University of Delaware
Department of Electrical and Computer Engineering
Computer Architecture and Parallel Systems Laboratory

The Benefits of Hardware-Assisted Fine-Grain Multithreading

Kevin B. Theobald
Guang R. Gao

CAPSL Technical Memo 32
July 20, 1999

Copyright © 1999 CAPSL at the University of Delaware

(Submitted for publication)

University of Delaware • 140 Evans Hall • Newark, Delaware 19716 • USA
http://www.capsl.udel.edu • ftp://ftp.capsl.udel.edu • capsladm@capsl.udel.edu

Abstract

Today there is widespread interest in using off-the-shelf computers to build economical supercomputers. Clusters such as Beowulf can use packages such as MPI to run coarse-grain parallel code. While this usually works well for applications with regular control structures and data distributions, it is not so effective for many irregular applications. For such problems, fine-grain parallel programs express the algorithm more naturally, adapt better to changing conditions, and balance the load more effectively. However, fine-grain parallel computing has overheads which have hurt performance, especially on off-the-shelf systems.

We show that fine-grain parallel programming *can* be supported efficiently on such systems, if there is a suitable program execution model and a small amount of specialized hardware. We present a general model for a thread hierarchy based on *fibers* and *threaded procedures*. The former are executed non-preemptively, which allows them to run efficiently on off-the-shelf processors. We show how the remaining features of the model (e.g., interaction between the fibers) can be supported efficiently in a small amount of external hardware assisting a commodity processor, and that this hardware can be added in an evolutionary manner. Experiments show our hardware support significantly reduces multithreading overheads and improves load balancing, leading to substantial improvements in processor utilization and speedups, especially for the most fine-grained benchmarks tested.

Contents

1 Introduction 1

2 Fine-Grain Parallelism and Multithreading 2

3 A Thread Model Based on Fibers 4
3.1 Fibers . . . 4
3.2 Threaded Procedures . . . 5
3.3 Examples of Two-Level Systems . . . 7

4 Hardware Support for Efficient Fibers 8
4.1 An Architecture with Hardware Co-Processors . . . 8
4.2 Benefits of Hardware Support . . . 10
4.3 An Evolutionary Approach . . . 11

5 Experiments 12
5.1 Experimental Platforms . . . 12
5.2 Performance of Low-Level Operations . . . 14
5.3 Performance on Real Programs . . . 15

6 Related Work 19

7 Conclusion 20

A Experiments with a Faster CPU 24

List of Figures

1 Sequential Fibonacci Example . . . 5
2 Call Graph for Fibonacci . . . 6
3 Threaded Fibonacci . . . 6
4 Architecture for Supporting Fibers . . . 9
5 Evolving Fiber Architectures . . . 11
6 Producer-Consumer Synchronization . . . 17
7 Speedups for Fibonacci (30) . . . 17
8 Speedups for N-Queens-P (10) . . . 17
9 Speedups for N-Queens-T (10) . . . 18
10 Speedups for Paraffins (20) . . . 18
11 Speedups for Tomcatv (257) . . . 19
12 Speedups on Fast EARTH-MANNA for Fibonacci (30) . . . 25
13 Speedups on Fast EARTH-MANNA for N-Queens-P (10) . . . 26
14 Speedups on Fast EARTH-MANNA for N-Queens-T (10) . . . 26
15 Speedups on Fast EARTH-MANNA for Paraffins (20) . . . 27
16 Speedups on Fast EARTH-MANNA for Tomcatv (257) . . . 27

List of Tables

1 Latency and Bandwidth on EARTH-MANNA . . . 14
2 EU Costs (nsec.) of Various Operations on EARTH-MANNA . . . 15
3 Benchmarks and Sequential Performance . . . 16
4 Parameters of Original and Modified MANNA . . . 25

1 Introduction

Most of today's general-purpose supercomputers are based on commercial off-the-shelf (COTS) processors. These systems range from high-end parallel computers, which typically use such microprocessors in an otherwise custom-designed system, to economical clusters such as the Beowulf [3], built entirely of mass-market PCs. Since the core processors are designed for uniprocessor PCs and workstations, these systems have had to adapt to the limitations of those processors, which lack the special parallel processing features found in full-custom multiprocessors such as Tera [2]. Most COTS-based machines support parallelism at the software level, with packages such as MPI [4].

