SU can process them. For practical reasons, the SU can work on only a small number of events simultaneously. The others wait in a substantial overflow section.

- The **internal Ready Queue** holds the list of enabled (fid, ip) pairs.

- An **outgoing message queue** buffers messages that are waiting to go out over the network.

- The **Token Queue** holds all tokens (generated by the TOKEN instruction) on this node that have not yet been assigned to a node.

- The **FID/IP** section stores the frame identifier and instruction pointer for each PE.

- An **internal cache** holds recently-accessed sync slots and data read by the SU (e.g., during data transfers).

These storage units are controlled by the following logic blocks:

- The **EU interface** handles loads and stores coming from the system bus. For a load, the EU interface either reads an entry from the Internal RQ and puts it on the data bus, or puts the FID left over from a previous load on the data bus. In the former case, it identifies the PE from the address and updates the corresponding entry in the FID/IP table. Writes are forwarded to the EU message assembly area.

- The **EU message assembly** area collects sequences of stores from the EU and converts slot and fiber numbers to actual addresses. Completed events are put into the EQ.

- The **Network interface** drives the link chip (if the link is external) or the link interface (if the link is internal). Outgoing messages are taken from the outgoing message queue. Incoming messages are forwarded to the remote message assembly area.

- The **Remote message assembly** area is like the EU message assembly area. It injects completed events into the EQ.
- The **Internal Event Queue** has logic for processing all the events in the EQ. It accesses all the other storage areas of the SU.

To compare the performance of an EARTH computer with hardware SUs to an off-the-shelf multiprocessor emulation of EARTH without actually building an SU requires an accurate simulator. The SEMi simulator, described in Section 7.2, was primarily built for this purpose. After SEMi was able to simulate the base MANNA machine successfully at an acceptable level of accuracy, a module simulating the behavior of a hardware SU was added. This module (2,400 lines of C code, including statistics-gathering and error-checking) was modeled after the design in Figure 8.3.

In keeping with our goal of realistic simulations, the parameters of the module are based, whenever possible, on hardware already existing on the MANNA. The speed of loads and stores to the SU, for instance, are the same as loads and stores to the MANNA's link chip. Internal operations in the SU are programmed with conservative timing assumptions.

The final step needed in order to make realistic experiments is to modify the Threaded-C compiler so that the executable accesses the SU's memory-mapped addresses rather than the software queues. This required small changes to the Threaded-C pre-processor (see Figure 7.3), and slightly more substantial modifications to the Threaded-C post-processor. However, the changes amounted to less than 10% of the source code lines in both modules, for the basic conversion techniques are the same.

8.1.4 Experimental Results

The beginning of Section 8.1 gave a list of reasons why an EARTH computer based on a custom SU should perform much better than a system based on a stock processor emulating SU functions. To validate the performance claims, the experiments from Sections 7.2.2 and 7.2.3 were repeated using the hardware-SU incarnation of SEMi, described in the previous subsection. The experimental results demonstrate the following benefits of a hardware SU:

Faster operations: The basic EARTH operations are faster, many considerably so, both in the dispatching of operations from the EU to the SU and in the

processing of these requests within the SU itself. This decreases message latency substantially and brings bandwidth close to the network maximum (Section 7.1.4.1).

Single-node gains: Single node performance, as measured by USE factors (see Section 7.1.4) increases for all benchmarks. The improvement is most dramatic for benchmarks with poorer USE factors in the EARTH-MANNA system (Section 8.1.4.2).

Improved scalability: Speedups (even relative speedups) are higher for all benchmarks. For most benchmarks, the "knee" of the curve occurs at a much higher number of nodes for the hardware-SU system than for EARTH-MANNA (Section 8.1.4.3).

Simpler programmability: Speedups were respectable, if not optimal, for "naive" programs in which no programmer effort was made to control parallelism. This suggests that less programmer effort is required to achieve a given level of performance on a hardware-SU system than on a software-emulation platform (Section 8.1.4.3).

8.1.4.1 Performance of Individual Operations

First, the performance of individual operations was measured using the same procedures as in Section 7.1.4.1. Table 8.1 shows the results of the pure latency and bandwidth tests (see Figure 7.4) applied to an SU-enhanced system, with the results from Table 7.1 included for comparison. The hardware SU shows a dramatic drop in raw latency (42% lower than the faster of the two off-the-shelf systems), making EARTH-MANNA-SU faster than all the commercial systems surveyed [28]. This is due to an across-the-board drop in delays for all parts of the synchronization operation except the network delay itself, which is not affected by the node architecture. (Indeed, our analysis shows that almost a quarter of the remaining latency is caused by the network.)

The bandwidth achieved is slightly higher than EARTH-MANNA-D, though the latter already does a good job of using available bandwidth. Most of the improvement is due to smaller packet headers in the block move. The hardware SU only needs a few bits to identify the type of the block, because it can use an internal

Parameter	SU	Dual-processor	Single-processor
Latency (ns)	**1414**	4091	2450
Latency (cycles)	**70.7**	204.5	122.5
Bandwidth (MB/s)	**44.4**	42.0	28.8
Bandwidth (% of peak)	**88.8**	83.9	57.5

Table 8.1: Latency and Bandwidth on EARTH-MANNA-SU vs. EARTH-MANNA-D/S

Operation	**Nodes with hardware SU**				Single-processor nodes			
	Sequential		**Pipelined**		Sequential		Pipelined	
	Loc.	**Rem.**	**Loc.**	**Rem.**	Loc.	Rem.	Loc.	Rem.
(r)sync	**480**	**1130**	**120**	**340**	1000	2290	380	668
(r)spawn	**480**	**1350**	—	—	920	2500	—	—
get_sync	**1060**	**2280**	**480**	**440**	1440	4666	700	1502
data_(r)sync	**600**	**2160**	**360**	**380**	1280	4340	560	1200
invoke (1 arg)	**1740**	**2640**	**1125**	**735**	2300	5360	1611	2165
invoke (5 args)	**2220**	**3660**	**1506**	**997**	2460	5640	1768	2231
invoke (9 args)	**2660**	**4361**	**1868**	**1105**	3060	6500	2368	3165
invoke (18 args)	**3020**	**5400**	**2148**	**1658**	3220	7620	2528	3537

Table 8.2: EARTH Operation Latencies (nsec.) on EARTH-MANNA-SU vs. EARTH-MANNA-S

hardware lookup table to dispatch the incoming block. The software emulators in EARTH-MANNA-D/S send the 32-bit address of the handler for the block move because that is more efficient than looking up the tag in software.

The experiment measuring the latencies of individual EARTH operations, in both sequential and pipelined modes, was repeated for the SU. The results are in Table 8.2, with the results for single-processor nodes included for comparison. (EARTH-MANNA-S is faster than EARTH-MANNA-D in all cases.) The table shows that all EARTH operations are faster on the hardware SU than on the emulated systems, 2 to 3 times faster in many cases. Especially important to note is the improvement in the pipelined cases, which gives an indication of the performance benefits which can be realized if the code effectively overlaps communication and computation.

Operation	SU	Dual
1. Local EU sends GET_SYNC to EQ	120	780
2. Local SU reads event from EQ	80	500
3. Local SU determines location of source, sends request there	40	500
4. Network transfers message	600	600
5. Remote SU reads message from network (+ polling delay)	100	610
6. Remote SU reads value from source reference in its memory	40	300
7. Remote SU sends message to network	60	500
8. Network transfers message	600	600
9. Local SU reads message from network (+ polling delay)	100	610
10. Local SU writes value to destination reference in its memory	80	200
11. Local SU decrements local sync count, places fiber in RQ	40	600
12. Local EU reads fiber from RQ	420	720
Total	2280	6520

Table 8.3: Component Latencies (nsec.) for GET_SYNC on EARTH-MANNA-SU and EARTH-MANNA-D

The sources of the improvements can be seen by analyzing, using SEMi, the number of cycles taken in each step of an EARTH operation. Table 8.3 breaks down the time taken by each phase in a remote GET_SYNC operation on EARTH-MANNA-SU and EARTH-MANNA-D. The twelve phases listed are the twelve phases shown in Figure 4.12, which illustrate GET_SYNC in an architecture with separate EU and SU.[1]

The final test of individual EARTH operations in Section 7.1.4.1 measured the time taken by the EU to execute specific operations (either by dispatching them to the SU, or performing the operations within the EU itself in single-processor nodes). Table 8.4 shows the data from Table 7.3 along with the times achieved by the hardware-SU simulation. The table lists only the time needed by the EU to *initiate* the operation on EARTH-MANNA-SU and EARTH-MANNA-D, not to *finish* them. They are therefore relevant only if there is other work which can be done while the SU is completing the operation, e.g., the operation is executed in the middle of a fiber or there are other fibers waiting in the RQ. If this is not the case, and the EU stalls until the operation is completed, then the costs will more resemble the numbers in Table 8.2.

[1]The slight discrepancy between the total time for EARTH-MANNA-D and the sequential latency for GET_SYNC reported in Table 7.2 reflects the inaccuracy of the simulator.

Operation	Hardware-SU		Dual-CPU		Single-CPU	
	Local	Remote	Local	Remote	Local	Remote
(r)sync	40	60	504	504	300	588
(r)spawn	40	100	721	580	323	640
end_fiber	282	—	530	—	441	—
incr_(r)sync	60	60	561	554	300	620
data_(r)sync	80	100	580	606	480	660
get_sync	100	180	580	620	620	700
invoke (1 arg)	198	200	760	620	479	806
end_procedure (1 arg)	287	—	794	—	760	—
invoke (5 args)	557	400	1039	907	599	936
end_procedure (5 args)	303	—	1203	—	800	—
invoke (9 args)	677	640	1223	1210	960	1406
end_procedure (9 args)	312	—	1372	—	1040	—
invoke (18 args)	877	820	1766	1512	1099	1670
end_procedure (18 args)	312	—	1728	—	1060	—

Table 8.4: EU Costs (nsec.) of EARTH Operation on EARTH-MANNA-SU vs. EARTH-MANNA-D/S

Once again, the hardware SU is significantly faster for all operations. The improvement is greatest for the smaller operations such as SYNC and DATA_SYNC. This is important, because the EARTH PXM was specifically designed to support a style of multithreading in which small threads (fibers in our lexicon) synchronize and exchange data with each other frequently. The results in Table 8.4 show that these types of operations can be made to operate very quickly on an off-the-shelf processor with only a small hardware extension. This means that this programming paradigm can be supported efficiently, encouraging its use.

8.1.4.2 Single-Node Performance

The superior performance of the hardware-SU system for individual operations naturally suggests that complete programs will run faster too. The benchmarks used in Section 7.1.4.3 were therefore run on EARTH-MANNA-SU. The results for a single-node machine are shown in Table 8.5, with the numbers from Table 7.5 included for comparison.

As expected, the improvements in latency for individual operations translates into improved performance for all benchmarks. Most significant is the fact that

Benchmark	Input	USE factor (%)		
		Hardware SU	Dual-processor	Single-processor
Fibonacci	15	**25.5**	8.6	15.7
	20	**23.1**	7.7	14.1
	25	**22.9**	7.6	13.9
	30	**22.9**	7.6	13.9
N-Queens-P	8	**65.6**	39.9	51.7
	10	**71.7**	46.8	56.1
	12	**77.0**	53.9	65.6
N-Queens-T	8	**85.9**	68.5	78.5
	10	**97.2**	93.1	95.3
	12	**99.4**	99.1	99.3
Paraffins	18	**98.9**	82.1	97.6
	20	**101.5**	85.4	101.4
	23	**100.5**	84.7	100.6
Tomcatv	33	**96.8**	89.3	92.2
	65	**98.9**	91.4	93.7
	129	**101.0**	93.2	95.6
	257	**101.9**	93.7	96.5

Table 8.5: Uni-Node Support Efficiencies on EARTH-MANNA-SU vs. EARTH-MANNA-D/S

the poor performers on the off-the-shelf machines showed the greatest improvement. Fibonacci, a worst-case scenario for multithreading overheads, runs nearly twice as fast with a hardware SU as on EARTH-MANNA-S. N-Queens-P, in spite of its unthrottled parallelism, achieves better than three-quarters the processor utilization of the sequential version. The other benchmarks show smaller improvements, since they were already doing well on the off-the-shelf machines, but the reduction in multithreading overheads helps these applications as well. Thus, a customized SU is not merely a special tool to help fine-grained applications only.

Furthermore, two benchmarks show USE factors above unity for some cases. The USE factor obtained for Paraffins is similar to the USE factor on EARTH-MANNA-S, which could be a compiling anomaly as suggested in Section 7.1.4.3. The superunity USE factors for the larger Tomcatv runs seem to be partially due to the large block moves; the SU is effectively providing the same benefit as a DMA chip.

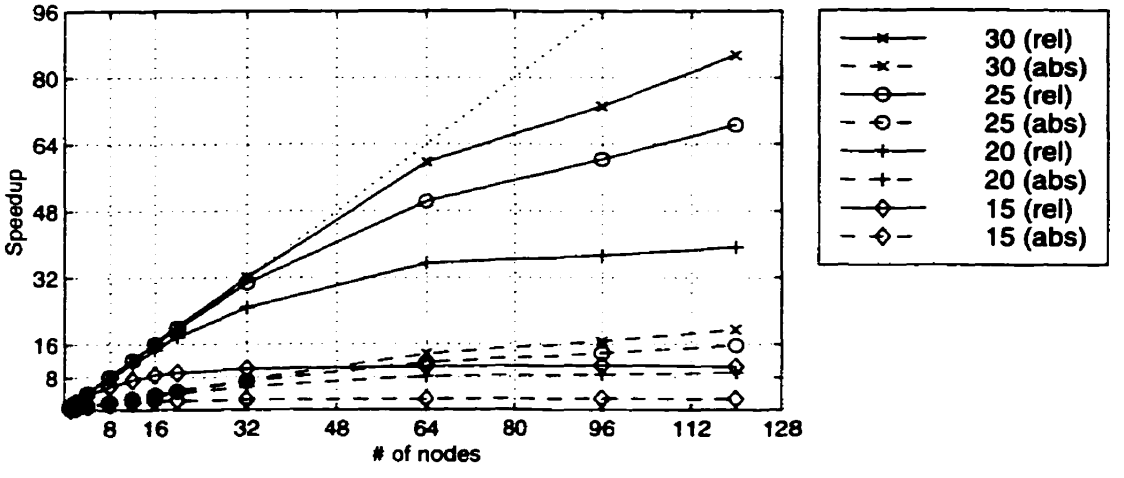


Figure 8.4: Speedups on EARTH-MANNA-SU for Fibonacci

8.1.4.3 Parallel Performance

It is to be expected that the improvement in USE factors provided by the custom SU will automatically boost absolute speedups by a corresponding factor. However, it was argued in the beginning of Section 8.1 that such an SU would improve *relative* speedups as well, because it could interact with the SUs on other nodes more effectively than a software-based SU. Tables 8.2 and 8.3 show that remote fetching is faster on EARTH-MANNA-SU, which can affect computations on critical paths (as can be seen in Sections 7.1.4.4 and 7.2.2,by comparing the performance of machines with single-processor nodes with and without polling watchdogs). Section 8.1 claimed that load balancing would also be more efficient, because the network could be polled without interfering with local memory access, allowing more frequent exchange of load information, and because specialized hardware could afford to make better balancing decisions.

Figures 8.4–8.8, which show absolute and relative speedups on MANNA-EARTH-SU for the benchmarks tested in Section 7.2.2, confirm this prediction. (Appendix C shows experimental results in a different format, allowing a direct comparison of different implementations on the same benchmark.) All benchmarks on EARTH-MANNA-SU show considerable improvements in absolute speed over both the dual-processor-node and single-processor-node emulated systems. Almost all benchmarks show better relative speedups as well, the only exceptions being the

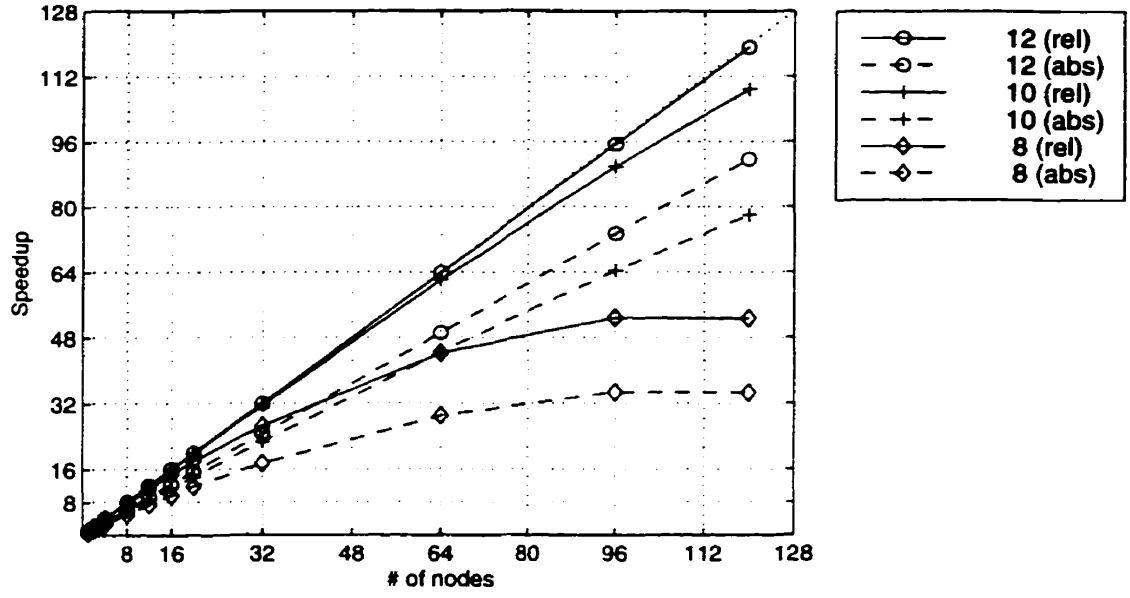


Figure 8.5: Speedups on EARTH-MANNA-SU for N-Queens-P

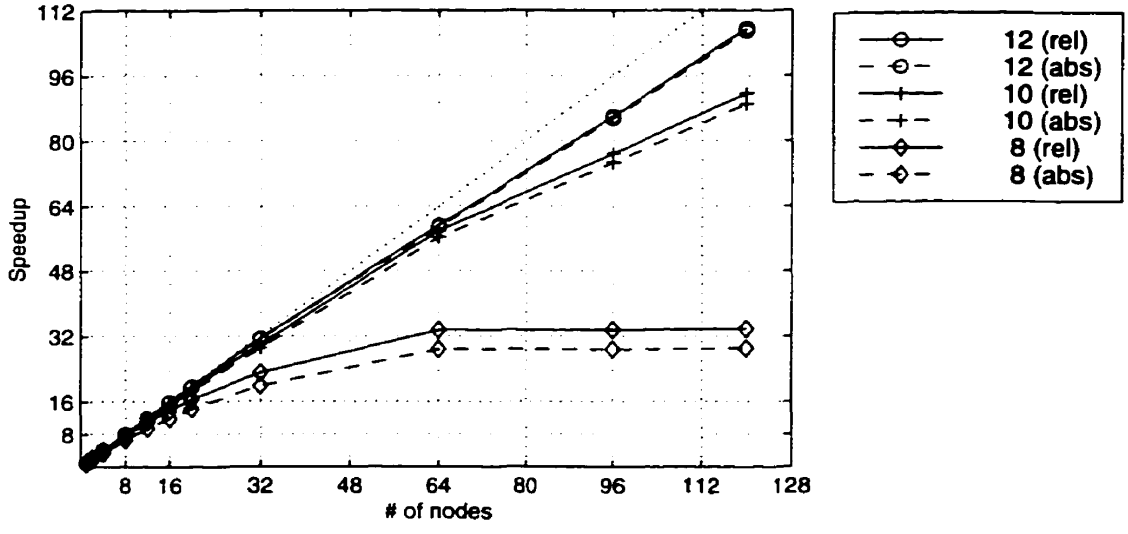


Figure 8.6: Speedups on EARTH-MANNA-SU for N-Queens-T

largest input case of Tomcatv, in which R_{120} is 4% less on EARTH-MANNA-SU than on EARTH-MANNA-D (nevertheless, A_{120} is still higher).

Both factors mentioned above, improved load balancing and faster interaction, have an effect on the improvement of the relative speedup. In most cases, the former

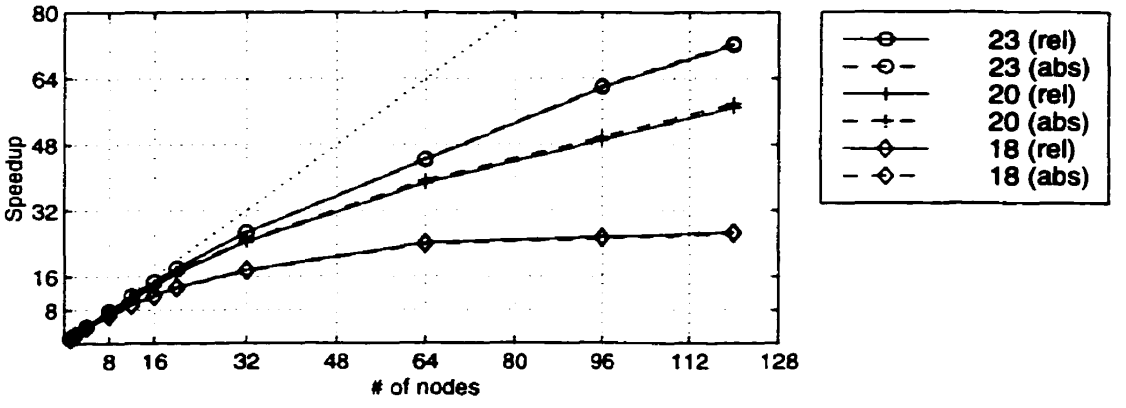


Figure 8.7: Speedups on EARTH-MANNA-SU for Paraffins

is the dominant factor. This is especially true for the first three benchmarks, which have enough parallelism to avoid critical paths.