However, these machines have had a mixed record of success. For applications with highly-regular data distribution and control structures, they can usually deliver high processor utilization rates. Irregular applications, on the other hand, tend to do much worse on such machines. In many cases, the high-end ASCI machine is only able to achieve 10% of peak performance, with load imbalance, software overheads and poor scalability cited as major impediments [22].

These performance problems are mainly due to the lack of support for effective fine-grain parallelism. Coarse-grain parallelism is usually adequate for regular applications; their regularity makes it easy to combine communication and computation tasks into large units, amortizing the creation and startup overheads of these tasks. Irregular problems, on the other hand, work better when grain sizes are small. Fine-grain parallel programs express the algorithm more naturally, expose more parallelism, adapt better to changing conditions, and balance the load more effectively.

We believe that irregular applications are best served by fine-grain multithreaded systems with the following characteristics:

1. There may be many threads per processor at runtime, and the number of threads may vary dynamically.
2. There is support for automatic load-balancing of the threads.
3. The costs of creating, removing, and switching between threads are very small.

(These properties, and the benefits they provide, are covered in Section 2.) Unfortunately, commodity microprocessors don't support efficient low-cost multithreading. As a result, fine-grain programs run on such processors suffer the overheads associated with multithreading.

This paper demonstrates that efficient fine-grain parallel programming *can* be achieved with COTS processors if they are augmented by a small amount of *external* hardware specifically designed to support a fine-grain parallel programming model. Using the EARTH multithreading system [14] as a base, we accurately simulate several different implementations, with and without external hardware support. Experimental results show that a co-processor-augmented COTS processor achieves significant improvements with fine-grain applications over comparable processor-only systems. The performance gap widens further when this hardware support is

added to a multiprocessor, because load balancing is much more effective than when processors are used alone. Finally, we show that such support can be added incrementally, in an *evolutionary* manner, with further improvements in efficiency as the specialized hardware is integrated with the processors.

The rest of this paper presents our models and experiments. The next section elaborates our definition of fine-grain multithreading and contrasts this with the types of multithreading used in other systems. In Section 3, we develop a threading model, based on *fibers*, which supports this type of fine-grain programming and is tuned to the needs of off-the-shelf processors. Section 4 discusses the role of hardware support and explains how it can improve performance on a machine running a fiber-based thread model. In Section 5, an accurate cycle-by-cycle simulator is used to measure the performance of various platforms, based on the same processor and running the same threading model, but with different levels of hardware support. Empirical observations of the various ways in which hardware support boosts multiprocessor performance are discussed in this section. Related work is surveyed in Section 6. The final section summarizes the paper.

2 Fine-Grain Parallelism and Multithreading

Terms such as "thread" and "fine-grain" appear frequently in the literature, but the field is still in flux and definitions vary widely from one paper to another. In this section, we clarify what we mean by these terms. Our concepts are shaped by our goals of supporting parallel applications (including irregular ones) effectively, and implementing a multithreaded system on off-the-shelf processors.

Essentially, we claim that the subtasks into which parallel applications are divided should be plentiful, lightweight and cheap. Most conventional parallel programming takes the opposite approach; tasks are made as coarse as possible to minimize the overheads associated with starting and terminating tasks and communicating between them. Coarse-grain task decomposition works well with many regular applications, because a regular application can be statically partitioned relatively easily, with simple techniques such as blocking used to combine tasks and amortize overheads. However, for many irregular applications, determining a good static partition is difficult or impossible, since the structure of the problem may depend on the input data. Furthermore, the data distribution may change dynamically.

Consider the class of unstructured mesh problems [13], an important tool for physical modeling applications. In one of these programs, a region of space is divided into an *irregular* grid whose density varies both spatially and temporally. During the simulation, a region of space may be *refined* or *coarsened* as the problem state changes.

A conventional parallel programming approach would try to divide the space into a small number of regions, perhaps only one or a few threads per processor, to cut down on communication and threading overheads. However, from the perspective of the application, it is more natural to think of each small region of space as a separate task or thread. It is an extra burden

to the programmer to combine small regions into larger sections, and, more importantly, to keep all sections at roughly the same size to balance the work among the processors. It would be easier, to the programmer, to divide the mesh into many atomic regions, and let the system balance these regions dynamically.

However, fine-grain programming has had its own failures, mainly due to the overheads of creating, terminating and managing a large number of threads. If fine-grain programming is ever to be an efficient programming technique for unstructured meshes and other applications, these overheads must be significantly reduced.

Therefore, when we discuss fine-grain multithreading, we are describing systems in which threads are

Abundant: There may be many threads per processor, and, as a corollary, each thread will be relatively short. Having an abundant pool of active threads on a processor increases processor utilization, because if one thread is delayed (e.g., due to a remote fetch), another thread can start execution. The system must make large-scale thread generation easy, if sufficient parallelism is available in the application. (For example, some of the benchmarks in Section 5 generate hundreds of thousands of threads.)

Balanced: A large pool of available threads provides better opportunities for load balancing on a parallel machine. There must be effective mechanisms to take advantage of this. Excess work generated on one processor should be readably and economically shifted to other processors so that total execution time is minimized. This is *essential* to achieve the adaptability required to support and manage dynamically changing data localities and workloads.

Cheap: Frequent thread switching and load balancing are possible only if these threads are extremely lightweight and have minimal overheads. For instance, creating or terminating such a thread should only involve the saving or restoring of a tiny fraction of the processor state – perhaps a few registers and nothing else. Communicating between threads and coordinating their actions should also have as little overhead as possible.

This definition should be contrasted with other concepts of multithreading. The term "fine-grain multithreading" is sometimes used to refer to the rapid *interleaving* of instructions from different threads, as in Tera [2], Alewife [1], the M-Machine [10], and Simultaneous Multi-threading processors [32]. During each cycle, instructions from one or more threads may be selected for execution. The processor may switch to other threads when there is a cache miss, as in Alewife, or as often as every clock cycle, as in the other designs listed. Such interleaving can be an effective way to tolerate long latencies and increase instruction issue rates. But it is orthogonal to the characteristics (length and quantity) of the threads themselves. Furthermore, these techniques are unavailable to contemporary COTS processors.

3 A Thread Model Based on Fibers

The previous section described the characteristics that threads need to have on a parallel system to handle the most general problems, including irregular applications. This section presents a model for a general, two-level thread hierarchy which is compatible with these requirements. Threads in this model are (or can be) abundant, balanced and cheap. They also have additional properties which make them suitable for efficient execution on COTS processors.

3.1 Fibers

The main distinguishing characteristic of this model is the division of threads into two levels, which we call *fibers* and *threaded procedures* [30]. We use the term "fiber," rather than the more generic term "thread," to distinguish our notion of lightweight fine-grain threads from other threading concepts.[1] In our model, a fiber is a *sequentially-executed, non-preemptive, atomically-scheduled* set of instructions.

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 multiple functional units, 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 obeys the sequential semantics of the code.

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 COTS 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 reload them when the fiber is restarted.[2] 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

[1] The term "fiber," 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.

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

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

Figure 1: Sequential Fibonacci Example

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 system (not the programmer) is responsible for making this determination and deciding when a fiber can start according to this restriction. The basis on which this determination is made depends on the specifics of the thread model; different synchronizing rules are compatible with our fiber model.

3.2 Threaded Procedures

Many threading systems have a single type of thread, for instance, where a thread corresponds to a function call. However, if fibers in our model are to be preemptive, a second layer *above* the fibers is needed. The need for this layer is illustrated with an example.

Figure 1 shows a simple recursive program for computing the nth 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 good benchmark for measuring multithreading overheads, as will be used later.) The call graph resulting from calling `fib(4)` is shown in Figure 2.