The effect of poor balancing can be seen in Table 8.6, which shows the evenness of load distributions for the first two benchmarks on intermediate problem sizes running on the largest simulatable configuration (120 nodes). In each of these benchmarks, procedure instances have the same running time, within a factor of 2 or 3, making the number of instances a reasonable approximation of the amount of work assigned to a node. The table lists, for each combination of benchmark and platform, the total number of procedure instances, the minimum load (number of instances on the least-used node), maximum load (number of instances on the most heavily-used node), and the standard deviation of the load distribution. Each of the last three is expressed as a percentage of the average; a minimum and maximum of 100% would indicate the work is perfectly balanced.

Faster responses to remote requests play less of a role in improving relative speedups for these benchmarks, but are still a factor. This is particularly true at the beginning of program execution, when only node 0 is executing the MAIN procedure and all other nodes are idle. Here, the MAIN function is clearly in the critical path, and the faster tokens and invokes can be sent to other nodes, the higher the speedup will be, especially for smaller problem sizes.

The benefits of faster responses can be seen most clearly in the Tomcatv benchmark, which uses static partitioning rather than dynamic load balancing. In Section 7.1.4.4, Tomcatv was shown to lose some performance when each node has only

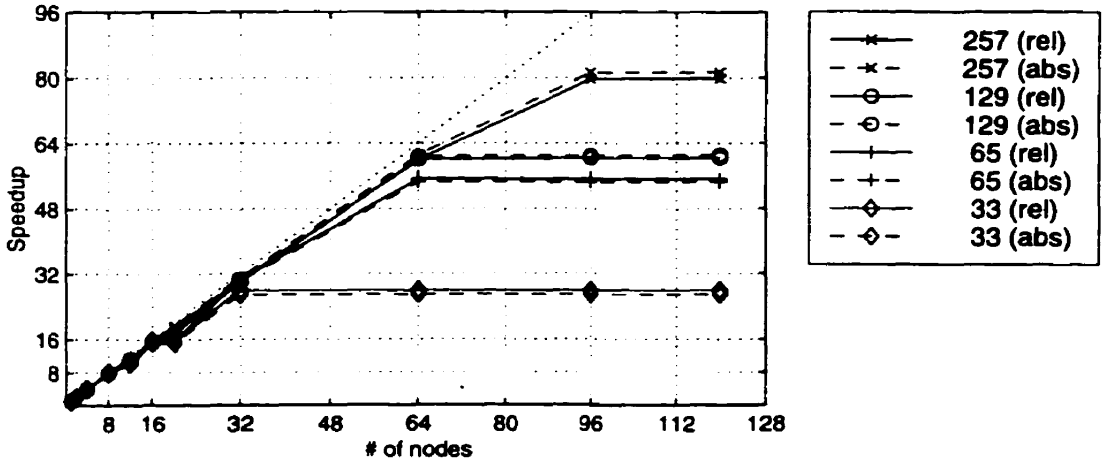


Figure 8.8: Speedups on EARTH-MANNA-SU for Tomcatv

Benchmark	Total proc. instances	Platform	% of average on node		
			Min.	Max.	σ
Fibonacci (25)	242,785	EARTH-MANNA-D	0	704	227
		EARTH-MANNA-S	66.8	182	26.4
		EARTH-MANNA-SU	59.0	171	36.2
N-Queens-P (10)	67,150	EARTH-MANNA-D	0	723	222
		EARTH-MANNA-S	60.9	153	26.9
		EARTH-MANNA-SU	88.8	110	5.17

Table 8.6: Comparison of Load Distributions on 120 Nodes in EARTH-MANNA Implementations

one procedure instance, because computation and communication can no longer be overlapped. This fact is not changed by a custom SU, but speeding up the communication reduces the performance loss. Switching from a processor-emulated SU (EARTH-MANNA-D) to a custom device raises the relative speedup for $N = 33$ from 24.2 to 27.9, and for $N = 65$ from 47.8 to 55.3; each cuts the distance to its theoretically maximum soeedup (31 and 63, respectively) by about half.

These improvements have corollary effects both on scalability and programmability. First, by comparing the speedup curves of different problem sizes for one benchmark, one can see the relationship between problem size and scalability. In all tests in this study, scalability improves as the problem size increases. But this is

true in general for more parallel programs, so it is more interesting to see how well a given program scales with *smaller* sizes.

In each case, the hardware SU produces a curve with a much higher limit of parallelism then the software SU. This holds true for the smaller problem sizes as well, though their limits are lower. In some cases, the software SU's curve peaks at a fairly low level of parallelism while the hardware SU's curve continues growing at the high end of the curve. The Paraffins benchmark (size 20) is one notable example: the software-SU versions (Figures 7.15, 7.20 and 7.23) reach their maximal parallelism before reaching 32 nodes, above which speed actually declines, while the hardware SU continues on an upward slope at 120 nodes (Figure 8.7).

The final observation concerns the impact of the SU on programmability. At the beginning of this chapter, it was argued that reducing the overheads associated with fine-grain parallelism would allow programmers to expose parallelism freely rather than make extra efforts to constraint it, which is sometimes difficult to do without losing performance. While we have not made any systematic study of this issue (the focus here being on maximizing parallelism), the two versions of N-Queens can be viewed as representative of the two alternate approaches. The algorithmically-constrained N-Queens-T clearly outperforms the unconstrained N-Queens-P on the software-emulated platforms. The difference is much smaller when a custom SU is used; in fact, for 8 queens, the fully-parallel version has a higher absolute speedup! This suggests that going to a custom SU will reduce the programmers' need to write coarse-grain code to maximize performance.

8.2 An Internal Synchronization Unit

The preceding section demonstrated positive results for taking the first evolutionary step away from pure off-the-shelf computers. A system containing custom SU hardware performs much better than a system with the same processor that emulates the SU functions in software. The guideline in Section 1.3 suggests that the next logical step would be to add the SU logic to an existing EU core. This would be an intermediate stage between an external SU and a complete integration of the SU functions into the EU (e.g., adding new opcodes to the EU's instruction set). It would be a more conservative jump, one that could use an existing processor core design without significant design changes, and use any compiling technology

developed for the external SU version of EARTH.

In this phase, the SU would be placed on the periphery of the core of an off-the-shelf processor. The EU would still trigger the EARTH operations by loading or storing special memory addresses; thus, no changes to the SU's instruction pipeline would be required. There would need to be modifications to the bus interface, since the loads and stores would need to be redirected to the SU hardware.

Similarly, it would be useful if the SU were to share the data cache of the EU, for data could be transferred between the two without invoking cache coherence mechanisms. For instance, if the EU writes to an array and then tells the SU to do a block move of that array, an external SU will likely trigger cache writebacks by the EU's data cache when the SU starts to load the array into its internal buffer. Sharing the cache would avoid the extra cycles needed for these writebacks.

Thus, there would be two primary benefits in taking this step:

1. Operations would be dispatched to the SU even faster, further reducing latencies. More importantly, fiber switching times would decrease, because the EU would be able to read the next fiber address from the RQ without the latency of going off-chip.

2. Bus traffic would be reduced. This is most important for applications that use EARTH operations frequently. Statistics gathered by SEMi show that fine-grain applications often saturate a node's local bus, causing the EU stall *not* because there is no work to do, but because the SU is using the bus.

The benefits of this next evolutionary step can be tested by making *minor* modifications to the SEMi simulator with its external SU module. An option was added to this version of SEMi to enable dispatching of EARTH commands directly to the SU and the sharing of caches between the EU and SU. No changes to the compiler (for the external SU) were needed.

Experimental results demonstrate the benefits of this next evolutionary step. Table 8.7 shows USE factors for the five benchmarks studied, and compares the results obtained from an internal SU to the external-SU results from Table 8.5. While all applications show improvements, except for the largest cases for Paraffins and Tomcatv, the most significant improvement is seen by the fine-grain applications. These programs make the most use of EARTH operations, and hence suffer the most from their overheads. The overheads of the fully-parallel version of N-Queens

Benchmark	Input	USE factor (%)		
		Internal SU	External SU	Net Improvement
Fibonacci	15	56.8	25.5	+31.3
	20	54.7	23.1	+31.6
	25	54.6	22.9	+31.7
	30	54.5	22.9	+31.7
N-Queens-P	8	89.7	65.6	+24.1
	10	92.6	71.7	+20.9
	12	94.2	77.0	+17.3
N-Queens-T	8	95.1	85.9	+9.2
	10	98.8	97.2	+1.7
	12	99.6	99.4	+0.1
Paraffins	18	100.2	98.9	+1.3
	20	101.8	101.5	+0.3
	23	100.5	100.5	0.0
Tomcatv	33	98.1	96.8	+1.3
	65	102.5	98.9	+3.6
	129	101.1	101.0	+0.1
	257	101.9	101.9	0.0

Table 8.7: Uni-Node Support Efficiencies with Internal and External SU

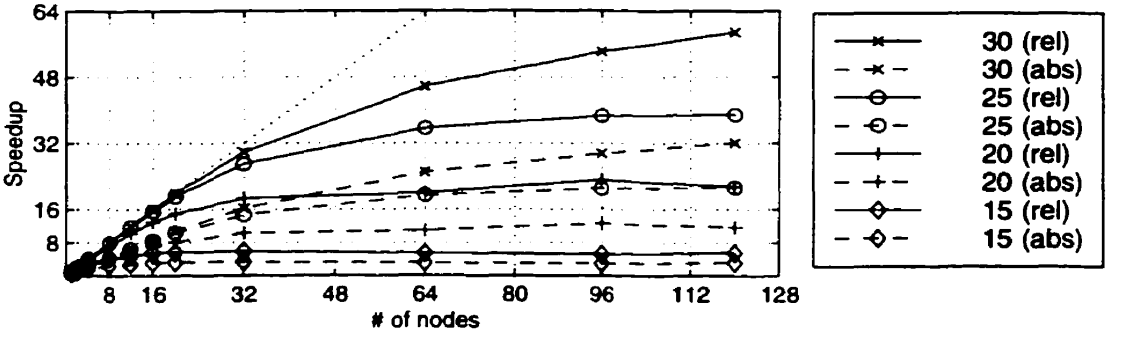


Figure 8.9: Speedups on EARTH-MANNA-SU (Internal) for Fibonacci

drop below 10% for all sizes tested, compared to the sequential recursive code. Even Fibonacci now loses less than half of its performance to multithreading overheads even though these operations make up such a dominant part of its code.

The benefits of fast internal sharing between EU and SU carry through to parallel performance. Figures 8.9–8.13 repeat the experiments of Section 8.1.4.3 with the

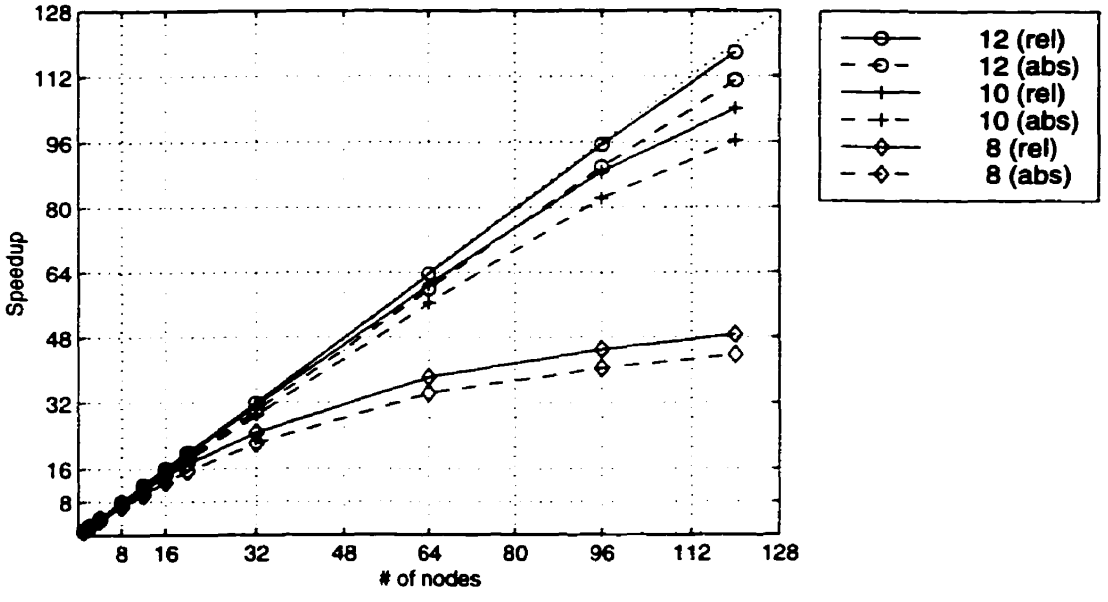


Figure 8.10: Speedups on EARTH-MANNA-SU (Internal) for N-Queens-P

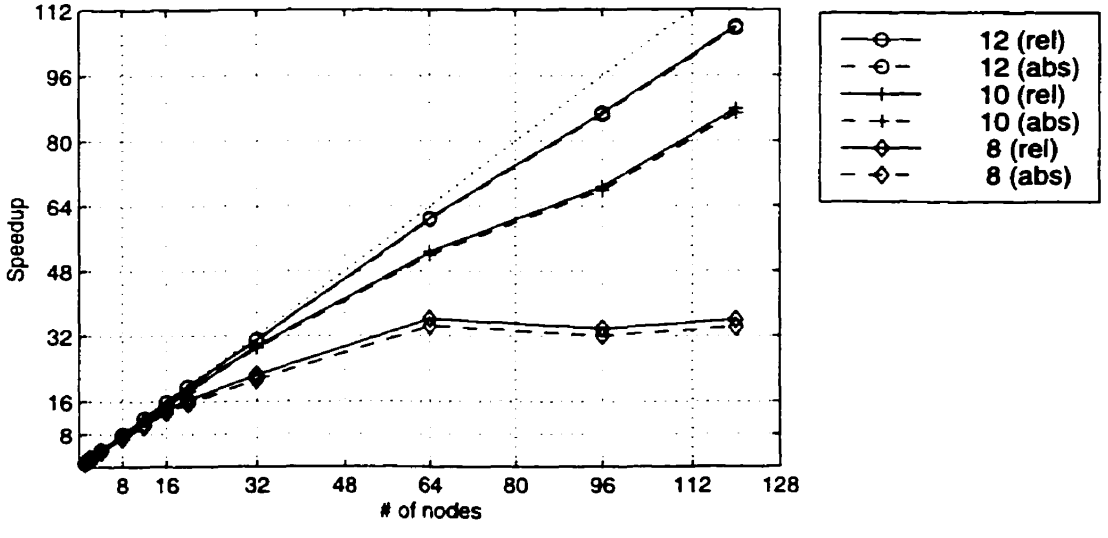


Figure 8.11: Speedups on EARTH-MANNA-SU (Internal) for N-Queens-T

new configuration. Most of the absolute speedups show improvements over the external SU. However, in this case most of the improvement is attributable to the gains in the USE factors. This makes sense, as the primary benefit of combining the

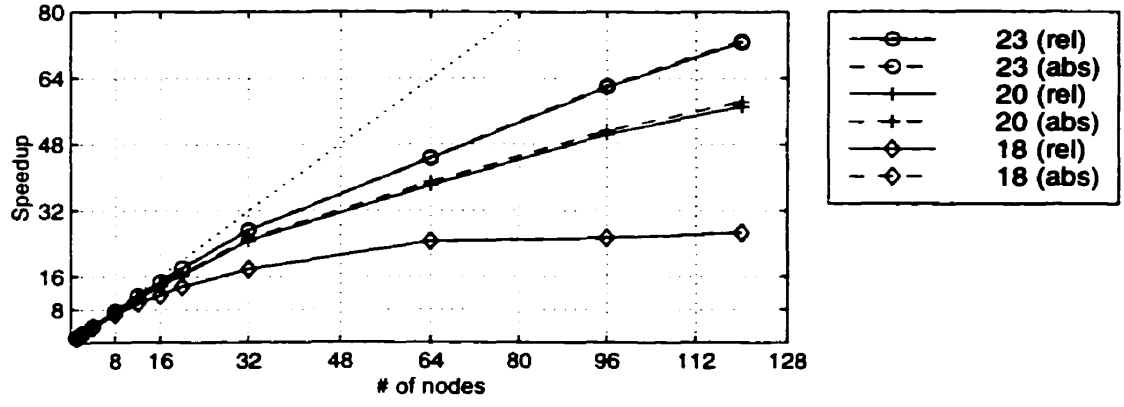


Figure 8.12: Speedups on EARTH-MANNA-SU (Internal) for Paraffins

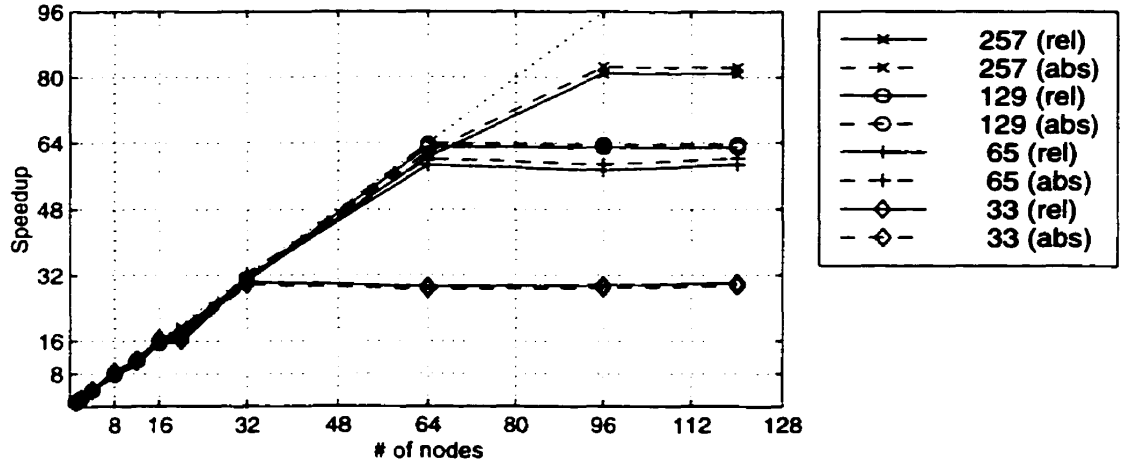


Figure 8.13: Speedups on EARTH-MANNA-SU (Internal) for Tomcatv

EU and SU is improving local interaction between the two; the speed of external access is the same as before. In fact, some of the *relative* speedup curves actually are worse when the EU is internal, though the absolute speedups still show improvement. In these cases, the internal connection enables Execution Units to start new fibers more quickly, which causes them to grab more work from the SU before the load balancer has had a chance to distribute the tokens to other nodes.

8.3 Future Directions

This dissertation has presented a program execution model for parallel multithreading, and outlined an evolutionary path by which this model can be introduced to parallel machines based on off-the-shelf processors, which are then improved by the gradual introduction of hardware customized for this model. The progression of experimental studies from the previous chapter and this chapter demonstrate the improvements which can be expected by following such an evolutionary path. But what are the implications of these results for future architecture development?

In the short term, computer designers may wish to consider starting down the path outlined by this work, supporting the EARTH PXM or a similar model. The risks are minimal, because the EARTH architecture features are *extensions* to a conventional machine. They do not affect the performance of programs that already run on that machine (sequential programs, or parallel programs using an off-the-shelf system such as MPI), which is not the case if one immediately jumps to a custom parallel machine.

The next step after combining the EU and SU cores, as outlined in Section 1.3, is to integrate the PXM fully into a processor. This would mean, for instance, supporting EARTH operations directly using special opcodes rather than accesses to memory locations. This would make fine-grain code even more efficient. For instance, a get_rsync_l operator takes three full addresses (not slot numbers) as arguments, which means that an i860 processor driving an external SU would need three store instructions. Such a sequence might look like the following:

```
st.l r16, 0x2400(r15)
st.l r17, 0(r15)
st.l r18, 0x20(r15)
```

where it is assumed that

- Register r15 contains the base address of the SU unit;
- Register r16 contains the sync slot address;
- Register r17 contains the source address;
- Register r18 contains the destination address;

- The immediate offsets encode tag information for the SU.

This could be replaced by a single instruction, such as

```
getsync.l r17, r18, r16
```

which can easily fits into the format of 3-register instructions in the i860 and most other RISC processors. Further experimentation will be needed to see how much improvement is possible.

There are other possibilities for improvements to the base architecture. Section 4.3 proposed extensions to the EARTH PXM to simplify programming the EARTH, and some of these work best if supported in hardware. Adding intelligence to the Ready Queue to prioritize fibers to exploit locality was also recommended in Section 4.2.5. These can be added to an external SU, though their incorporation into an internal SU would provide even greater benefits. Other potential improvements primarily affect the architecture and are transparent to the PXM. For instance, just as the Register Use Cache (proposed in Section 4.2.5) could enable an EU with multiple register sets to use these efficiently, an analogous module could speculate memory accesses, based on the state of the RQ, and try to manipulate the L2 or L1 cache accordingly. For instance, if the SU sees that a particular fiber is about to be read from the RQ, the SU could ensure that certain parts of that fiber's frame are in the L2 or L1 cache (depending on whether the SU is external or internal) on the assumption that the fiber is likely to access them soon. Another possible improvement, appropriate for an SU which is fully integrated into a processor with multiple register sets, is for the SU to preload a register set with values from the frame before a fiber begins execution.

In the long term, there will be fundamental changes in processor design as chips continue to get larger and denser. The question often asked during the past decade by architects who expect conventional ILP methods to yield diminishing returns is: how can the extra transistors brought by increasing integration be exploited? It is generally assumed that there will be more than one processing element on a single chip, though there are many ways these can be organized.

For instance, the simplest organization scheme is to build independent processor nodes on a single chip, as shown in Figure 8.14. In this abstract architecture, it is assumed that each of the "execution pipelines" fetches and dispatches instructions from a separate thread, and has its own memory, registers and functional units.

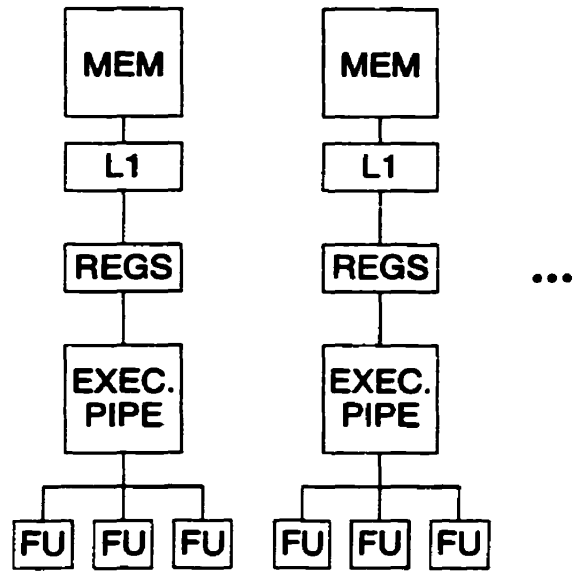


Figure 8.14: Simple Multiple-CPU Organization on Single Chip

"Memory" is used here in a generic sense; some have proposed placing large RAM blocks on the processor chip [72], but this could also be a second-level cache.

Most other proposed organizations can be viewed as a variant of this structure with some components shared between common elements:

1. Functional units can be shared among the execution pipelines, provided that the latency of the interconnections can be kept low. This can increase utilization of the functional unit hardware, as functional units driven by single instruction streams tend to be underutilized due to unevenness in the mix of instructions performed.

2. The preceding structure can be enhanced by giving each execution pipeline a superscalar-like multiple-issue capability, and by sharing register sets and memory. This is the Simultaneous Multithreading architecture [126], whose simulation has demonstrated efficient use of the functional units, leading to a high instruction issue rate, provided there are enough independent threads.

3. Either of the preceding two can have separate cache/memory units, possibly connected internally for maintaining some coherence, or can share the same cache lines and memory through a fast interconnection network.

Arguments have been made for many of the possible multiple-CPU organizations, and it is difficult to predict which will prevail, because the tradeoffs mostly depend on details of the implementation, many of which are unknown at this time. Yet all of these architectures require multiple streams of instructions in order to achieve their potential, and EARTH provides the means for programmers to generate these threads easily. EARTH's two-level thread hierarchy allows different fibers to run with different contexts, so the simpler organizations above would support EARTH adequately, meaning that the EARTH model does not mandate one organization or the other. Synchronization Units could be incorporated into one of these chips, and programmed to exploit whatever sharing is provided.

In conclusion, these multi-CPU designs could all benefit from supporting the EARTH PXM. This would complete the last link in the evolutionary chain from off-the-shelf hardware to full-custom support for our execution model. We have shown in the last two chapters that efficient parallel programs can be written in a language based on the EARTH PXM, and that they can be run with increasing levels of efficiency as the architecture evolves. While this chapter has presented, in a fair amount of detail, possible designs for some of the intermediate points in the path, only time (and much continued research) will tell the best way to incorporate this model into the high-performance chips of the future.

Chapter 9

Other Related Work

Chapter 3 surveyed dataflow machines and multithreaded machines which are based on dataflow principles. Other models of multithreading also have long and rich traditions of research and innovation. This chapter covers some of the work which is most relevant to EARTH.

As with the dataflow-style multithreaded systems in Section 3.2, there are two basic approaches to implementing a multithreaded system. The first is to design custom hardware and the software to drive it, and the second is to focus on software systems for existing multiprocessors. The first approach, covered in Section 9.1, has the potential for greater performance in the long run, but has high short-term costs, especially for academic research projects. Software systems, described in Section 9.2, define a programming model, often in the form of a language or a library of functions, and provide a means of translating the user's application to code which can run on an off-the-shelf system. This is the approach taken by EARTH at the beginning of the evolutionary path (see Section 7.1). Most of the programming models discussed in this section make no reference to specific architecture features, or to hardware support, but it is clear that many could benefit from specialized hardware support for their specific program execution models, as does EARTH.

9.1 Multithreaded Architecture Developments

This section primarily focuses on the thread models of the various machines rather than performance issues. A thread model is mainly concerned with how a program is divided into threads, how threads are created and terminated, how they share

data, and how they are coordinated. Many of the multithreaded machines discussed herein have multiple active threads, each with its own set of registers, allow some sharing of these registers among different threads, and use variations of *full/empty* bits to synchronize the producers and consumers of data.

In this scheme, a register or memory location is tagged with an extra bit which is set when a value is written into that location. A location with an unset tag is considered "empty." If the consumer attempts to read an empty location or write a full location, the transfer is aborted. In some cases, the consumer must keep trying the same operation ("busy-waiting"), while other machines have mechanisms for suspending and automatically restarting such threads (a luxury not available to EARTH).

Threads are typically initiated by special instructions contained within other threads and terminated by special instructions within the same thread. This allows various kinds of scheduling mechanisms, such as fork-join, to be implemented. In some cases, these mechanisms are made explicit in the programming language.

Since off-the-shelf processors are designed for executing only single threads efficiently, with no provision for switching to other threads quickly, the focus in EARTH has been on running threads quickly and without stalls, which means running a thread only when it is ready. Custom processors can employ hardware to switch between threads rapidly, allowing multiple threads to be active at the same time.

This ability to schedule from multiple threads addresses the latency problem (see Section 1.2.1). Long-latency operations won't cause the functional units to lie idle, as in a single-thread processor, because the processor will be able to fill up the idle space with work from other threads. Exactly how the processor chooses which thread(s) to execute varies from machine to machine. The two most common approaches are *interleaving* ("round-robin") and *switch-when-needed.* Examples of machines using interleaving include HEP (Heterogeneous Element Processor), Horizon, MTA (Multi-Threaded Architecture), MASA (Multilisp Architecture for Symbolic Applications), and the M-Machine. Machines which only switch threads when the current thread stalls include Alewife and the J-Machine.

HEP [73] has a simple form of interleaving. There are sixteen *task queues*, eight for user tasks and eight for supervisor tasks. Each queue can hold up to 64 activity specifiers. When the execution pipeline needs work to do, it dequeues the activity specifiers at the heads of all the queues, and then feeds them one by one into the

execution pipe. The other end of the pipe generates updated activity specifiers that are put back into the task queues. The execution pipe itself has eight stages, and has no logic for checking for hazards. If fewer than eight task queues are non-empty when the pipeline fetches activity specifiers from them, then there will be bubbles in the pipeline.

One problem with this arrangement is that at least eight threads must be present on a processor in order to get full use of that processor. This may not always be possible, particularly when executing a sequential portion of the program. Horizon [75] and MTA [4] have a more sophisticated form of interleaving. There still is no hazard-checking logic in the execution pipe. However, a special field in each instruction is set to n to indicate that the next n instructions in the same thread are guaranteed not to have any hazards with the current instruction. Therefore, it is not necessary to have as many threads as pipeline stages, so long as the existing threads have enough independent instructions among them to prevent bubbles in the pipeline. Because instructions in a thread are executed in sequential order, the compiler can analyze the code and calculate the value of n for each instruction.

This form of interleaving makes scheduling more flexible. For instance, if the sequential section of code is short, it is easier to keep the operations in one thread than to create separate parallel threads. However, it still requires that the number of independent instructions that can run in parallel be at least equal to the number of pipeline stages.

Processors which perform extensive thread interleaving demand compilers capable of finding this parallelism. Tera, which builds the MTA, the most ambitious multithreaded project to date, has put as much effort in compiler development as in hardware, with the result that they can extract, from many application programs, enough parallelism to support this interleaving, even when there are high memory latencies.

MASA [50] employs a HEP-like interleaving of threads, with each thread (called a *task*) having a separate register set. MASA was intended primarily for parallel symbolic computations, and was designed to support the language MultiLisp [49], which uses a form of lazy evaluation called *futures*. Because this language involves heavy use of recursion, MASA allows parent and child functions to share register sets, improving efficiency of function calls.

The M-Machine [33] is an advanced successor to the J-Machine (described below). The M-Machine has a sophisticated hierarchy, both of threads and of processing units. The processor is called the Multi-ALU Processor (MAP) [68], which contains three execution *clusters*, each of which can issue up to three instructions per cycle (two integer, one FP) in a VLIW-like manner.

An instruction stream is partitioned into *horizontal threads* (*H-Threads*) and *vertical threads* (*V-Threads*). Each H-Thread runs on a specific cluster and instructions are grouped in triples to exploit the 3-instruction-issue capability of the cluster. H-Threads are then grouped into V-Threads (up to 3 per V-Thread) and when a V-Thread runs, each of its H-Threads runs on a separate cluster. H-Threads can communicate with other H-Threads in the same V-Thread and synchronize with them through registers. Up to five V-Threads may be active at the same time. In each cycle, each cluster chooses one of the five H-Threads (the H-Threads on that cluster belonging to the active V-Threads) and issues all three instructions. The H-Threads are interleaved cycle-by-cycle.

The Alewife machine [1] only interleaves when necessary. Alewife is based on the Sparcle chip, a modified Sparc processor [2]. This processor can use the bypass logic that comes with a normal sequential processor. Therefore, the processor executes one thread sequentially until the thread encounters a long-latency operation, such as a remote fetch, and then switches to another thread. Sparcle takes advantage of the multiple register windows present in the normal Sparc processor by creating separate thread contexts. Since the Sparc already has logic for rapid switching between register windows, thread switching is very fast. Since the Sparc has only enough registers for four independent contexts, software tries to stay within one group of threads as much as possible, because it is much faster to switch register windows than to load in a new context.

The J-Machine [95] is based on a processor supporting efficient message passing. Context-switch times are kept low by only having four registers per context, allowing contexts to be switched in 10 cycles. A second register set allows the processor to handle two priority levels without switching.

Machines such as EARTH must adapt to whatever instruction set comes with the off-the-shelf processors on which they are based. With a custom processor, designers have the ability to support a specific multithreading program execution model more fully in hardware, which gives them more flexibility in choosing such a

model. Therefore, when comparing one of these machines with EARTH, one should remember that the two architectures were built for different purposes. EARTH is intended for off-the-shelf processors, with at most only moderate hardware augmentation, and is thus necessarily kept simple. It is also useful to keep in mind that many of the machines described in this section, as well as most of those in Chapter 3, have never actually been built, while other machines that *were* constructed had such a large lag time that their single-node performance lagged behind state-of-the-art microprocessors. This does not mean that the designs are impractical, but they do suffer the problems of competing with the "killer micros" as mentioned in the Introduction.

Recently, there has been a flurry of research in the use of multithreading to boost the performance of a single processor rather than to drive a large parallel machine. These proposals don't provide programming models for exploitation of large-scale parallelism, but rather a framework for improving the speed of programs executed according to ordinary sequential semantics. Nevertheless, the ideas may be applicable to EARTH and could be used to increase the speed of Execution Units on an EARTH machine. Since they are intended for conventional sequential programs, it is possible that one or more of these ideas could find their way into off-the-shelf processors in a few years.

Simultaneous Multithreading [126], which was discussed in Sections 4.2.1 and 8.3, combines the moderate levels of ILP within individual threads exploitable by superscalar units with the parallelism and latency-tolerance of running multiple threads concurrently. Up to eight threads may be active (meaning that they are allocated registers) at one time. In each cycle, the instruction fetch unit chooses two threads that are not currently stalled (e.g., due to an I-cache miss) and fetches eight instructions from each. It then narrows this group down to a total of eight independent instructions, and sends these to the functional units. Simulations show that high functional unit utilization rates can be achieved with this organization. Simultaneous Multithreading is not designed specifically for single applications which are multithreaded for speed, but is intended for merging separate task streams together, such as in database programs [81].

Thread-level speculation can also be used to get higher performance out of a multithreaded processor. In Multiscalar Processors [107], a program is divided into *tasks*, where a task is a section of the control flow graph (CFG) whose execution

corresponds to a contiguous region of the dynamic instruction sequence (e.g., a loop body). There is a sequential order among the tasks defined by the program semantics, and tasks are speculatively sent to processing units for execution by a global sequencer in this order. Hardware continually checks for dependences between instructions in different tasks. If the hardware determines that a data or control dependence was violated (e.g., a later task consumed a value before that value was properly produced by the consumer), then the later task, and any tasks *following* that task that were speculatively started, are squashed. A similar form of speculation, used for speeding up single-thread programs, is found in the Superthread Architecture [125].

A more restricted form of task speculation is done by the Single-Program Speculative Multithreading (SPSM) architecture [29]. At certain points in a program, *fork* instructions may be added to start another stream of execution at a specified address later in the instruction stream. Execution simultaneously continues at the instruction following the fork. The new thread stops when it executes an explicit *suspend* instruction, and is merged with the original thread when the latter catches up to the address where the new thread started. As with the previous two machines, there are mechanisms to squash the new thread if hardware subsequently detects a dependence that was violated.

9.2 Software Multithreading Systems

One of the premises of the EARTH project is that it is possible to implement an efficient multithreading system using off-the-shelf hardware, rather than building specialized processors. Other projects have also demonstrated this. The TAM project, which is based on dataflow principles, is discussed in Section 3.2.2. There are other software systems, not based on dataflow, which we cover here.

One popular approach is adding support for multithreading as a library or extension to a normally non-multithreaded system. This can be done with a threads package such as POSIX Threads or Solaris Threads [79]. These provide libraries of functions which can be called from user programs. These functions provide for the creation, termination, and synchronization of "lightweight" threads with much less state than a typical OS-level process. Most libraries provide a diverse set of functions supporting many different types of communication patterns, such

as divide-and-conquer and producer-consumer, and provide most of the well-known synchronization techniques such as fork-join and mutual exclusion. Threads have also been included as a basic part of the Java language [96].

With such systems, it is possible to implement coarse-grain multithreaded programs much more efficiently than if conventional OS processes were used. Unfortunately, these systems are typically built on top of kernel-level threads, and rely on the kernel for thread interaction. These make the overheads of calling these functions much higher than for an implementation such as EARTH. Furthermore, the state of a lightweight thread, though much smaller than the state of a process, is still considerably larger than the state of a procedure instance or fiber in EARTH. These systems, therefore, are not practical for fine-grain parallel programming.

The Cilk system [35] solves this problem by taking the OS out of the picture. Cilk is a programming language in which standard C is augmented with special keywords for supporting parallel function calls and synchronizations between a function and its children. It has been implemented on several uniprocessors and symmetric multiprocessors with shared memory.

The primary source of parallelism in Cilk programs is parallel function invocation. Functions *spawn* other functions, much like the token operator in EARTH, except that Cilk functions have an explicit return value. Barriers are used to block execution of the function until the child function(s) have returned their values.

One of the unique features of Cilk is the *elision* property: if all Cilk keywords are removed from a *semantically correct Cilk function without race conditions*, the result is a sequential function which is equivalent to the original parallel function, producing the same results. The advantage of this is that the compiler can generate two versions of the code for each function, called the *fast clone* and the *slow clone*. The fast clone is simply a sequential version of the function (made by ignoring the Cilk keywords), and the slow clone is the parallel version with all of its overheads. Function calls are placed in a dequeue pending invocation, much as EARTH places tokens in the Token Queue, and both systems use work-stealing to distribute work among nodes. The difference between the two is when the Cilk runtime system removes a function instance from the top of its local dequeue, it invokes the fast clone, avoiding the overheads of a parallel function call. The use of a dequeue prevents the number of simultaneously-active instances from growing exponentially in recursive programs, much as it does in EARTH. More importantly, however, it causes most of

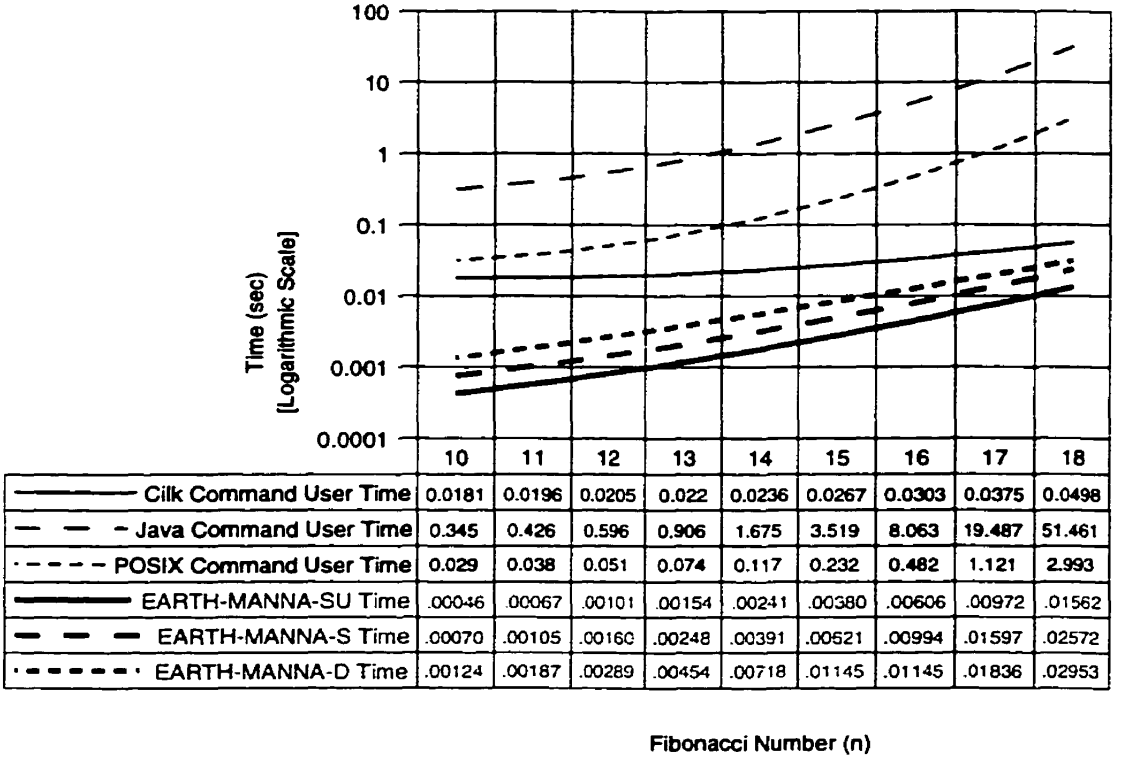
the function instances to be executed sequentially, improving efficiency considerably. A Cilk system running small, highly-recursive functions (such as N-Queens) does not need to rely either on special hardware (such as an SU) to reduce the overheads of multithreading or on efforts by the programmer to restrain parallelism (as with the EARTH implementation of N-Queens-T).

This makes Cilk highly effective for divide-and-conquer programs. The utility of Cilk in this area has been demonstrated; chess programs written in Cilk have won awards at several international computer chess tournaments. There are, however, two disadvantages to Cilk:

1. In its current form, it can run efficiently only on shared-memory machines. Execution on distributed-memory machines would conflict with the elision property. Consider the N-Queens problem, described in Chapter 6. Each procedure instance needs to make a private copy of the data structure holding the current state of the board, in order to avoid interfering with other procedures running in parallel. To maintain elision, this would have to be written as a plain copy (e.g., assignment of array elements within a loop). On a shared-memory system, this simply reads from shared memory. On a distributed-memory system, if the child and parent function are on different nodes, then the copy might be disallowed, or copy from a local address rather than the desired remote address, or trigger a remote load which would stall the processor.

2. Maintaining the elision property limits the types of programming paradigms which are supported. Communication is normally between parent and child only. Peer communications, such as in a producer-consumer relationship or bidirectional data exchange (as used in some of the applications coded in Threaded-C, such as Tomcatv) are difficult to express in Cilk.