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. Yet the property of non-preemptiveness disallows this.

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,

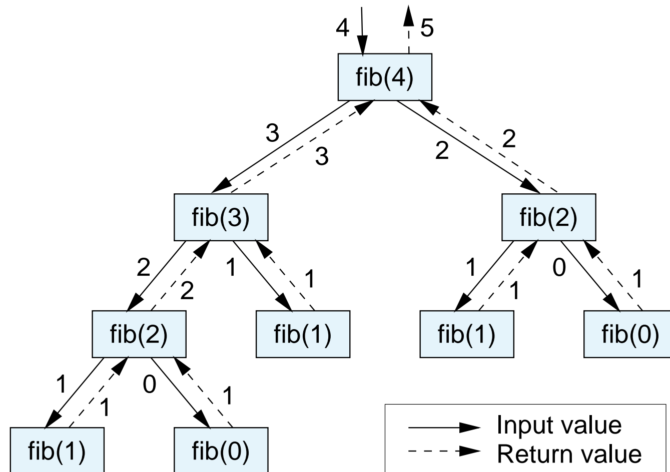

Figure 2: Call Graph for Fibonacci

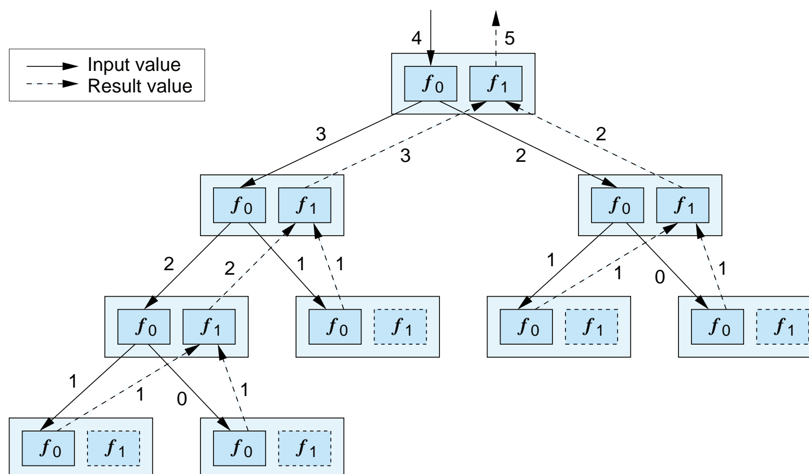

Figure 3: Threaded Fibonacci

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 3 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. 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 need leads to a *two-level thread hierarchy*. In the Fibonacci example of Figure 3, the

fibers f_0 and f_1 are combined into a single *threaded procedure.* 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. 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. Each threaded procedure has one fiber designated as the *initial fiber,* which starts automatically when the procedure is invoked. This fiber initializes the frame and sets up any synchronization mechanisms required by the fibers in that procedure, according to the specific synchronization mechanism in the particular threading system used. A threaded procedure instance remains "live" even if none of its fibers are currently active; a threaded procedure must explicitly terminate itself.

Thus, *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 and local variables. 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.

3.3 Examples of Two-Level Systems

Two examples of fine-grain systems with two-level thread hierarchies are EARTH [15, 20] and Cilk [11]. Cilk has an effective method for reducing overheads. The Cilk system can compile both parallel and sequential versions of recursive functions, and switch from running the parallel versions to the sequential versions once enough parallelism has been achieved. However, it is limited to shared-memory machines, and works primarily with divide-and-conquer algorithms, while other programming paradigms such as producer-consumer do not work as well on Cilk. EARTH-MANNA (discussed in the next section) works on distributed-memory machines and a wider range of programming models, and its overheads are small compared to most parallel programming systems. However, they are still large enough to become significant when thread

sizes are very small.

Both of these systems are currently implemented purely in software on off-the-shelf multiprocessors. The limitations of both these systems suggest that implementing a fine-grain system, using COTS processors, with abundant, balanced and cheap threads will be difficult.