Figure 9.1 compares the performance of threaded implementations of Fibonacci running various problem sizes on one processor. The runtimes for Cilk, Java and P-Threads were reported in a multithreading workshop in 1998 [113] and were taken from a 167MHz UltraSPARC processor. The runtimes for EARTH are taken from the MANNA, with its 50MHz i860 processor (the EARTH-MANNA-SU results were obtained using SEMi).



	10	11	12	13	14	15	16	17	18
Cilk Command User Time	0.0181	0.0196	0.0205	0.022	0.0236	0.0267	0.0303	0.0375	0.0498
Java Command User Time	0.345	0.426	0.596	0.906	1.675	3.519	8.063	19.487	51.461
POSIX Command User Time	0.029	0.038	0.051	0.074	0.117	0.232	0.482	1.121	2.993
EARTH-MANNA-SU Time	.00046	.00067	.00101	.00154	.00241	.00380	.00606	.00972	.01562
EARTH-MANNA-S Time	.00070	.00105	.00160	.00248	.00391	.00521	.00994	.01597	.02572
EARTH-MANNA-D Time	.00124	.00187	.00289	.00454	.00718	.01145	.01145	.01836	.02953

Figure 9.1: Comparison of Multithreaded Systems

Chapter 10

Conclusions

This dissertation has presented EARTH, an Efficient Architecture for Running Threads, which is a multithreaded system based on dataflow principles, but designed to be efficiently implemented using off-the-shelf processors.

For years, computer architects have been drawn to the promise of parallel computing, only to be disappointed by the results. The problem with parallel computing in the 90s is primarily economic; custom processors designed to support special parallel paradigms can't command the same resources as mass-market microprocessors in a fiercely-competitive market. Most parallel systems, therefore, have been built around these "killer micros," only to suffer problems with latency, synchronization, and programmability because off-the-shelf processors haven't had the need to address these issues.

In order to penetrate the market successfully, parallel systems need two things. First, they need a programming model which allows programmers to express parallelism easily, but efficiently. Second, there must be an evolutionary path which allows a gradual migration from a simple multiprocessor based on off-the-shelf components to a custom hardware implementation specifically designed to support the programming model. The programming model should enable most programs to be implemented reasonably efficiently on off-the-shelf multiprocessors at the beginning of the evolutionary path, while permitting performance gains as custom hardware support for the model is added little by little. Such custom improvements should not hinder ordinary sequential code in any way, for that would simply cause a rejection of such improvements by most of the market.

EARTH is a multithreaded machine based on dataflow principles. Multithreading solves the problem of latency in multiprocessors, because a processor can switch to another thread whenever there is a long-latency operation such as a remote fetch. Dataflow provides an effective method of synchronization, based on the simple principle that a thread should not commence until its operands are ready. Combining the two yields a machine with the strengths of both.

The requirement for efficient execution on off-the-shelf processors imposes constraints on the types of threads that can be used on EARTH. The EARTH Program Execution Model (PXM) is based on a two-level hierarchy of fibers and procedures. The former are non-preemptive, eliminating the need to switch threads in midstream, which can be costly on commodity processors. Fibers in the same procedure instance share data in a frame seen by all these fibers. Fibers synchronize their activities with each other through special EARTH operations, which enforce the dataflow principles on which EARTH is based.

The EARTH Architecture Model describes the components and behavior of a generic machine implementing the EARTH PXM. In this model, an EARTH node consists of an off-the-shelf microprocessor for executing fibers (Execution Unit) and a second unit for synchronizing between threads. This second unit (the Synchronization Unit) can be implemented in custom hardware as a small, simple ASIC chip (its functionality is not complex) or by an off-the-shelf co-processor running a software emulator of the SU's functions.

For the first step in the evolutionary chain, the PXM was implemented on the MANNA multiprocessor, which is based on the i860 processor. First, the abstract PXM was fleshed out more fully, in a form called the EARTH Virtual Machine, which serves as a specification of the EARTH data types and operations while leaving normal processor-dependent components unspecified, for maximal flexibility. Then the Threaded-C language was constructed based on the EVM. Finally, a runtime system was constructed to emulate the behavior of the SU on the second processor of each MANNA node. Results showed good speedups up to 20 nodes on most applications, and reasonably low overheads for the multithreading operations, provided they do not dominate the code.

Extending the speedup curves out to 120 nodes using a simulator with accurate timing revealed limitations in the load balancer and other components of the emulated-SU system. Use of the same tool to simulate a custom SU with special

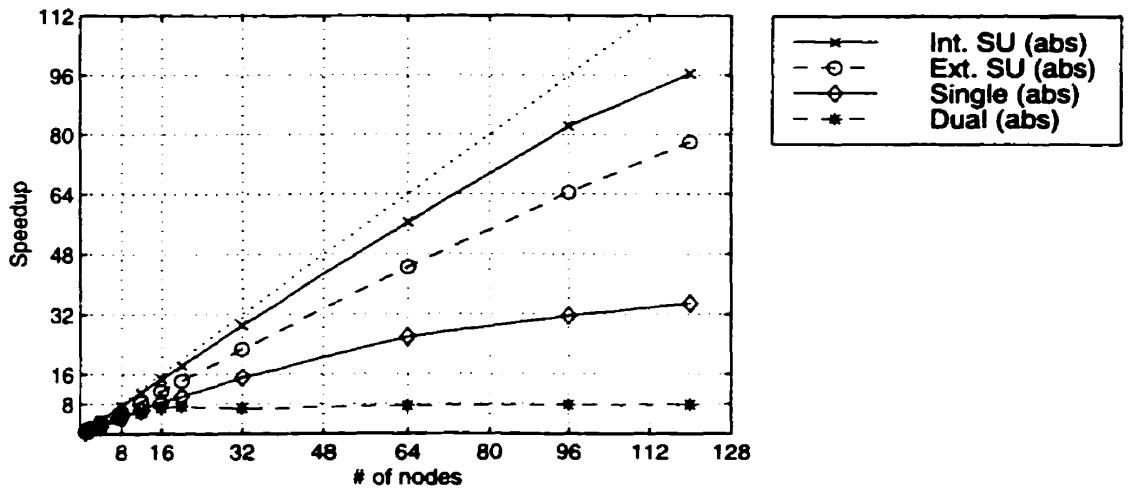


Figure 10.1: Comparison of EARTH Implementations for N-Queens-P (10)

support for the EARTH operations demonstrated that moving along the evolutionary path, first by constructing a separate hardware SU external to the EU, and then by placing the SU on the same chip as the EU, would further reduce multithreading overheads, reduce the latencies between fibers in different nodes, and improve load balancing dramatically. (Figure 10.1 shows a sample comparison between four implementations on the path running the fully-parallel version of N-Queens for 10 queens, showing that a custom SU improves both the overall speed and the scalability, i.e., the speedups are far more linear at the high end.) Moving along the path would therefore reduce the penalties for having excess parallelism in a program, which would spare the programmer the effort of having to reduce the number of threads in a program through algorithmic changes.

We can conclude that the EARTH model does provide the basis for efficient parallel programming using off-the-shelf technology, even for the machines at the near end of the evolutionary path, and that the costs and risks of moving along the path are fairly small, since the bulk of the costs, as measured by design time, are still represented by the core logic of the off-the-shelf processor.

10.1 Future Work

This dissertation has defined an abstract programming model, specified a virtual machine implementing the model, developed a programming language with which one can write code for this machine, and presented implementations of this virtual machine on various platforms along the evolutionary path. Although this research has achieved its goals by demonstrating the viability of this evolutionary approach, the topic is not closed by any means. There are many directions for future research, some of which have already been discussed in previous chapters or which can be inferred from their contents.

One obvious continuation of the current work is to continue to fill in data points along the evolutionary path. This means, for instance, building a hardware SU according to the specifications in Chapter 8 or something similar. The far end of the path mapped out in Section 1.3, integrating an SU into the logic of a microprocessor and adding EARTH operations to its instruction set, was discussed in Section 8.3 but not carried out. Clearly this step would improve the speed of an EARTH node but it would be interesting to see by how much. Another interesting area to explore would be EU design, constructing an EU which can execute multiple threads simultaneously. Can such processors, such as Simultaneous Multithreading, work with the EARTH model, as claimed in Chapter 8? What about Multiscalar and other multithreaded processors discussed at the end of Section 9.1?

Several previous chapters have discussed possible improvements to the PXM and the EVM. Section 4.3 listed several features which would be desirable additions to the EARTH PXM, such as support for mutual exclusion and speculative execution. Section 5.2 proposed an alternate EARTH Virtual Machine based on abstract frame identifiers rather than absolute addresses. Some benefits were listed, such as enhanced error detection.

Another, more useful benefit of EVM-F would be the possibility for dynamic procedure migration, i.e., moving a procedure instance to another node *after* the procedure has started execution. This would be difficult with global addresses, because other procedures which have references to objects in the frame being moved would have to be given new addresses, but easy if the mapping from FID to node number could be changed dynamically. Of course, changes to the SU or runtime system would be required to support such migration, and further research would

need to determine if migration should be automatic or controlled by the program. This feature would be most useful for applications in which fibers in a procedure are executed and exchange data with other procedures repeatedly,[1] especially if the work load changes unpredictably over time. One important class of problems with these characteristics is the class of adaptive mesh applications, in which an irregular mesh changes over time to adapt to changes in the objects being modeled by the mesh [51].

One important issue affecting parallel systems is deadlock. Up to now, deadlock has not been addressed in the EARTH project. While the pure dataflow model underlying EARTH is deadlock-free, EARTH gives more flexibility to the programmer, and it is possible to write Threaded-C programs that reach a deadlock state because two or more fibers are waiting for sync signals from the other. EARTH programs can also deadlock because system resources are exhausted during program execution. Further research should be done in both areas. First, programming tools can be developed to detect deadlock, or the programming language can be restricted to make it impossible to write deadlocking programs. Second, the resource limitation problem should be addressed by adding mechanisms to detect resource constraints at runtime.

Another area where improvements would be quite useful is the Threaded-C language. Historically, Threaded-C was seen primarily as a target for translation from a language which hides threading details from the programmer, so the mapping from Threaded-C to EVM-A was made very direct to simplify compiling and allow the use of C compilers native to the target processors. However, for many applications, programmers can produce more efficient code if given explicit control over the threads. For these uses, Threaded-C should be restructured and made more programmer-friendly. For instance, mechanisms could be added to count synchronizations and set the initial and reset counts of sync slots automatically (wrong counts are a common bug in Threaded-C code). Another possibility is creating a library of communication templates (e.g., the nearest neighbor communication pattern used in Section 6.2.3) and allowing the programmer to express the data dependences in terms of these templates.

Other languages could also be given support for EVM-A or EVM-F. C++ and Java, which have become popular since the introduction of Threaded-C, come to

[1] It would be useless for one-time fibers like those in Fibonacci or N-Queens.

mind. Even Fortran could be threaded. It would be another interesting research project to see if library-based parallel programming systems, such as P-Threads and MPI, can be converted to EARTH through translation of the library calls.

Another interesting topic to explore is the addition of improved multiprocessor memory models to EARTH. Section 4.1.2 states that the EARTH PXM does not include a memory model, since that would limit its portability across off-the-shelf platforms. However, this essentially limits EARTH to a distributed-memory model, though EARTH could be run on a shared-memory machine as pointed out in Section 4.1.2. Recent research in memory models has shown that the *Location Consistency* model [43, 44] addresses problems with traditional shared-memory systems and can lead to a highly-efficient cache-coherence implementation. It would be interesting to see if this model can be integrated with the EARTH PXM to simplify writing code for EARTH by reducing the need for programmers to include explicit split-phase transactions in their code.

EARTH was primarily designed for supporting off-the-shelf processors. However, this does not mean the EARTH PXM is not useful for non-commodity processors. Several ongoing projects involving custom-designed processors have program execution models based on EARTH.

The Superstrand Architecture [87] shows that the adoption of EARTH-style fibers (synchronized and non-preemptive) can reduce the hardware requirements of modern uniprocessors substantially, without losing performance. In this architecture, *strands* are similar to EARTH fibers, but do not allow conditional branches, avoiding the need for branch prediction. Flow control is handled by conditional *synchronizations*, and sync slots are handled efficiently by a unique hardware mechanism similar to scoreboarding. Automatic thread-partitioning techniques can be used to convert ordinary sequential code into threaded code with a moderate amount of parallelism, enough to take advantage of a Superstrand uniprocessor.

The Hybrid Technology Multi-Threading (HTMT) project [39] is a long-term study of the feasibility of achieving a sustained speed of 1 petaFLOPS (10^{15} FLOPS) by combining high-speed superconductor processors, semiconductor memories with built-in processors, high-speed optical interconnects, and high-density holographic storage. Roughly a dozen research groups across the United States are involved. Memory latencies are expected to range from a few clock cycles for processor registers to the order of a million cycles for the largest memories, so latency tolerance is

crucial for HTMT. Therefore, its Program Execution Model [45] builds on the knowledge and experience gained from EARTH. The HTMT PXM is partially based on the EARTH PXM, but adds a stringent requirement that before a fiber can begin execution, the data it needs must be not only present, but also *physically close to the processor*, to avoid huge miss penalties. HTMT's Processor-In-Memory modules [72] collect data and contexts and "percolate" them to and from the processors.

This dissertation has shown that the EARTH model leads to efficient parallel systems which can be built from off-the-shelf processors intended for the uniprocessor market. The preceding two projects show that the basic principles of EARTH go beyond its original intended domain, and that many applications which face the challenges of large-scale multiprocessing can benefit from the ideas in this thesis.

Bibliography

[1] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Kenneth Mackenzie, and Donald Yeung. The MIT Alewife machine: Architecture and performance. In *Proc. of the 22nd Ann. Intl. Symp. on Computer Architecture*, pages 2–13, Santa Margherita Ligure, Italy, Jun. 1995.

[2] Anant Agarwal, John Kubiatowicz, David Kranz, Beng-Hong Lim, Donald Yeung, Godfrey D'Souza, and Mike Parkin. Sparcle: An evolutionary processor design for multiprocessors. *IEEE Micro*, 13(3):48–61, Jun. 1993.

[3] Anant Agarwal, Beng-Hong Lim, David Kranz, and John Kubiatowicz. APRIL: A processor architecture for multiprocessing. In *Proc. of the 17th Ann. Intl. Symp. on Computer Architecture*, pages 104–114, Seattle, Wash., May 1990.

[4] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. In *Conf. Proc., 1990 Intl. Conf. on Supercomputing*, pages 1–6, Amsterdam, The Netherlands, Jun. 1990.

[5] Makoto Amamiya. An ultra-multiprocessing architecture for functional languages. In Gaudiot and Bic [47], chapter 3, pages 95–119.

[6] José Nelson Amaral and Guang R. Gao. Implementation of I-structures as a library of functions in Portable Threaded-C. CAPSL Tech. Note 04, Dept. of Elec. and Computer Eng., U. of Delaware, Newark, Delaware, Jun. 1998.

[7] Boon Seong Ang, Arvind, and Derek Chiou. StarT the Next Generation: Integrating global caches and dataflow architecture. CSG Memo 354, Computation Structures Group, MIT Lab. for Comp. Sci., Aug. 1994.

[8] Arvind and Kim P. Gostelow. The U-Interpreter. *Computer*, 15(2):42–49, Feb. 1982.

[9] Arvind and Robert A. Iannucci. A critique of multiprocessing von Neumann style. In *Proc. of the 10th Ann. Intl. Symp. on Computer Architecture*, pages 426–436, Stockholm, Sweden, Jun. 1983.

[10] Arvind and Rishiyur S. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Trans. on Computers*, 39(3):300–318, Mar. 1990.

[11] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-structures: Data structures for parallel computing. *ACM Trans. on Programming Languages and Systems*, 11(4):598–632, Oct. 1989.

[12] Todd M. Austin and Gurindar S. Sohi. Dynamic dependency analysis of ordinary programs. In *Proc. of the 19th Ann. Intl. Symp. on Computer Architecture*, pages 342–351, Gold Coast, Australia, May 1992.

[13] David H. Bailey. Misleading performance in the supercomputing field. In *Proc. of Supercomputing '92*, pages 155–158, Minneapolis, Minn., Nov. 1992. Invited talk.

[14] P. S. Barth, R. S. Nikhil, and Arvind. M-structures: Extending a parallel, non-strict, functional language with state. CSG Memo 327, Computation Structures Group, MIT Lab. for Comp. Sci., Mar. 1991.

[15] Donald J. Becker, Thomas Sterling, Daniel Savarese, John E. Dorband, Udaya A. Ranawak, and Charles V. Packer. Beowulf: A parallel workstation for scientific computation. In *Proc. of the 1995 Intl. Conf. on Parallel Processing*, volume I, Oconomowoc, Wisconsin, Aug. 1995.

[16] Eugene Brooks. The attack of the killer micros, Nov. 1989. Presentation in the Teraflop Computing Panel Discussion at *Supercomputing '89*.

[17] U. Bruening, W. K. Giloi, and W. Schroeder-Preikschat. Latency hiding in message-passing architectures. In *Proc. of the 8th Intl. Parallel Processing Symp.*, pages 704–709, Cancún, Mexico, Apr. 1994. IEEE Comp. Soc.

[18] Michael Butler, Tse-Yu Yeh, Yale Patt, Mitch Alsup, Hunter Scales, and Michael Shebanow. Single instruction stream parallelism is greater than two. In *Proc. of the 18th Ann. Intl. Symp. on Computer Architecture*, pages 276–286, Toronto, Ontario, May 1991.

[19] Haiying Cai. Dynamic load balancing on the EARTH-SP system. Master's thesis, McGill U., Montréal, Qué., May 1997.

[20] L. E. Cannon. *A Cellular Computer to Implement the Kalman Filter Algorithm.* PhD thesis, Montana State U., Bozeman, Montana, 1969.

[21] David E. Culler, Anurag Sah, Klaus Erik Schauser, Thorsten von Eicken, and John Wawrzynek. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In *Proc. of the Fourth Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 164–175, Santa Clara, Calif., Apr. 1991.

[22] William J. Dally. Directions in concurrent computing. In *Proc. of the IEEE Intl. Conf. on Computer Design*, pages 102–106, Port Chester, N. Y., Oct. 1986.

[23] Jack B. Dennis. First version of a data-flow procedure language. In *Proc. of the Colloque sur la Programmation*, number 19 in Lec. Notes in Comp. Sci., pages 362–376, Paris, France, Apr. 9–11, 1974. Springer-Verlag.

[24] Jack B. Dennis. Data flow supercomputers. *Computer*, 13(11):48–56, Nov. 1980.

[25] Jack B. Dennis and Guang R. Gao. An efficient pipelined dataflow processor architecture. In *Proc. of Supercomputing '88*, pages 368–373, Orlando, Flor., Nov. 1988.

[26] Jack B. Dennis, Guang-Rong Gao, and Kenneth W. Todd. Modeling the weather with a data flow supercomputer. *IEEE Trans. on Computers*, 33(7):592–603, Jul. 1984.

[27] Jack B. Dennis and David P. Misunas. A preliminary architecture for a basic data-flow processor. In *Proc. of the 2nd Ann. Symp. on Computer Architecture*, pages 126–132, Houston, Tex., Jan. 1975.

[28] Jack J. Dongarra, Iain S. Duff, Danny C. Sorensen, and Henk A. van der Vorst. *Numerical Linear Algebra for High-Performance Computers.* Soc. for Industrial and Applied Mathematics, Philadelphia, Penn., 1998.

[29] Pradeep K. Dubey, Kevin O'Brien, Kathryn O'Brien, and Charles Barton. Single-program speculative multithreading (SPSM) architecture: Compiler-assisted fine-grained multithreading. In *Proc. of the IFIP WG 10.3 Working Conf. on Parallel Architectures and Compilation Techniques, PACT '95*, pages 109–121, Limassol, Cyprus, Jun. 1995.

[30] Nasser Elmasri. TCL: Experiences on a multiprocessor with dual-processor nodes. Master's thesis, McGill U., Montréal, Qué., Jul. 1995.

[31] John T. Feo, David C. Cann, and Rodney R. Oldehoeft. A report on the Sisal language project. *J. of Parallel and Distrib. Computing*, 10(4):349–366, Dec. 1990.

[32] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. on Programming Languages and Systems*, 9(3):319–349, Jul. 1987.

[33] Marco Fillo, Stephen W. Keckler, William J. Dally, Nicholas P. Carter, Andrew Chang, Yevgeny Gurevich, and Whay S. Lee. The M-Machine multicomputer. In *Proc. of the 28th Ann. Intl. Symp. on Microarchitecture*, pages 146–156, Ann Arbor, Mich., Nov.–Dec. 1995.

[34] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. on Computers*, 21(9):948–960, Sep. 1972.

[35] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proc. of the ACM SIGPLAN '98 Conf. on Programming Language Design and Implementation*, pages 212–223, Montréal, Qué., Jun. 1998.

[36] Daniel D. Gajski, David A. Padua, David J. Kuck, and Robert H. Kuhn. A second opinion on data flow machines and languages. *Computer*, 15(2):58–69, Feb. 1982.

[37] G. R. Gao. An efficient hybrid dataflow architecture model. *J. of Parallel and Distrib. Computing*, 19(4):293–307, Dec. 1993.