We show in this paper that an off-the-shelf processor can be assisted by a special, relatively small hardware co-processor designed to support fibers. Replacing a software-based fiber manager with a hardware manager has two major benefits. First, it reduces the overheads to an acceptable level, improving the performance of threaded applications on a single node. The degree of improvement is most significant for the finest-grain applications. Second, it improves the speed of the threaded load-balancer, enabling it to make more optimal load-balance decisions and improve speedups. Our results show near-linear speedups up to 120 processors for nearly all benchmarks tested.

4 Hardware Support for Efficient Fibers

The previous sections made the case for having a programming model based on fine-grain, lightweight threads, and defined a general two-level thread hierarchy designed with COTS processors in mind. This section considers how a fiber-based model could be implemented on such processors efficiently. We first discuss general implementation issues, and then describe how efficient support for fibers could be added to mainstream processors in an evolutionary manner.

4.1 An Architecture with Hardware Co-Processors

Section 3.1 pointed out that modern conventional processors execute sequential code very efficiently, and can exploit much of the limited ILP that exists within a fiber. Considerable effort has been spent on the design of ILP logic within the CPU. We believe that far less would be required to add effective support for fine-grain multithreading as described in this paper, since it is based on "fibers" of sequential code.

However, mainstream CPU manufacturers are not likely to do this until the concept is proven and the market demands it. Most parallel computer manufacturers, on the other hand, lack the resources required to build components comparable to current mass-market processors.[3] In the short term, therefore, parallel computers based on our two-level thread model will have to use microprocessors which don't have built-in support for most of the features of the model, such as fiber management, scheduling and load balancing.

One can simply live with the fact and program the support needed into the processors, as is done in many multithreaded systems [5, 6, 11, 21]. An alternate approach is to use a regular processor for that which it can do well (running sequential fibers), and move the tasks

[3]Tera Computer is one notable exception.

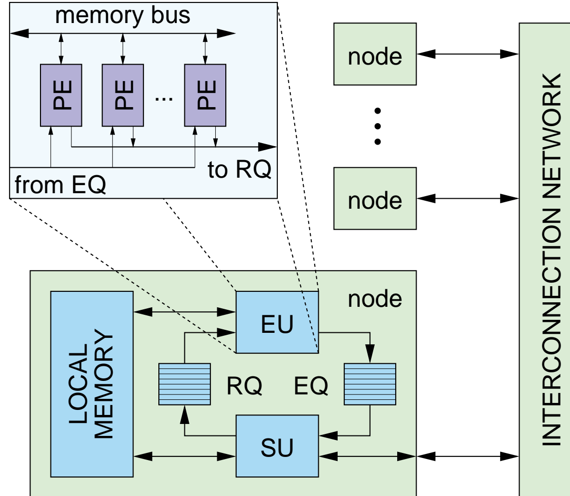

Figure 4: Architecture for Supporting Fibers

specific to the thread model to a custom co-processor. Such a machine might look something like Figure 4. This computer would consist of one or more multithreading *nodes* connected by a network. Each node would have 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;
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. A link to the interconnection network.

The simplest implementation would use one single-threaded COTS processor for each EU. However, the EU in this model can have processing resources for executing more than one fiber simultaneously. This is shown in Figure 4 as a set of parallel *Processing Elements* (PEs). These PEs could be separate processors (as in an SMP machine), or could abstractly represent the different streams of an interleaving processor (e.g., a Simultaneous Multithreading processor [32]).

The SU performs all multithreading features specific to the two-level threading model (and generally not supported by COTS processors). This includes EU and network interfacing, event decoding, sync slot management, data transfers, fiber scheduling, and load balancing.

The EU and SU communicate to each other through queues called the *Ready Queue* (RQ) and *Event Queue* (EQ). If a fiber running on the EU needs to perform an operation relating to

other fibers (e.g., to spawn a new fiber or send data to another fiber), it will send a request (an *event*) to the EQ for processing by the SU. The SU, meanwhile, manages the fibers, and places any fiber ready to execute in the RQ. When the EU finishes executing a fiber, it goes to the RQ to get a new fiber to execute. The queues may be implemented using off-the-shelf devices such as FIFO chips, incorporated into a hardware SU, or kept in main memory.