[38] G. R. Gao and K. Theobald. An enable memory controller chip for a static data flow computer. CSG Note 18, Computation Structures Group, MIT Lab. for Comp. Sci., Jan. 1985.

[39] Guang Gao, Konstantin K. Likharev, Paul C. Messina, and Thomas L. Sterling. Hybrid technology multi-threaded architecture. In *Proc. of Frontiers '96: The Sixth Symp. on the Frontiers of Massively Parallel Computation*, pages 98–105, Annapolis, Maryland, Oct. 1996.

[40] Guang R. Gao. Maximum pipelining linear recurrence on static data flow computers. *Intl. J. of Parallel Programming*, 15(2):127–149, Apr. 1986.

[41] Guang R. Gao, Lubomir Bic, and Jean-Luc Gaudiot, editors. *Advanced Topics in Dataflow Computing and Multithreading.* IEEE Comp. Soc. Press, 1995. Book contains papers presented at the Second Intl. Work. on Dataflow Computers, Hamilton Island, Australia, May 1992.

[42] Guang R. Gao, Herbert H. J. Hum, and Yue-Bong Wong. Parallel function invocation in a dynamic argument-fetching dataflow architecture. In *Proc. of PARBASE-90: Intl. Conf. on Databases, Parallel Architectures, and Their Applications*, pages 112–116, Miami Beach, Flor., Mar. 7–9, 1990. IEEE Comp. Soc.

[43] Guang R. Gao and Vivek Sarkar. Location consistency: Stepping beyond memory coherence barrier. In *Proc. of the 1995 Intl. Conf. on Parallel Processing*, volume II, pages 73–76, Oconomowoc, Wisconsin, Aug. 1995.

[44] Guang R. Gao and Vivek Sarkar. On the importance of an end-to-end view of memory consistency in future computer systems. In *Proc. of the Intl. Symp. on High Performance Computing*, pages 30–41, Fukuoka, Japan, 1997.

[45] Guang R. Gao, Kevin B. Theobald, Andrés Márquez, and Thomas Sterling. The HTMT program execution model. CAPSL Tech. Memo 09, Dept. of Elec. and Computer Eng., U. of Delaware, Newark, Delaware, Jul. 1997. In ftp://ftp.capsl.udel.edu/pub/doc/memos.

[46] Guang Rong Gao. A pipelined code mapping scheme for static dataflow computers. Tech. Rep. MIT/LCS/TR-371, MIT Lab. for Comp. Sci., Aug. 1986. PhD thesis.

[47] Jean-Luc Gaudiot and Lubomir Bic, editors. *Advanced Topics in Data-Flow Computing.* Prentice-Hall, Englewood Cliffs, N. Jersey, 1991. Book contains papers presented at the First Workshop on Data-Flow Computing, Eilat, Israel, May 1989.

[48] J. R. Gurd, C. C. Kirkham, and I. Watson. The Manchester prototype dataflow computer. *Comm. of the ACM*, 28(1):34–52, Jan. 1985.

[49] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Trans. on Programming Languages and Systems*, 7(4):501–538, Oct. 1985.

[50] Robert H. Halstead, Jr. and Tetsuya Fujita. MASA: A multithreaded processor architecture for parallel symbolic computing. In *Proc. of the 15th Ann. Intl.*

Symp. on Computer Architecture, pages 443–451, Honolulu, Haw., May–Jun. 1988.

[51] Gerd Heber, Rupak Biswas, and Guang R. Gao. Using multithreading for the automatic load balancing of adaptive finite element meshes. In H. D. Simon, editor, *Proc. of the 5th Symp. on Solving Irregularly Structured Problems in Parallel*, volume 1457 of *Lec. Notes in Comp. Sci.*, Berkeley, Calif., 1998. Springer-Verlag.

[52] Laurie J. Hendren, Xinan Tang, Yingchun Zhu, Guang R. Gao, Xun Xue, Haiying Cai, and Pierre Ouellet. Compiling C for the EARTH multithreaded architecture. In *Proc. of the 1996 Conf. on Parallel Architectures and Compilation Techniques (PACT '96)*, pages 12–23, Boston, Mass., Oct. 1996.

[53] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Pub., Inc., San Mateo, Calif., 1990.

[54] Dana S. Henry and Christopher F. Joerg. A tightly-coupled processor-network interface. In *Proc. of the Fifth Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 111–122, Boston, Mass., Oct. 1992.

[55] Kei Hiraki, Satoshi Sekiguchi, and Toshio Shimada. Status report of SIGMA-1: A data-flow supercomputer. In Gaudiot and Bic [47], chapter 7, pages 207–223.

[56] Herbert H. J. Hum and Guang R. Gao. A novel high-speed memory organization for fine-grain multi-thread computing. In *Proc. of PARLE '91 - Parallel Architectures and Languages Europe,* volume I, number 505 in Lec. Notes in Comp. Sci., pages 34–51, Eindhoven, The Netherlands, Jun. 1991. Springer-Verlag.

[57] Herbert H. J. Hum and Guang R. Gao. A high-speed memory organization for hybrid dataflow/von Neumann computing. *Future Generation Computer Systems*, 8(4):287–301, Sep. 1992.

[58] Herbert H. J. Hum, Olivier Maquelin, Kevin B. Theobald, Xinmin Tian, Guang R. Gao, and Laurie J. Hendren. A study of the EARTH-MANNA multithreaded system. *Intl. J. of Parallel Programming*, 24(4):319–347, Aug. 1996.

[59] Herbert H. J. Hum, Olivier Maquelin, Kevin B. Theobald, Xinmin Tian, Xinan Tang, Guang R. Gao, Phil Cupryk, Nasser Elmasri, Laurie J. Hendren, Alberto Jimenez, Shoba Krishnan, Andres Marquez, Shamir Merali, Shashank S. Nemawarkar, Prakash Panangaden, Xun Xue, and Yingchun Zhu. A design study of the EARTH multiprocessor. In *Proc. of the IFIP WG 10.3 Working Conf. on Parallel Architectures and Compilation Techniques, PACT '95*, pages 59–68, Limassol, Cyprus, Jun. 1995.

[60] Herbert H. J. Hum, Kevin B. Theobald, and Guang R. Gao. Definition of the Multi-Threaded Architecture (MTA) model. ACAPS Tech. Note 40, Sch. of Comp. Sci., McGill U., Montréal, Qué., Oct. 1993.

[61] Herbert H. J. Hum, Kevin B. Theobald, and Guang R. Gao. Building multithreaded architectures with off-the-shelf microprocessors. In *Proc. of the 8th Intl. Parallel Processing Symp.*, pages 288–294, Cancún, Mexico, Apr. 1994. IEEE Comp. Soc.

[62] Herbert Hing-Jing Hum. *The Super-Actor Machine: a Hybrid Dataflow/von Neumann Architecture.* PhD thesis, McGill U., Montréal, Qué., May 1992.

[63] Robert A. Iannucci. A dataflow/von Neumann hybrid architecture. Tech. Rep. MIT/LCS/TR-418, MIT Lab. for Comp. Sci., Jul. 1988. PhD thesis, May 1988.

[64] Robert A. Iannucci. Toward a dataflow/von Neumann hybrid architecture. In *Proc. of the 15th Ann. Intl. Symp. on Computer Architecture*, pages 131–140, Honolulu, Haw., May–Jun. 1988.

[65] Robert A. Iannucci, Guang R. Gao, Robert H. Halstead, Jr., and Burton Smith, editors. *Multithreaded Computer Architecture: A Summary of the State of the Art.* Kluwer Academic Pub., Norwell, Mass., 1994. Book contains papers presented at the Workshop on Multithreaded Computers, Albuquerque, N. Mex., Nov. 1991.

[66] Intel Corporation. *i860 Microprocessor Family Programmer's Reference Manual.* Santa Clara, Calif., 1992. Order number 240875-002.

[67] Tetsuo Kawano, Shigeru Kusakabe, Rin ichiro Taniguchi, and Makoto Amamiya. Fine-grain multi-thread processor architecture for massively parallel processing. In *Proc. of the First Intl. Symp. on High-Performance Computer Architecture*, pages 308–317, Raleigh, North Carolina, Jan. 1995.

[68] Stephen W. Keckler, William J. Dally, Daniel Maskit, Nicholas P. Carter, Andrew Chang, and Whay S. Lee. Exploiting fine-grain thread level parallelism on the MIT multi-ALU processor. In *Proc. of the 25th Ann. Intl. Symp. on Computer Architecture*, pages 306–317, Barcelona, Spain, Jun.–Jul. 1998.

[69] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice-Hall, Inc., Englewood Cliffs, N. Jersey, 2nd edition, 1988.

[70] Yuetsu Kodama, Yasuhito Koumura, Mitsuhisa Sato, Hirohumi Sakane, Shuichi Sakai, and Yoshinori Yamaguchi. EMC-Y: Parallel processing element optimizing communication and computation. In *Conf. Proc., 1993 Intl. Conf. on Supercomputing*, pages 167–174, Tokyo, Japan, Jul. 1993.

[71] Peter M. Kogge. *The Architecture of Pipelined Computers.* McGraw-Hill Book Co., New York, N. Y., 1981.

[72] Peter M. Kogge, Steven C. Bass, Jay B. Brockman, Danny Z. Chen, and Edwin Sha. Pursuing a petaflop: Point designs for 100 TF computers using PIM technologies. In *Proc. of Frontiers '96: The Sixth Symp. on the Frontiers of Massively Parallel Computation*, pages 88–97, Annapolis, Maryland, Oct. 1996.

[73] Janusz S. Kowalik, editor. *Parallel MIMD Computation: The HEP Supercomputer and its Applications.* MIT Press, 1985.

[74] David J. Kuck, Yoichi Muraoka, and Shyh-Ching Chen. On the number of operations simultaneously executable in Fortran-like programs and their resulting speedup. *IEEE Trans. on Computers*, 21(12):1293–1310, Dec. 1972.

[75] James T. Kuehn and Burton J. Smith. The Horizon supercomputing system: Architecture and software. In *Proc. of Supercomputing '88*, pages 28–34, Orlando, Flor., Nov. 1988.

[76] Manoj Kumar. Measuring parallelism in computation-intensive scientific/engineering applications. *IEEE Trans. on Computers*, 37(9):1088–1098, Sep. 1988.

[77] Monica S. Lam and Robert P. Wilson. Limits of control flow on parallelism. In *Proc. of the 19th Ann. Intl. Symp. on Computer Architecture*, pages 46–57, Gold Coast, Australia, May 1992.

[78] Johnny K. F. Lee and Alan Jay Smith. Branch prediction strategies and branch target buffer design. *Computer*, 17(1):6–22, Jan. 1984.

[79] Bil Lewis and Daniel J. Berg. *Threads Primer — A Guide to Multithreaded Programming.* Prentice-Hall, Inc., Englewood Cliffs, N. Jersey, 1996. Copyright Sun Microsystems, Inc.

[80] Cheng Li. Efficiently supporting EARTH on a cluster of SMPs. Master's thesis, U. of Delaware, Newark, Delaware, Jun. 1999. (Expected).

[81] Jack L. Lo, Luiz André Barroso, Susan J. Eggers, Kourosh Gharachorloo, Henry M. Levy, and Sujay S. Parekh. An analysis of database workload performance on simultaneous multithreaded processors. In *Proc. of the 25th Ann. Intl. Symp. on Computer Architecture*, pages 39–50, Barcelona, Spain, Jun.–Jul. 1998.

[82] Olivier Maquelin. The ADAM architecture and its simulation. TIK-Schriftenreihe 4, Computer Engineering and Networks Laboratory, Swiss Federal Institute of Technology, Zürich, Switzerland, 1994. PhD thesis, 1994.

[83] Olivier Maquelin, Guang R. Gao, Herbert H. J. Hum, Kevin B. Theobald, and Xin-Min Tian. Polling Watchdog: Combining polling and interrupts for efficient message handling. In *Proc. of the 23rd Ann. Intl. Symp. on Computer Architecture*, pages 178–188, Philadelphia, Penn., May 1996.

[84] Olivier C. Maquelin. Load balancing and resource management in the ADAM machine. In Gao et al. [41], pages 307–323.

[85] Olivier C. Maquelin, Herbert H. J. Hum, and Guang R. Gao. Costs and benefits of multithreading with off-the-shelf RISC processors. In *Proc. of the First Intl. EURO-PAR Conf.*, number 966 in Lec. Notes in Comp. Sci., pages 117–128, Stockholm, Sweden, Aug. 1995. Springer-Verlag.

[86] Andrés Márquez, Kevin B. Theobald, Xinan Tang, and Guang R. Gao. A superstrand architecture. CAPSL Tech. Memo 14, Dept. of Elec. and Computer Eng., U. of Delaware, Newark, Delaware, Dec. 1997. In ftp://ftp.capsl.udel.edu/pub/doc/memos.

[87] Andrés Márquez, Kevin B. Theobald, Xinan Tang, and Guang R. Gao. A superstrand architecture and its compilaton. In *Proceedings of the 1999 Workshop on Multi-Threaded Execution, Architecture and Compilation*, Orlando, Flor., Jan. 1999. Held in conjunction with the Fifth Intl. Symp. on High-Performance Computer Architecture.

[88] W. Najjar and J.-L. Gaudiot. Multi-level execution in data-flow architectures. In *Proc. of the 1987 Intl. Conf. on Parallel Processing*, pages 32–39, St. Charles, Ill., Aug. 1987.

[89] Walid A. Najjar, William Marcus Miller, and A. P. Wim Böhm. Locality and latency in hybrid dataflow. In Gao et al. [41], pages 417–434.

[90] Alexandru Nicolau and J. A. Fisher. Measuring the parallelism available for very long instruction word architectures. *IEEE Trans. on Computers*, 33(11):968–976, Nov. 1984.

[91] R. S. Nikhil, G. M. Papadopoulos, and Arvind. *T: A multithreaded massively parallel architecture. In *Proc. of the 19th Ann. Intl. Symp. on Computer Architecture*, pages 156–167, Gold Coast, Australia, May 1992.

[92] Rishiyur S. Nikhil and Arvind. Can dataflow subsume von Neumann computing? In *Proc. of the 16th Ann. Intl. Symp. on Computer Architecture*, pages 262–272, Jerusalem, Israel, May–Jun. 1989.

[93] Rishiyur S. Nikhil and Arvind. Id: a language with implicit parallelism. CSG Memo 305, Computation Structures Group, MIT Lab. for Comp. Sci., Feb. 1990.

[94] Rishiyur Sivaswami Nikhil. The parallel programming language Id and its compilation for parallel machines. *Intl. J. of High Speed Computing*, 5(2):171–223, 1993.

[95] Michael D. Noakes, Deborah A. Wallah, and William J. Dally. The J-Machine multicomputer: An architectural evaluation. In *Proc. of the 20th Ann. Intl. Symp. on Computer Architecture*, pages 224–235, San Diego, Calif., May 1993.

[96] Scott Oaks and Henry Wong. *Java Threads*. O'Reilly & Associates, Inc., Cambridge, 1997.

[97] Shien-Tai Pan, Kimming So, and Joseph T. Rahmeh. Improving the accuracy of dynamic branch prediction using branch correlation. In *Proc. of the Fifth Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 76–84, Boston, Mass., Oct. 1992.

[98] Gregory Michael Papadopoulos. Implementation of a general purpose dataflow multiprocessor. Tech. Rep. MIT/LCS/TR-432, MIT Lab. for Comp. Sci., Aug. 1988. PhD thesis.

[99] James Rumbaugh. A data flow multiprocessor. *IEEE Trans. on Computers*, 26(2):138–146, Feb. 1977.

[100] James Edward Rumbaugh. A parallel asynchronous computer architecture for data flow programs. Tech. Rep. MIT/LCS/TR-150, MIT Lab. for Comp. Sci., May 1975. PhD thesis.

[101] Shuichi Sakai, Kazuaki Okamoto, Hiroshi Matsuoka, Hideo Hirono, Yuetsu Kodama, and Mitsuhisa Sato. Super-threading: Architectural and software mechanisms for optimizing parallel computation. In *Conf. Proc., 1993 Intl. Conf. on Supercomputing*, pages 251–260, Tokyo, Japan, Jul. 1993.

[102] Mitsuhisa Sato, Yuetsu Kodama, Suichi Sakai, Yoshinori Yamaguchi, and Yasuhito Koumura. Thread-based programming for the EM-4 hybrid dataflow machine. In *Proc. of the 19th Ann. Intl. Symp. on Computer Architecture*, pages 146–155, Gold Coast, Australia, May 1992.

[103] Burton Smith. The architecture of HEP. In Kowalik [73], pages 41–55.

[104] Burton Smith. The end of architecture. *Computer Arch. News*, 18(4):10–17, Dec. 1991. Keynote address at the 17th Ann. Intl. Symp. on Computer Architecture, Seattle, Wash., May 29, 1990.

[105] James E. Smith. A study of branch prediction strategies. In *Proc. of the 8th Ann. Symp. on Computer Architecture*, pages 135–148, Minneapolis, Minn., May 1981.

[106] Michael D. Smith, Monica S. Lam, and Mark A. Horowitz. Boosting beyond static scheduling in a superscalar processor. In *Proc. of the 17th Ann. Intl. Symp. on Computer Architecture*, pages 344–354, Seattle, Wash., May 1990.

[107] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proc. of the 22nd Ann. Intl. Symp. on Computer Architecture*, pages 414–425, Santa Margherita Ligure, Italy, Jun. 1995.

[108] Ellen Spertus, Seth Copen Goldstein, Klaus Erik Schauser, Thorsten von Eicken, David E. Culler, and William J. Dally. Evaluation of mechanisms for fine-grained parallel programs in the J-Machine and the CM-5. In *Proc. of the 20th Ann. Intl. Symp. on Computer Architecture*, pages 302–313, San Diego, Calif., May 1993.

[109] Vason P. Srini. An architectural comparison of dataflow systems. *Computer*, 19(3):68–88, Mar. 1986.

[110] Harold S. Stone. *High-Performance Computer Architecture.* Addison-Wesley Pub. Co., 2nd edition, 1990.

[111] Xinan Tang, Jian Wang, Kevin B. Theobald, and Guang R. Gao. Thread partitioning and scheduling based on cost model. In *Proc. of the 9th Ann. ACM Symp. on Parallel Algorithms and Architectures*, pages 272–281, Newport, Rhode Island, Jun. 1997.

[112] Maria-Dana Tarlescu, Kevin B. Theobald, and Guang R. Gao. Elastic history buffer: A low-cost method to improve branch prediction accuracy. In *Proc. of the 1997 Intl. IEEE Conf. on Computer Design*, pages 82–87, Austin, Tex., Oct. 1997.

[113] Scott R. Taylor. A comparison of multithreading implementations. In *Proc. of the Yale Multithreaded Programming Work.*, New Haven, Conn., Jun. 1998.

[114] Kevin B. Theobald. Panel sessions of The 1991 Workshop on Multithreaded Computers. *Computer Arch. News*, 22(1):2–33, Mar. 1994. Workshop held at *Supercomputing '91,*Albuquerque, N. Mex., Nov. 1991.

[115] Kevin B. Theobald. SEMi: A simulator for EARTH, MANNA, and the i860 (version 0.15). ACAPS Tech. Note 43 (Revised), Sch. of Comp. Sci., McGill U., Montréal, Qué., Oct. 1996.

[116] Kevin B. Theobald, Guang R. Gao, and Laurie J. Hendren. On the limits of program parallelism and its smoothability. In *Proc. of the 25th Ann. Intl. Symp. on Microarchitecture*, pages 10–19, Portland, Ore., Dec. 1992.

[117] Kevin B. Theobald, Guang R. Gao, and Laurie J. Hendren. On the limits of program parallelism and its smoothability. ACAPS Tech. Memo 40, Sch. of Comp. Sci., McGill U., Montréal, Qué., Jun. 1992. In ftp://ftp-acaps.cs.mcgill.ca/pub/doc/memos.

[118] Kevin B. Theobald, Guang R. Gao, and Laurie J. Hendren. Speculative execution and branch prediction on parallel machines. ACAPS Tech. Memo 57, Sch. of Comp. Sci., McGill U., Montréal, Qué., Dec. 1992. In ftp://ftp-acaps.cs.mcgill.ca/pub/doc/memos.

[119] Kevin B. Theobald, Guang R. Gao, and Laurie J. Hendren. Speculative execution and branch prediction on parallel machines. In *Conf. Proc., 1993 Intl. Conf. on Supercomputing*, pages 77–86, Tokyo, Japan, Jul. 1993.

[120] Kevin B. Theobald, Guang R. Gao, and Laurie J. Hendren. The effects of resource limitations on program parallelism. In Gao et al. [41], pages 367–392.

[121] Kevin B. Theobald, Herbert H. J. Hum, and Guang R. Gao. A design framework for hybrid-access caches. In *Proc. of the First Intl. Symp. on High-Performance Computer Architecture*, pages 144–153, Raleigh, North Carolina, Jan. 1995.

[122] Kevin Bryan Theobald. Adding fault-tolerance to a static data flow supercomputer. Tech. Rep. MIT/LCS/TR-499, MIT Lab. for Comp. Sci., Apr. 1991. Master's thesis, Dec., 1990.

[123] Garold S. Tjaden and Michael J. Flynn. Detection and parallel execution of independent instructions. *IEEE Trans. on Computers*, 19(10):889–895, Oct. 1970.