4.2 Benefits of Hardware Support

What are the advantages of using a separate hardware SU instead of emulating the SU functions in software? How would a hardware SU contribute to the goal of making fibers abundant, balanced and cheap?

First, auxiliary tasks can be efficiently offloaded onto the SU. If a single processor were used in each node, that processor would have to handle fiber support, diverting CPU resources from the execution of fibers. Even a dual-processor configuration, in which one processor is dedicated to fiber support, would not be as effective. Most general-purpose processors would have to communicate through memory, while a special-purpose device could use memory-mapped I/O, which would allow for optimizations such as using different addresses for different operations. This would speed up the dispatching of event requests from the EU.

Second, operations performed in hardware would be much faster in many cases. Many of the operations for fiber support would involve simple subtasks such as checking counters and following pointers. These could be combined and performed in parallel in perhaps only a few clock cycles, whereas emulating them in software might require 10 or 20 instructions with some conditional branches. Some operations might require tasks such as associative searches of queues or explicit cache control, which can be performed quickly by custom hardware but are generally not possible in general-purpose processors except as long loops.

Finally, many of the SU's tasks can be done in parallel. For instance, one part of the SU can be making load-balancing decisions while another part is sending a packet to the network and a third part is receiving an event from the EU. A conventional processor would have to switch between these tasks.

In general, these three differences would contribute to fiber efficiency in a system with a hardware SU. Offloading fiber operations to the SU and speeding up those operations would reduce the overheads associated with each fiber, making each fiber cheaper. A faster load-balancer, running in parallel with other components, would be able to spread fibers around more quickly, or alternately, to implement a more advanced load-balancing scheme to produce more optimal results. In either case, work would be distributed more evenly. Finally, special-purpose hardware would be able to handle more fibers in a given amount of time, allowing programmers to make fibers more abundant.

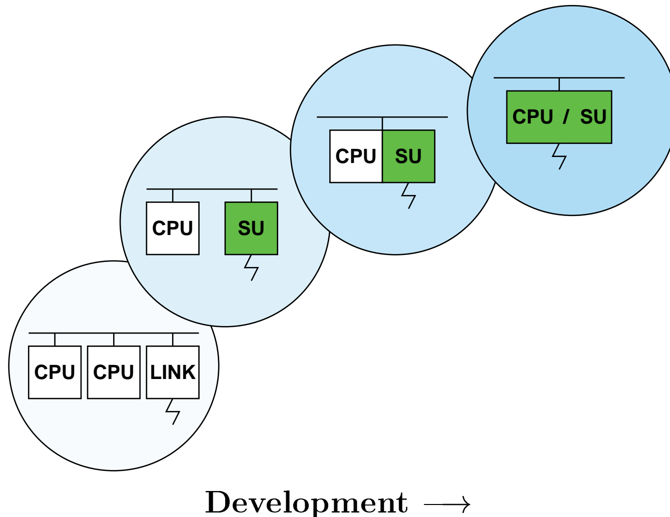

Figure 5: Evolving Fiber Architectures

4.3 An Evolutionary Approach

In spite of the potential benefits of fiber support, we aren't likely to see them incorporated into mainstream processors soon. A major benefit of this design is that the extra support can be be added incrementally, in an *evolutionary* manner, rather than in a single large step. Smaller steps are more feasible and present less risk, yet each step could yield tangible benefits. Figure 5 represents one possible evolutionary path toward a full-custom multithreaded processor, with the following steps:

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 experiments in the next section examine systems in the first three stages.