[124] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM J. of Res. & Dev.*, 11(1):25–33, Jan. 1967.

[125] Jean-Yuan Tsai and Pen-Chung Yew. The superthreaded architecture: Thread pipelining with run-time data dependence checking and control speculation. In *Proc. of the 1996 Conf. on Parallel Architectures and Compilation Techniques (PACT '96)*, pages 35–46, Boston, Mass., Oct. 1996.

[126] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proc. of the 22nd Ann. Intl. Symp. on Computer Architecture*, pages 392–403, Santa Margherita Ligure, Italy, Jun. 1995.

[127] L. G. Valiant. A bridging model for parallel computation. *Comm. of the ACM*, 33(8):103–111, Aug. 1990.

[128] David W. Wall. Limits of instruction-level parallelism. In *Proc. of the Fourth Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 176–188, Santa Clara, Calif., Apr. 1991.

Appendix A

Previous Studies of Parallelism

Most previous studies on the limits of program parallelism have taken one of two approaches. One approach is to analyze a benchmark at either the source-code or object-code level, usually with a special interpreter that is based on a specific parallel machine model. The other approach, used in Chapter 2, is to schedule machine instructions from a trace generated by an actual run of the executable code.

An early experimental tool, designed by Kuck, Muraoka and Chen [74], compiled small Fortran programs to run on an abstract parallel machine. By recognizing parallelism at the source code level (e.g., parallel iterations of DO-loops) and by aggressively reordering complex expressions, they were able to speed up most programs by a factor of 2–7, and speed up some by as much as 25. Tjaden and Flynn built a simulator based on the superscalar model [123]. Their simulator could execute ordinary IBM 7094 machine code, but attempted to reschedule operations dynamically using a look-ahead window of between 2 and 10 operations. They obtained speedups of up to 3 on a suite of 31 library routines. Both studies got modest results because they only looked for parallelism within small blocks, and could not execute independent instructions from separate blocks. These blocks are typically separated by conditional branches, and if the decision whether to execute one block is controlled by the outcome of a conditional branch in another block, the first block can't begin execution until the branch has been resolved.

Nicolau and Fisher wrote an interpreter for intermediate code generated by a compiler front-end, and a tool to analyze the instruction trace generated by the interpreter [90]. They addressed the shortcomings of the previous studies by allowing

operations from many parts of the program to execute concurrently. They introduced the oracle model, which they added to their analysis tool. The analyzer also ignored any dependence which was not a "true" data dependence. They were able to obtain respectable speedups from small scientific routines.

Kumar [76] developed a tool that automatically added statistics-gathering statements to Fortran code. By running the modified programs, he was able to measure parallelism in scientific applications. The tool also performed control-dependence analysis, so that independent statements separated by barriers could still run concurrently. Kumar found the potential for high levels of parallelism in regular numerical code.

Wall [128] analyzed benchmarks (SPEC89 programs, system utilities, and toy benchmarks) in which the number of executed operations ranged from 1 million to 2 billion. He developed a series of models representing various levels of optimism about what the hardware and compiler could do. He then ran the benchmarks on a MIPS R3000 processor, and analyzed the instruction traces in a manner similar to Tjaden and Flynn, and Nicolau and Fisher. Even under his "perfect" model, with oracle-like branch prediction, parallelism rarely exceeded 40. This is because his scheduler used only 64 processors, a finite (2K) input window from which to fill parallel instructions, and no memory renaming (register renaming only).

Two recent studies measured the effects of limiting the size of the scheduling window, which determines how broad the search for parallelism in the sequential trace can be, on the parallelism available in SPEC89 benchmarks. Butler, et al [18], ran each trace for 10 million instructions on an M88000 processor, while Austin and Sohi [12] ran each trace for up to 100 million instructions on a MIPS processor. Both assumed perfect (oracular) branch prediction, but limited how far apart two instructions could be in the sequential trace and still be scheduled in the same parallel instruction. They both concluded that instructions must be drawn from regions far apart in order to achieve significant amounts of parallelism.

Wilson and Lam studied the way control flow limits parallelism [77]. They demonstrated that substantially higher parallelism can be achieved by relaxing the constraints imposed by control flow on parallelism using control-dependence analysis, executing multiple flows of control simultaneously, and performing speculative execution. They tested six SPEC89 programs and 4 other programs, running each program on a MIPS R3000 for up to 100 million instructions.

Appendix B

Definition of Threaded-C

This appendix gives a complete list of all the EARTH Threaded-C operations and briefly explains how they are used.

B.1 Fibers and Procedures

THREADED
Keyword for a threaded procedure declaration.

THREAD_n
Marks the beginning of a fiber. The fiber label within a procedure, n, is an integer greater than 0.

void END_THREAD(void)
Marks the end of a fiber. When an END_THREAD is found, the control switches to another ready fiber.

int NUM_NODES
Run-time system variable set to the number of nodes available in the system.

int NODE_ID
Run-time system variable set to the number of the local node. This number ranges from 0 to NUM_NODES - 1.

void POLL(void)

Polls the network and handles any available messages. Along with the NUM_NODES and NODE_ID constants, this is one of the only Threaded-C components that can be used in non-threaded functions. Inserting POLL statements into long fibers can significantly improve overall performance because it allows for the faster handling of external requests.

void CALL(func_name, ...)

Calls function *func_name* sequentially and blocks until that function terminates. Functions invoked with CALL must terminate with RETURN instead of END_FUNCTION.

void RETURN(void)

Ends a procedure that is called with the CALL operation. This tells the compiler to generate sequential entry/exit code.

void INVOKE(int node, func_name, ...)

Starts the execution of **func_name** on the node specified by the programmer without blocking. Procedure *func_name* must terminate with END_FUNCTION.

void TOKEN(func_name, ...)

Starts the execution of **func_name** on a node selected by the runtime system. The node is selected trying to optimize the distribution of the workload in the machine. Procedure *func_name* must terminate with END_FUNCTION.

void END_FUNCTION(void)

Marks the end of a threaded procedure that is called with INVOKE or TOKEN.

B.2 Fiber Synchronization

Fibers are often associated with a synchronization slot. The sync count in that slot represents how many signals the fiber has to wait for before it can be activated. The programmer can initialize the sync count and update the count to control the firing of a fiber. We use the following EARTH operations to operate on the sync

slots:

SLOT

A pre-defined type used to allocate the synchronization slots that will be used in the procedure.

SLOT SYNC_SLOTS[N]

This is the declaration of synchronization slots using the pre-defined type SLOT. This declaration must appear at the beginning of a procedure.

SPTR

A pre-defined type for global sync slot addresses. It is defined as: **typedef SLOT *GLOBAL SPTR.**

void *GLOBAL FRAME_ADR(void)

Returns the global address of the current frame.

void *IP_ADR(int fiber_num)

Returns a (local) pointer to the first instruction of fiber *fiber_num*. Notice that the pointer returned does not need to be a global address because each node has a copy of the program code loaded at the same address.

SPTR SLOT_ADR(int slot_num)

Returns the global address of sync slot *slot_num*.

void INIT_SYNC(int slot_num, int init_cnt, int reset_cnt, int fiber_num)

Initializes sync slot *slot_num* with the initial counter value *init_cnt*, the reset value *reset_cnt*, and the ip corresponding to fiber *fiber_num*.

void SYNC(int slot_num)

Decreases the sync count of slot *slot_num* by one. If the count reaches zero the corresponding fiber is scheduled for execution.

void RSYNC(SPTR slot_adr)

Same as SYNC(), but the sync slot is specified by a global address.

void INCR_SYNC(int slot_num, int val)

Increases the sync count of slot *slot_num* by *val*. If the count becomes zero the corresponding fiber is scheduled for execution.

void INCR_RSYNC(SPTR slot_adr, int val)

Same as INCR_SYNC(), but the sync slot is specified by a global address.

void SPAWN(int fiber_num)

Schedules local fiber *fiber_num* for execution.

void RSPAWN(void *GLOBAL FP, void *IP)

Schedules local fiber *fiber_num* for execution. The fiber is specified through its frame and instruction pointers.

Implicit sync operation

All data transfer operations also perform a sync operation after the data has reached its destination.

B.3 Data Transfer Operations

The data transfer operations support remote memory accesses and block data transfers. Short data transfers of single bytes or words of memory are supported by the GET_SYNC_x and DATA_SYNC_x. Several versions of these operations exist, which are distinguished by their suffix. For example, the suffix _L is used for 32-bit (long word) values. Here is the complete list of suffixes:

_B (char): Single byte (8 bits).

_S (short): Short word (16 bits).

_L (long): Long word (32 bits).

_F (float): Float size (32 bits).

_D (double): Double size (64 bits).

_G (void *GLOBAL): Global address (either 32 or 64 bits, depending on implementation).

In addition, the sync slot that should be signaled when the operation terminates can either be specified as a (local) slot number (_SYNC_ variants), or as a global address (_RSYNC_ variants). Here are the basic communication operations:

void DATA_SYNC_x(T datum, void *GLOBAL dest, int slot_num)

Sends a value to the destination address and then update the specified sync slot. The type of the value has to be either a byte, a short, a long, a float, a double, or a global address.

void DATA_RSYNC_x(T datum, void *GLOBAL dest, SPTR slot_adr)

Same as DATA_SYNC_x(), but the sync slot is specified as a global address.

void GET_SYNC_x(void *GLOBAL src, void *GLOBAL dest, int slot_num)

Reads a value from the source address and copies it to the destination address, then updates the specified sync slot.

void GET_RSYNC_x(void *GLOBAL src, void *GLOBAL dest, SPTR slot_adr)

Reads a value from the source address and copies it to the destination address, then updates the specified sync slot. The sync slot is specified as a global address.

void BLKMOV_SYNC(void *GLOBAL src, void *GLOBAL dest, long length, int slot_num)

Copies *length* bytes of data from the source to the destination address and updates the specified sync slot.

void BLKMOV_RSYNC(void *GLOBAL src, void *GLOBAL dest, long length, SPTR slot_adr)

Same as BLKMOV_SYNC(), but the sync slot is specified as a global address.

B.4 Global Address Support

GLOBAL

Type qualifier used to distinguish global addresses (64-bit entities) from local (normal) pointers.

T *GLOBAL TO_GLOBAL(T *ptr)

Turns a local pointer into a global address that points to address **ptr** on the local node. In the portable implementation the type of the result depends on the type of the argument. On MANNA, the result is of type pointer to void.

T *TO_LOCAL(T *GLOBAL gptr)

Turns a global address into a local pointer (extracts the address part of a global address). Note that it is possible to dereference a global address without first turning it into a local pointer. On MANNA, the result is of type pointer to void.

T *GLOBAL MAKE_GPTR(T *ptr, int node)

Takes a node number and a local address and returns the corresponding global address. On MANNA, the result is of type pointer to void.

int OWNER_OF(T *GLOBAL gptr)

Returns the node pointed to by *gptr* (extracts the node part of a global address).

int IS_OWNER(T *GLOBAL gptr)

Returns *true* if *gptr* points to the local node.

B.5 Differences Between Threaded-C and ANSI C

The programmer should be aware that some standard C features are not supported in the current version of Threaded-C. Many of these features are related to the *static* and *extern* keywords. This section lists the unsupported features.

- A threaded procedure cannot be static.

```
THREADED foo ()
{
    static int i = 1; /* not allowed */

    int x = 0;     /* not allowed */

}
```

(a) Static and Initialized Variables

```
THREADED foo (int y); /* NO */

THREADED foo (int x); /* Yes */

THREADED foo(int x)
{
  ...
}
```

(b) Prototypes

Figure B.1: Examples of Illegal Use

- In the declaration of a threaded procedure, the **extern** specifier must be omitted.
- When defined within the scope of a threaded procedure, a static variable cannot be initialized. For example, in Figure B.1(a), the initialization of static variable `i` is illegal.
- A threaded procedure is not allowed to initialize variables in their definitions. As shown in Figure B.1(a), initializing variable **x** is forbidden.
- Forward declarations of threaded procedures (prototypes) are supported, but in a limited fashion. If the programmer wants an ANSI prototype declaration, the parameter names must be the same as the original procedure declarations. For example, in Figure B.1(b), using a different parameter name **y** is not allowed. However, using the same parameter name **x** is OK.

In the current version, the Threaded-C compiler does not check for the "illegal" usages above. It is up to the programmer to avoid using them.

Appendix C

Summary of the Experiments

The following figures (C.1–C.17) show the absolute speedups achieved with EARTH-MANNA-D, EARTH-MANNA-S, and the two hardware-SU platforms, for the five benchmarks fully studied. Unlike the graphs in Chapters 7 and 8, these graphs place all experiments corresponding to one benchmark and input in a single graph, allowing an easy comparison of the different implementations.

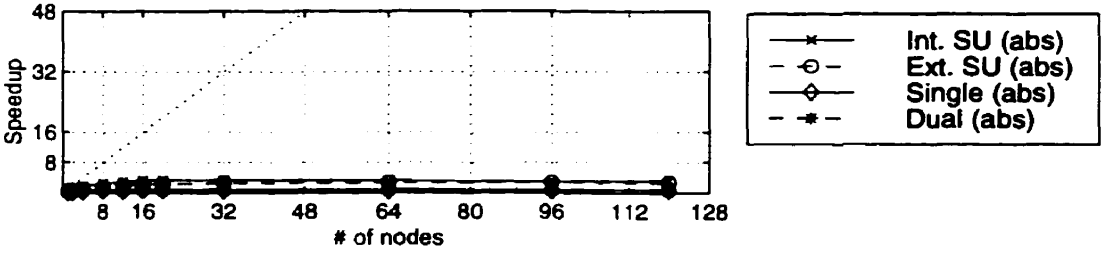


Figure C.1: Absolute Speedups for Fibonacci (15)

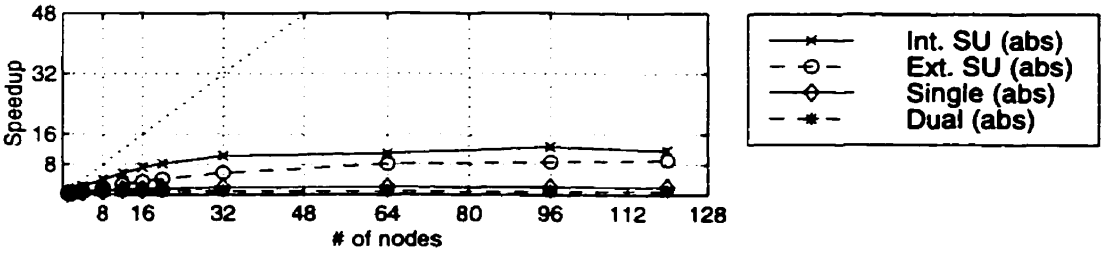


Figure C.2: Absolute Speedups for Fibonacci (20)

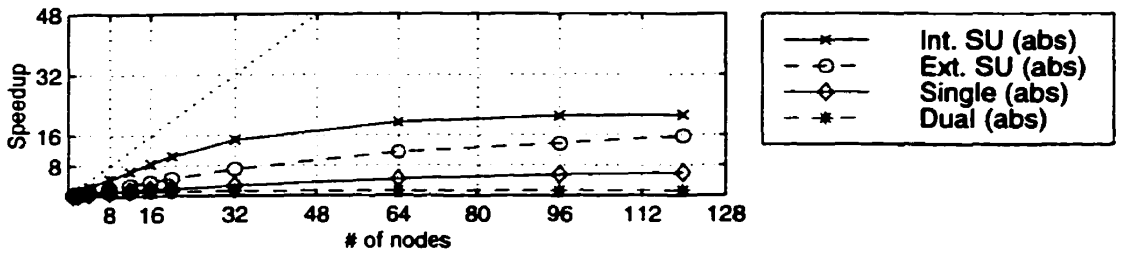


Figure C.3: Absolute Speedups for Fibonacci (25)

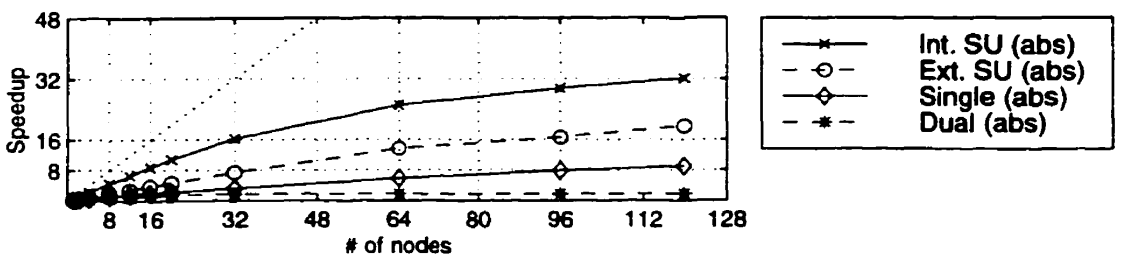


Figure C.4: Absolute Speedups for Fibonacci (30)

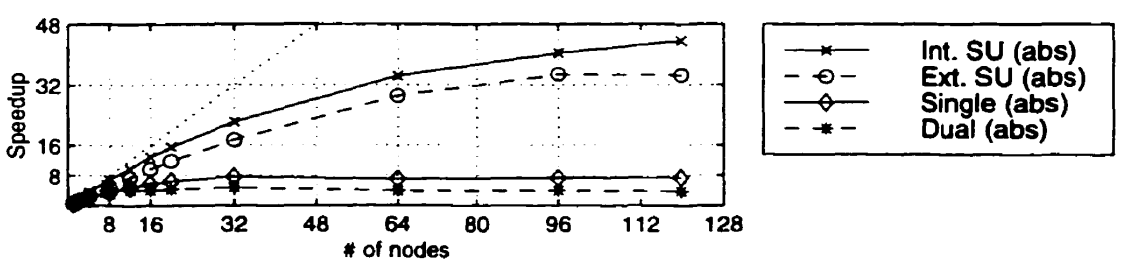


Figure C.5: Absolute Speedups for N-Queens-P (8)

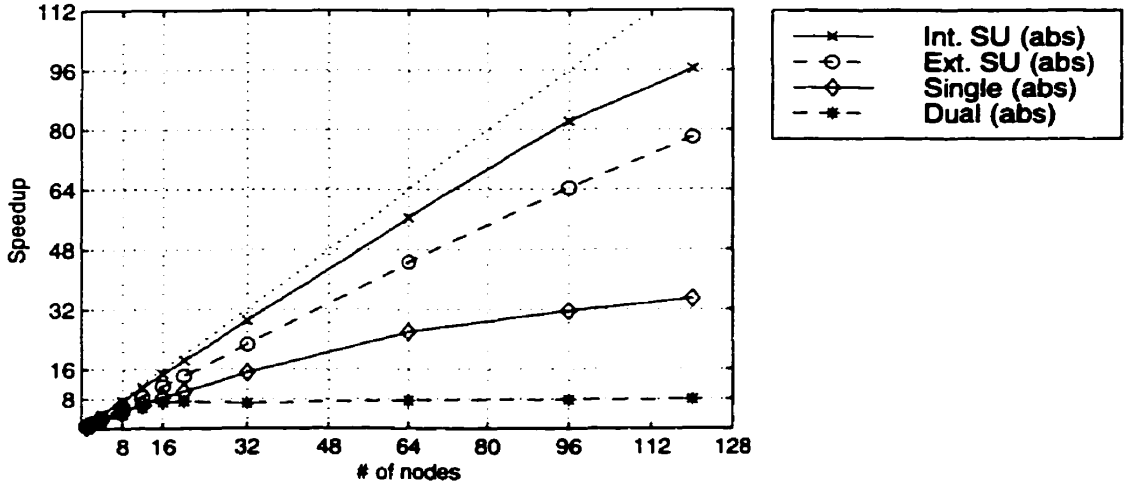


Figure C.6: Absolute Speedups for N-Queens-P (10)

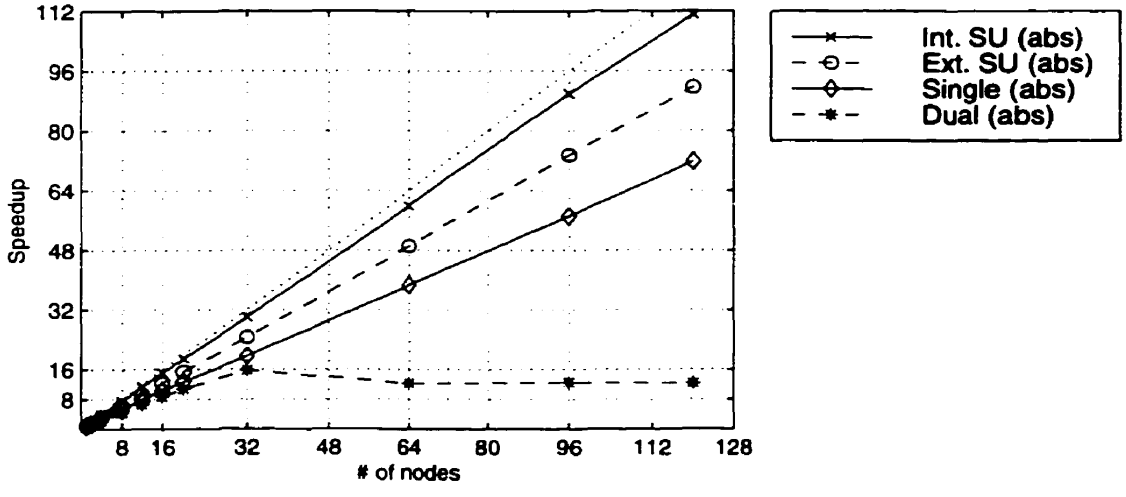


Figure C.7: Absolute Speedups for N-Queens-P (12)

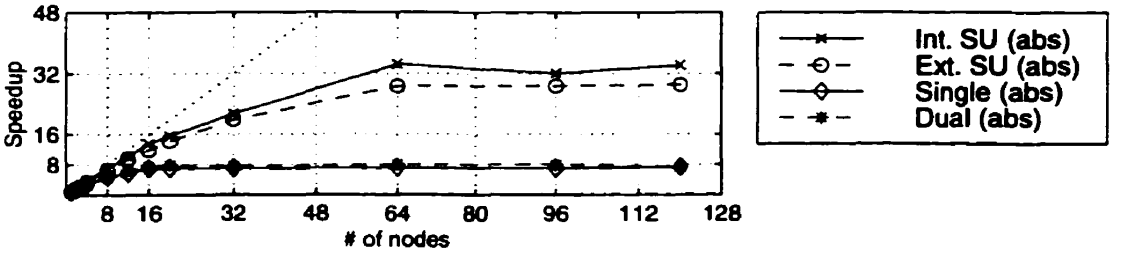


Figure C.8: Absolute Speedups for N-Queens-T (8)

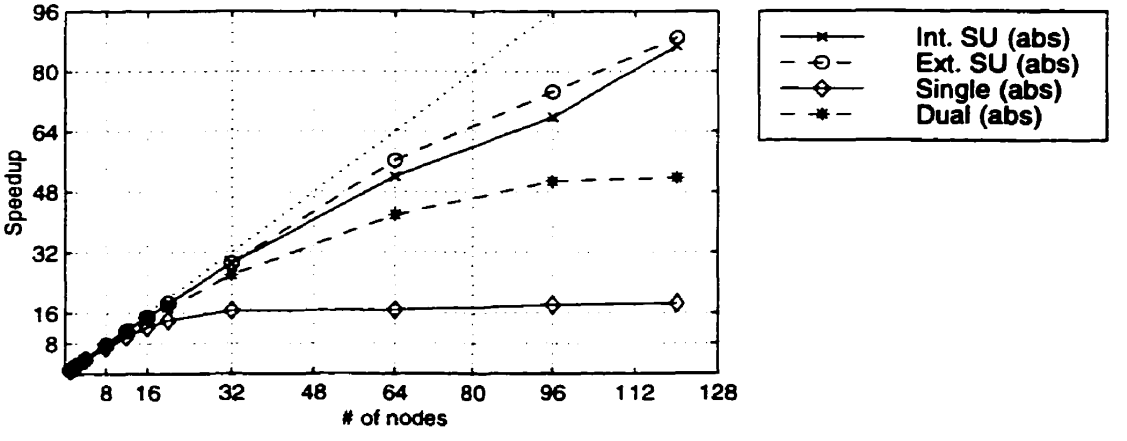


Figure C.9: Absolute Speedups for N-Queens-T (10)

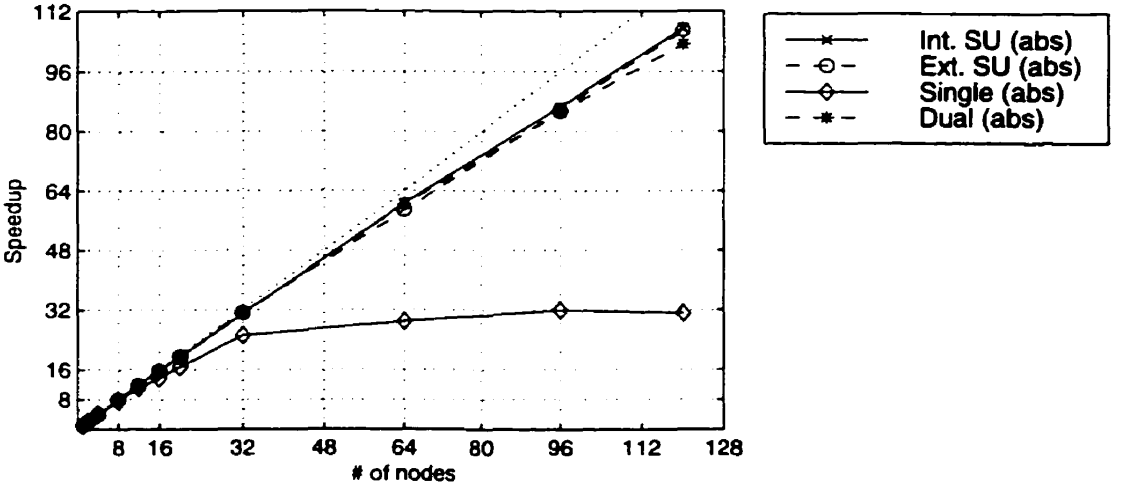


Figure C.10: Absolute Speedups for N-Queens-T (12)

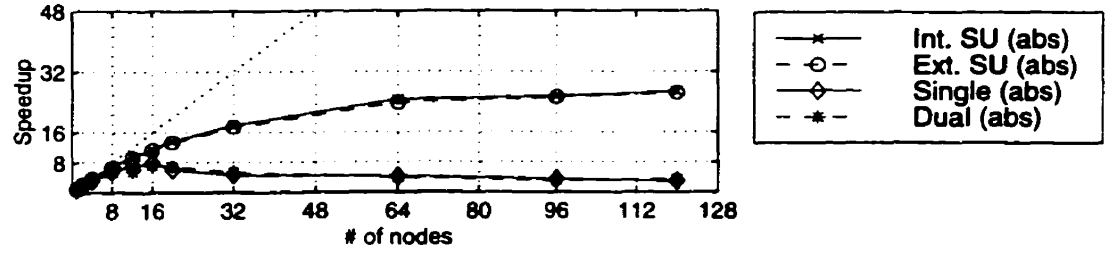


Figure C.11: Absolute Speedups for Paraffins (18)

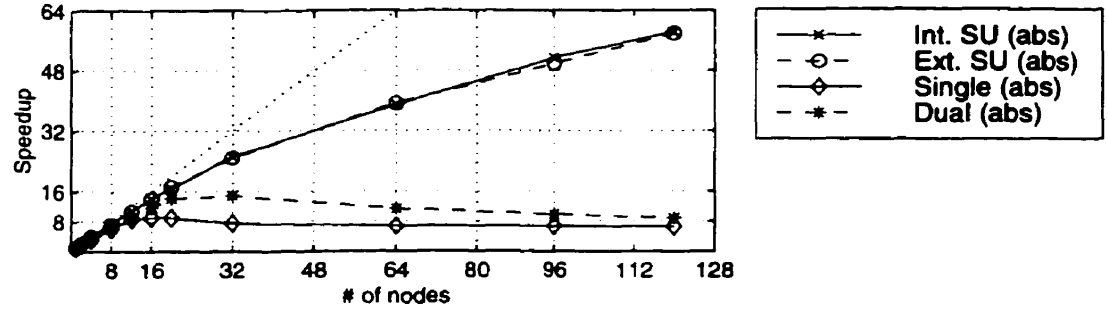


Figure C.12: Absolute Speedups for Paraffins (20)

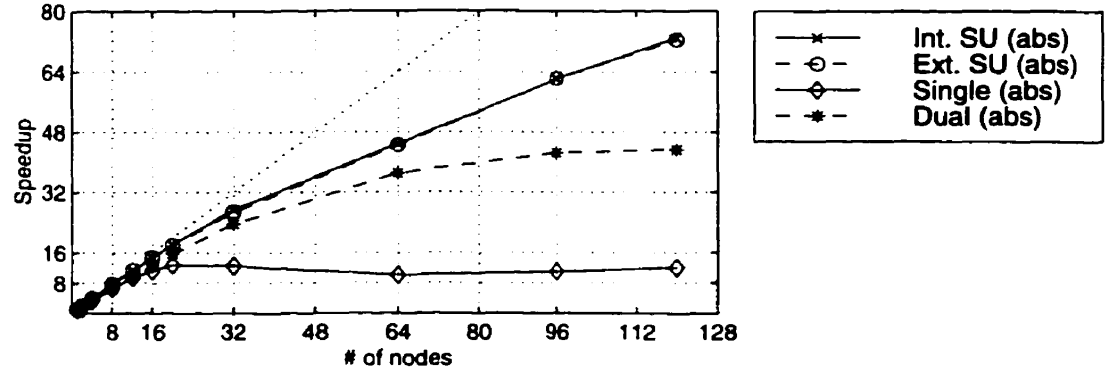


Figure C.13: Absolute Speedups for Paraffins (23)

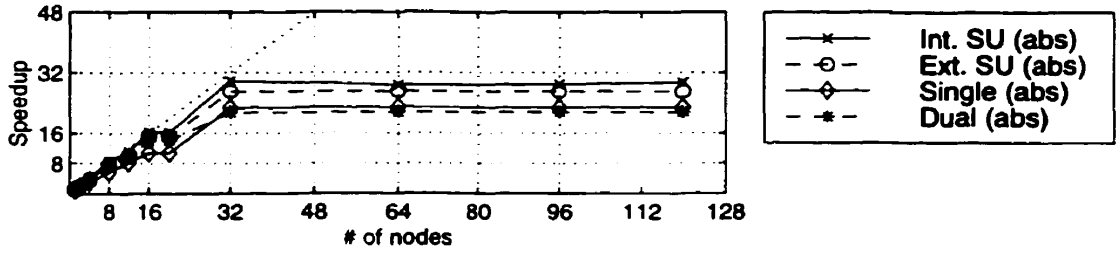


Figure C.14: Absolute Speedups for Tomcatv (33)

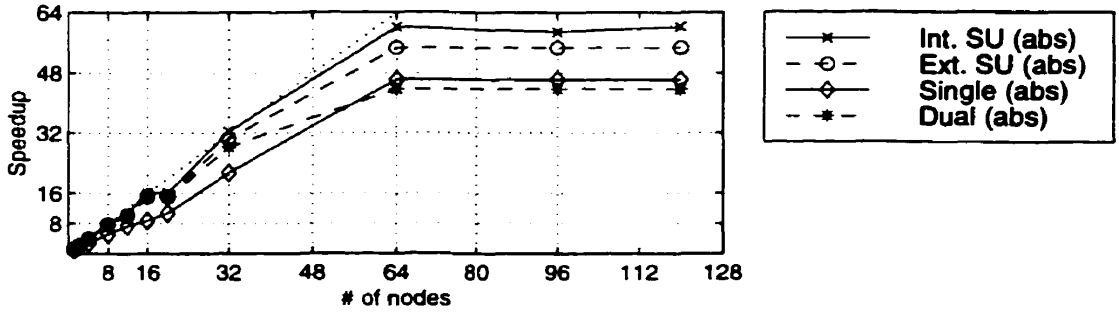


Figure C.15: Absolute Speedups for Tomcatv (65)

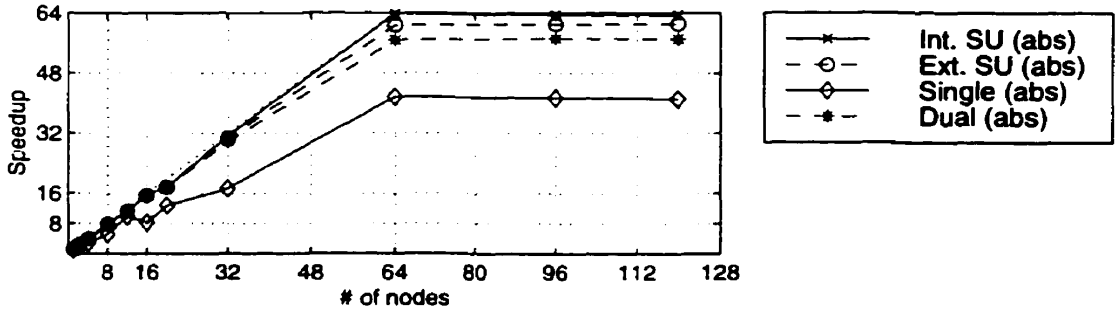


Figure C.16: Absolute Speedups for Tomcatv (129)

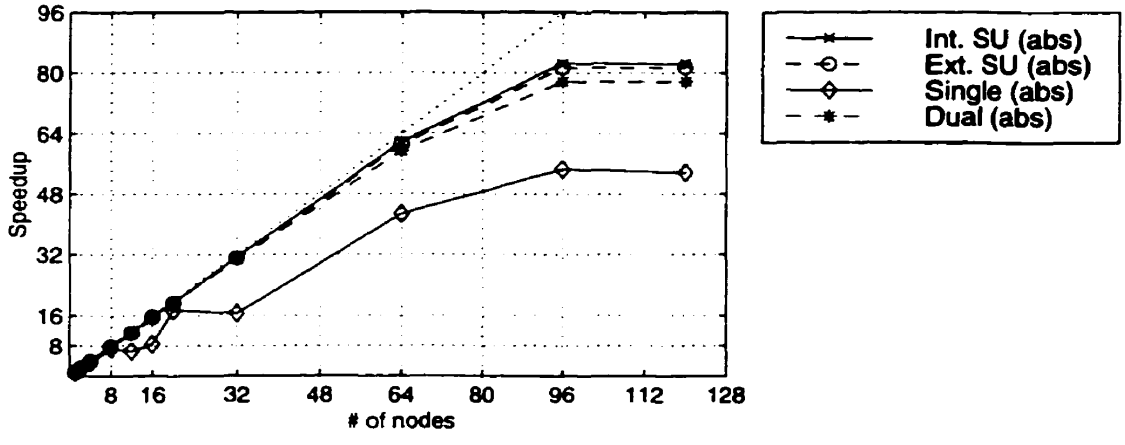


Figure C.17: Absolute Speedups for Tomcatv (257)

Appendix D

Performance of EARTH-MANNA Systems with Updated Hardware

This appendix presents the data from the enhanced-CPU experiments, first described in Section 7.2.1. In these experiments, experiments from Chapters 7 and 8 (recapitulated in the previous appendix) are repeated on the SEMi simulator with different parameters for the processor and memory system. These parameters are listed, along with the original parameters of the i860XP and MANNA, in Table D.1. The parameters for the faster "MANNA" are mostly taken from the new PowerMANNA, which is based on the PowerPC 620 processor. Other differences are discussed in Section 7.2.1.

The sequential running times and USE factors for all benchmarks and inputs on the four simulated systems are listed in Table D.2. The sequential running times are computed assuming a 200MHz processor. To see how the modifications affect the instruction issue rate of the i860, the T_{seq} times should be multipled by 4 and compared with the T_{seq} times in Table 7.6.

The remaining figures (D.1–D.17) show the absolute speedups for the seventeen benchmark/input combinations. As in the previous appendix, they are grouped by benchmark and input, so that different implementations can be compared.

While some differences in performance can be seen between the two versions of the machine, there is no definite trend; sometimes the speedups are better on the faster machine, and sometimes they are worse. Speedups drop considerably for Fibonacci. This is because the multiple-issue capabilities of the modified processor work very well on the sequential code (it is more than 50% faster on the modified

Module	Parameter	Original MANNA	Faster "MANNA"
CPU	Clock speed (MHz)	50	200
	IPC	single or dual (explicit)	multiple (in order)
	FP reservation table	no	yes
	FP load stalls CPU	yes	no
L1 cache (I,D)	Size (Kbytes)	16	32
	Line size (bytes)	32	32
	Set associativity	4	8
	Hit read time (CPU cycles)	1	1
	Access	blocking	non-blocking
Bus	Clock speed (MHz)	50	66.7
L2 cache (unified)	Size (Mbytes)	N/A	1
	Line size (bytes)	N/A	32
	Set associativity	N/A	1
	Hit read time (CPU cycles)	N/A	6
Memory	Miss read time (CPU cycles)	8	20

Table D.1: Parameters of Original and Modified MANNA

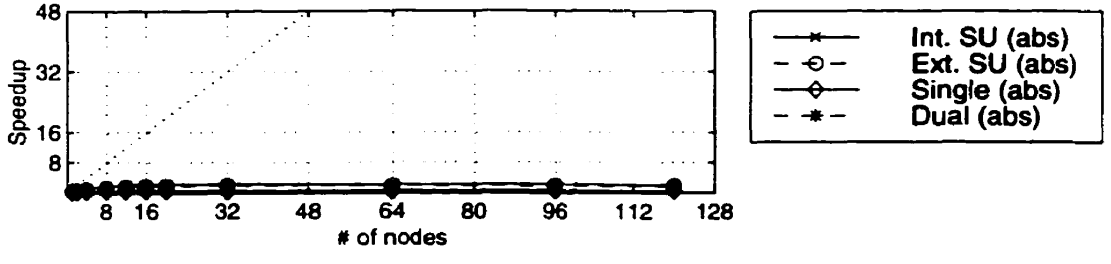


Figure D.1: Absolute Speedups on Fast EARTH-MANNA for Fibonacci (15)

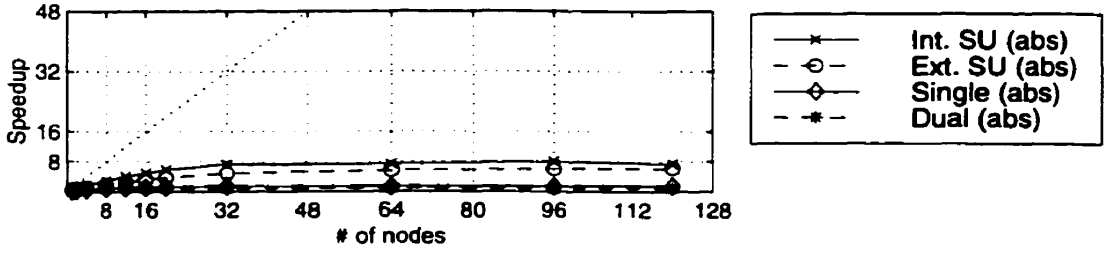


Figure D.2: Absolute Speedups on Fast EARTH-MANNA for Fibonacci (20)

Benchmark	Input	T_{seq} (sec)	USE factor (%)			
			Dual-proc.	Single-proc.	Ext. SU	Int. SU
Fibonacci	15	0.000126	11.3	16.4	26.5	45.3
	20	0.00112	9.8	13.8	23.2	41.2
	25	0.0122	9.6	13.5	22.8	40.8
	30	0.135	9.6	13.5	22.8	40.8
N-Queens-P	8	0.00347	42.8	53.0	63.1	82.0
	10	0.0887	51.1	58.7	70.8	87.3
	12	2.94	59.1	68.9	76.3	90.4
N-Queens-T	8	0.00347	70.5	79.0	83.9	91.7
	10	0.0887	93.7	95.2	96.6	97.9
	12	2.94	98.8	98.9	99.0	99.1
Paraffins	18	0.0182	80.5	97.5	98.3	98.7
	20	0.108	81.1	98.5	98.7	98.8
	23	1.82	81.9	99.4	99.4	99.4
Tomcatv	33	0.117	80.1	83.4	90.4	94.3
	65	0.479	64.3	71.6	92.6	94.3
	129	1.96	44.7	56.3	92.0	92.8
	257	8.45	27.6	42.4	97.1	97.4

Table D.2: Uni-Node Support Efficiencies on SEMi Simulation of Faster EARTH-MANNA

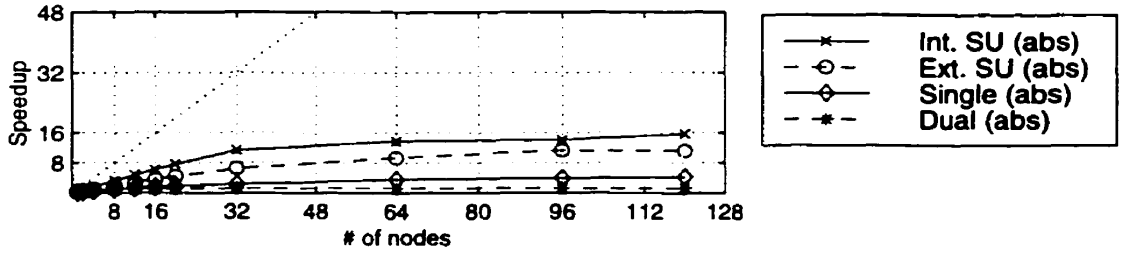


Figure D.3: Absolute Speedups on Fast EARTH-MANNA for Fibonacci (25)

processor in terms of cycles), while the multithreaded codes require frequent bus transfers, which we assume cannot be run in parallel. (Fully integrating the SU with the EU, as described in Section 8.3, may improve the issue rate on the faster CPU.)

Some anomolous results can also be seen with the last two benchmarks. With Paraffins, the external-SU speedups for the largest problem size become much worse,

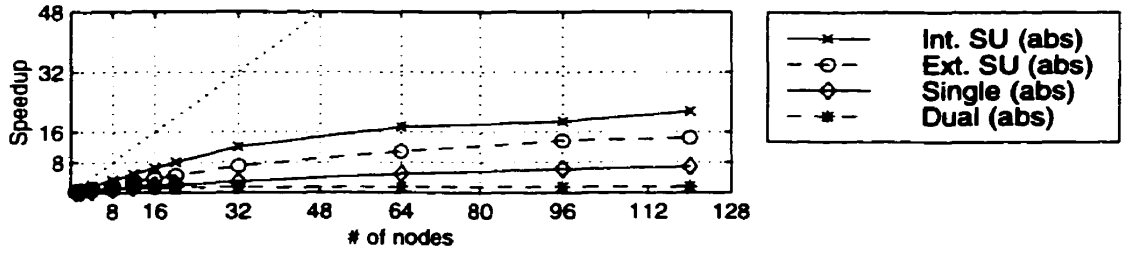


Figure D.4: Absolute Speedups on Fast EARTH-MANNA for Fibonacci (30)

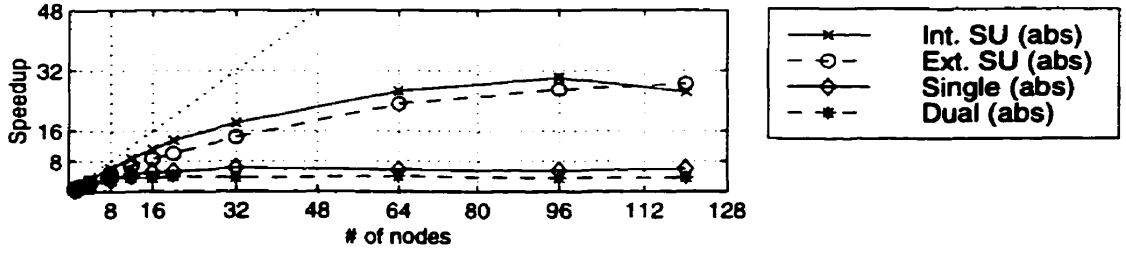


Figure D.5: Absolute Speedups on Fast EARTH-MANNA for N-Queens-P (8)

yet the internal-SU speedups become much better, where both the absolute and relative speedup become almost linear (compare Figures C.13 and D.13). The behavior

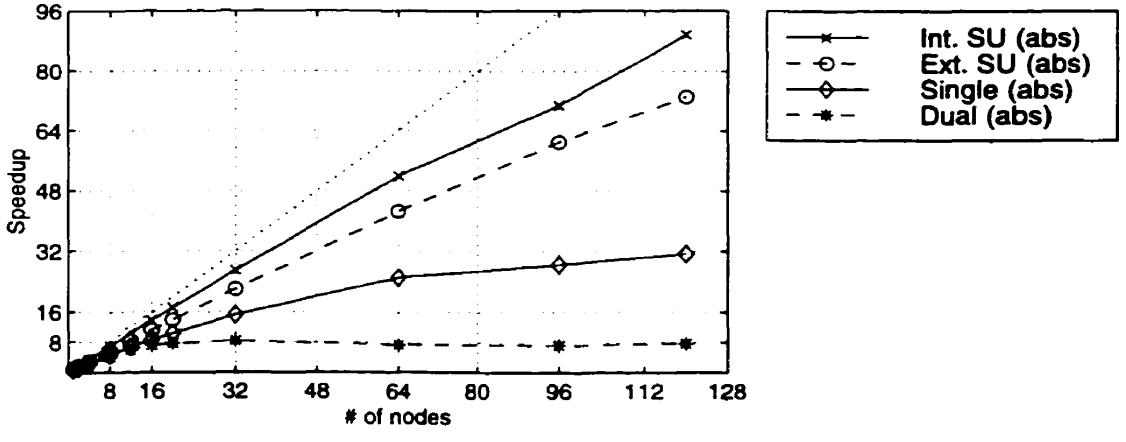


Figure D.6: Absolute Speedups on Fast EARTH-MANNA for N-Queens-P (10)

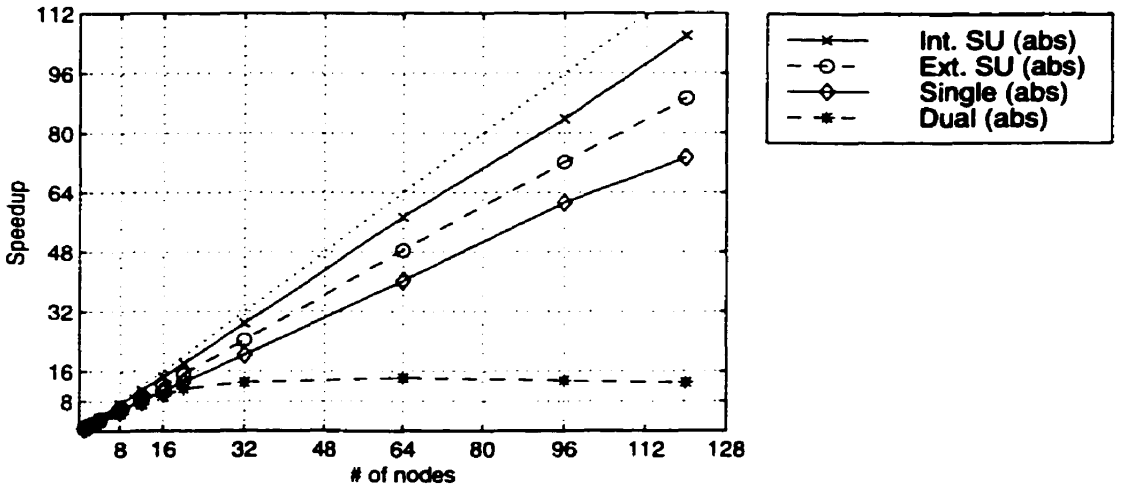


Figure D.7: Absolute Speedups on Fast EARTH-MANNA for N-Queens-P (12)

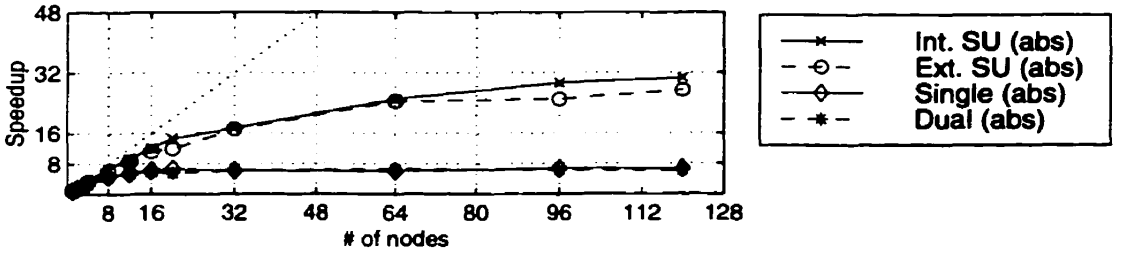


Figure D.8: Absolute Speedups on Fast EARTH-MANNA for N-Queens-T (8)

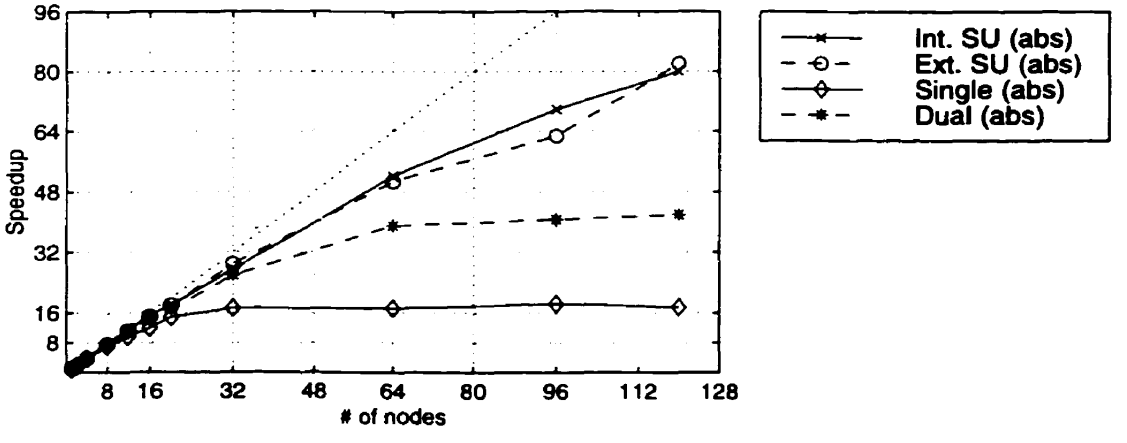


Figure D.9: Absolute Speedups on Fast EARTH-MANNA for N-Queens-T (10)

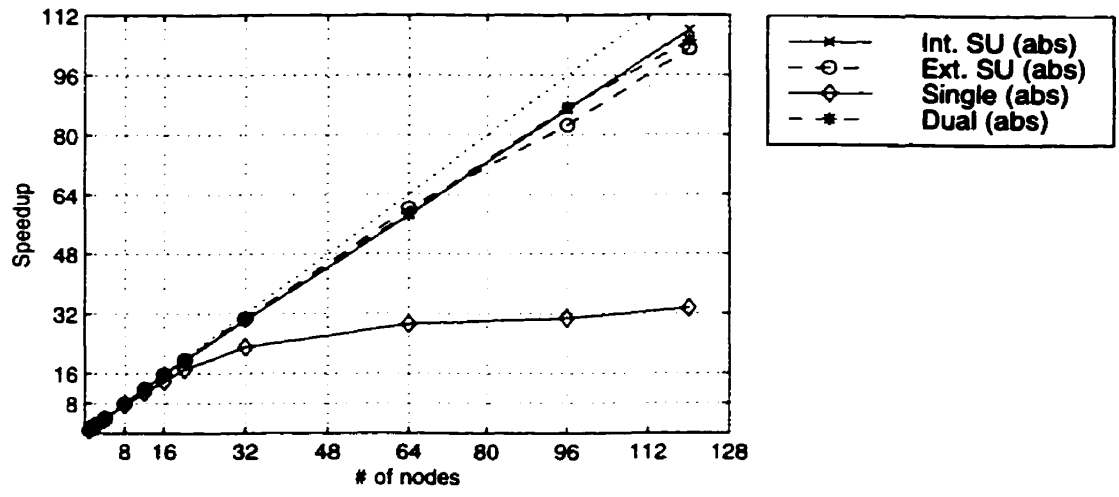


Figure D.10: Absolute Speedups on Fast EARTH-MANNA for N-Queens-T (12)

of the SU module, as coded in SEMi, was not tuned for the new performance parameters; a more detailed analysis could be used to adjust the load balancer. The most unusual results came from Tomcatv. The USE factors for Tomcatv on the software-based SU platforms drop sharply as the problem size increases, and the problem is compounded by relative speedups that level off far below their theoretical maxima. Cache statistics gathered by SEMi showed that the large block moves inherent in this application are the cause; there is significant contention in the EU on-chip caches between the fibers running in the EU and the SU copying or transferring entire rows, and a high volume of cache-coherence traffic on the memory bus, exacerbated by the lower relative performance of the main memory. Tuning the SU's runtime system could help, but it is also interesting to note that this problem disappears in both of the hardware-SU-based platforms.

The results from these experiments suggest that the main conclusions of this dissertation, the ability to support multithreading on off-the-shelf systems and the benefits of custom hardware to support the multithreading program execution model, apply not only to older processors such as the i860, but to higher-performance superscalar processors with multiple-instruction issue and higher clock speeds.

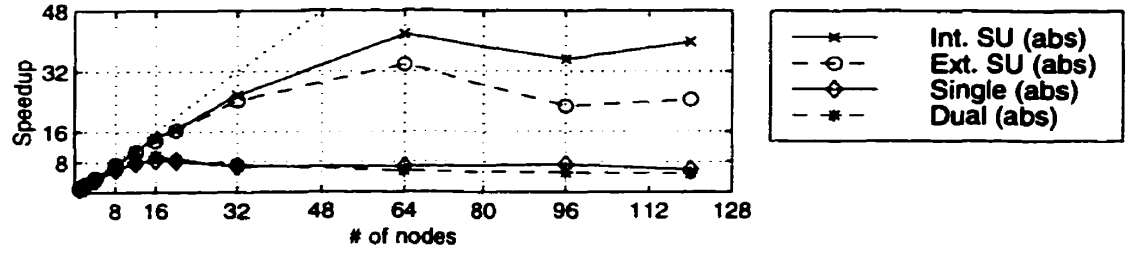


Figure D.11: Absolute Speedups on Fast EARTH-MANNA for Paraffins (18)

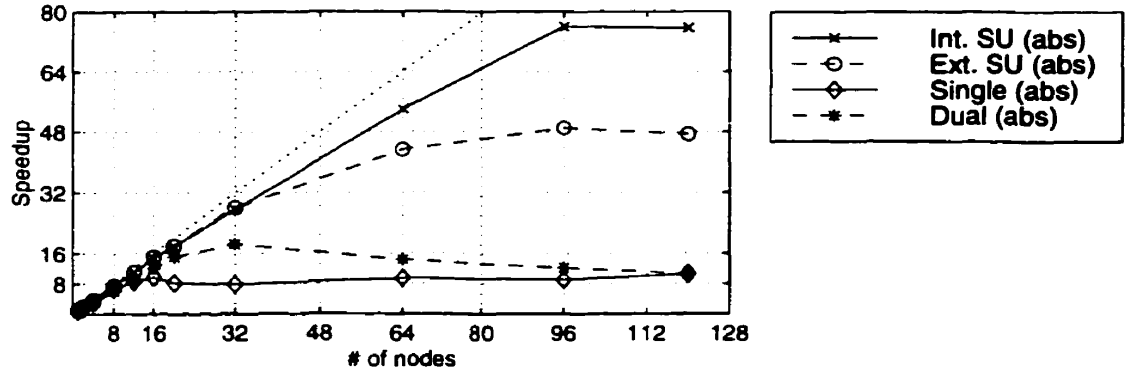


Figure D.12: Absolute Speedups on Fast EARTH-MANNA for Paraffins (20)

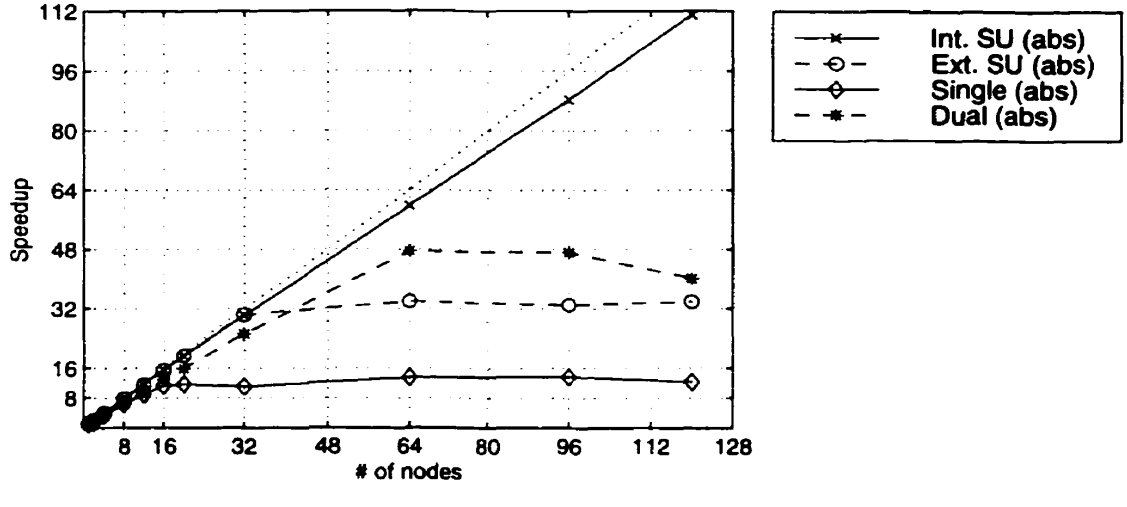


Figure D.13: Absolute Speedups on Fast EARTH-MANNA for Paraffins (23)

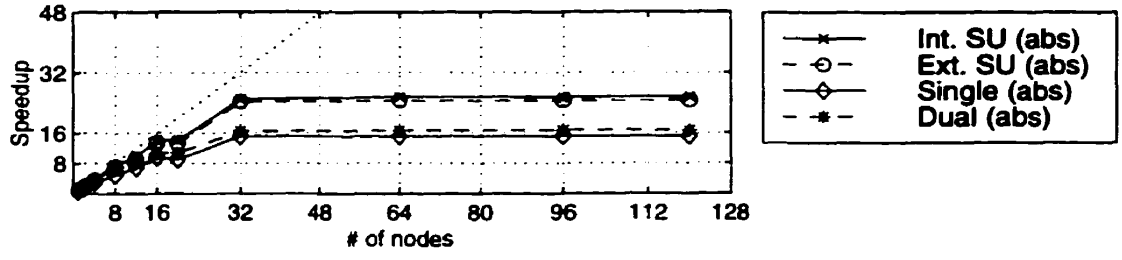


Figure D.14: Absolute Speedups on Fast EARTH-MANNA for Tomcatv (33)

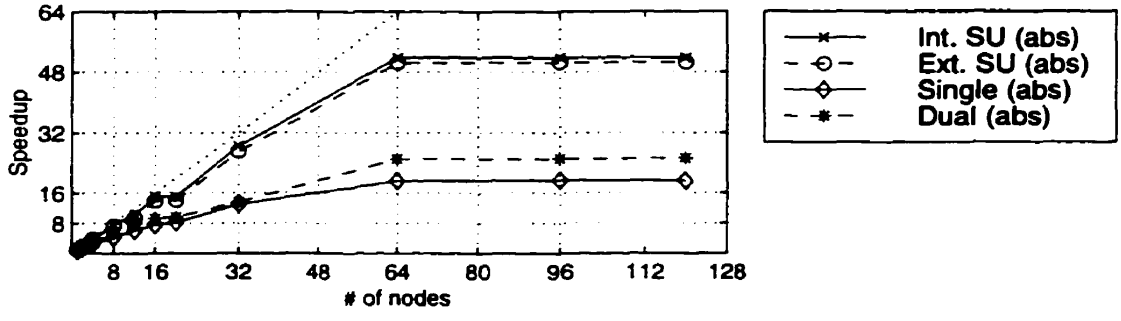


Figure D.15: Absolute Speedups on Fast EARTH-MANNA for Tomcatv (65)

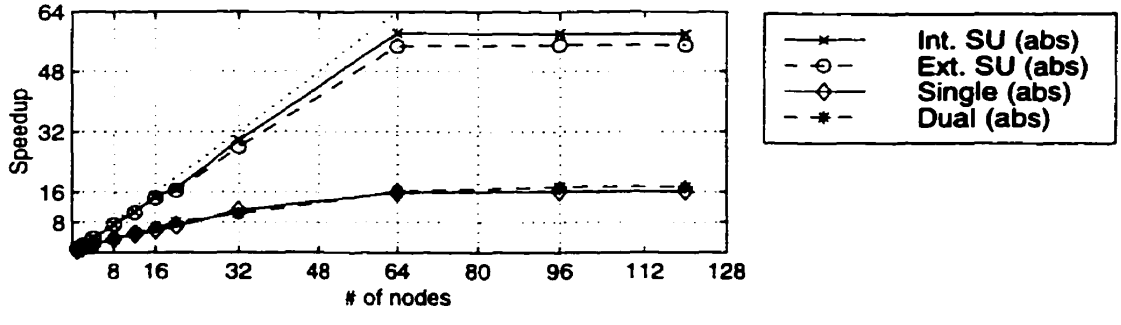


Figure D.16: Absolute Speedups on Fast EARTH-MANNA for Tomcatv (129)

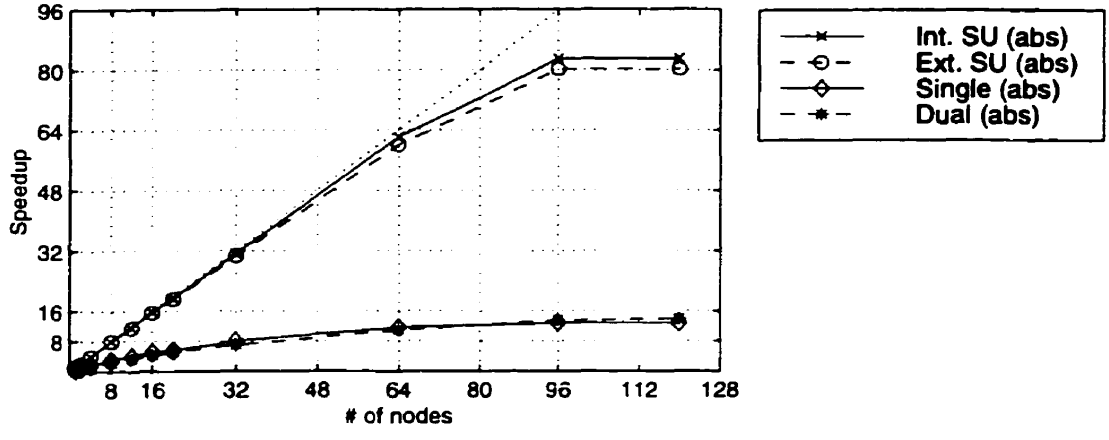


Figure D.17: Absolute Speedups on Fast EARTH-MANNA for Tomcatv (257)