5 Experiments

Section 2 argued that to achieve efficiency in a fine-grain multithreaded system on the most general applications, threads need to be *abundant, balanced* and *cheap.* We show in this section that all requirements can be met on a COTS-processor-based system if it is assisted by custom hardware. In particular:

- Threads are *cheaper* because their overheads are much lower. For instance, EARTH operations can be requested by the EU 1.5–7.5 times faster with an ASIC SU co-processor than with a software-emulated SU.
- *Balancing* is much more effective because the hardware SU, being specialized to its tasks, can make more optimal load balancing decisions without imposing excess software overheads. Therefore, the hardware-assisted implementations have much better speedups, mainly due to a more even load distribution.
- Improving the other two properties allows threads to be more *abundant.* The benefit of this is shown by comparing the performance of a fully parallel program with an alternative version of the same program, which has been algorithmically "throttled" to limit parallelism (and its overheads). The results show that the hardware-supported system has less need for such throttling, reducing the burden on the programmer.

5.1 Experimental Platforms

To quantify the benefits of a hardware SU, we perform a case study comparing hardware and software implementations. Our study is based on EARTH (Efficient Architecture for Running THreads), a fine-grain two-level multithreading model originally developed at McGill University [15, 14, 20, 30]. The observations about the benefits of hardware support for fibers should be applicable to other systems with a similar two-layer thread model.

EARTH uses a two-layer hierarchy of procedures and fibers compliant with the model described in Section 3. Fibers are atomically scheduled (see Section 3.1), but the order among fibers is not completely specified. Instead, the EARTH model allows any fiber to execute, so long as all data and control dependences needed by that fiber have been satisfied. The SU is responsible for keeping track of dependences and scheduling fibers whose dependences have been met.

Dependences are made explicit in the EARTH code through the use of *synchronization signals* and *synchronization slots.* A synchronization signal is sent from one fiber to another to tell the recipient that a specific dependence has been satisfied. For instance, the sending fiber may have produced data required by the receiving fiber; since the latter can't run before that data is 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

synchronization slot is used to count the incoming signals so it is known when a fiber is ready to be enabled. (For more on the EARTH model, see the published papers on EARTH [15, 14, 20] or the first author's dissertation [30].)

EARTH currently runs on five platforms. The study in this paper is based on the MANNA platform, in which each node has two i860XP CPUs, memory, and a link to the interprocessor network, which is based on low-latency crossbars. The local interface to the link chip is on the main system bus and is memory-mapped. The configuration is similar to the lower-left corner of Figure 5.

There are two MANNA implementations. EARTH-MANNA-D uses both CPUs, using one for the EU (executing fibers) and the other to emulate the behavior of the SU in software. The latter CPU also polls the link for incoming messages and performs all load-balancing, which is based on a work-stealing algorithm [14, 19]. The Event Queue and Ready Queue are kept in memory and accessed by both CPUs. The second implementation, EARTH-MANNA-S, uses only a single CPU, which means that the SU functions are emulated in the EU (usually through inlining) [20]. In both implementations, network access and EARTH-specific operations (sending events to the EQ and reading fibers from the RQ) are performed entirely in the user code without kernel intervention.

In our study, we compare these implementations against implementations with special hardware support. We assume the second CPU in EARTH-MANNA-D can be replaced by an ASIC chip performing all the SU functions, such as synchronizing and scheduling fibers and performing load balancing, with all the optimizations described in Section 4.2. Furthermore, the Event and Ready Queues can be moved from memory to the SU chip, and memory-mapped, so that the EU can request EARTH operations simply by writing the event to the appropriate address, and read the Ready Queue by reading a specific address. Finally, since the link chip itself is a rather simple ASIC with two small queues, we assume it can be included in the SU. This will give the SU direct access to the link, reducing traffic on the system bus.

The experiments in this study were based on SEMi (Simulator for EARTH, MANNA and the i860) [30], an accurate, complete cycle-by-cycle simulator of the i860XP processors, system bus, memory system and interconnection network used in the MANNA. We used a simulator both to extend the speedup curves of the EARTH benchmarks from 20 nodes (the size of the MANNA) to 120 nodes,[4] and to simulate the benefits of the hardware extensions proposed in this paper. The simulator is highly accurate, as was validated by comparing the timing results of programs running side-by-side on the real MANNA and on SEMi. The timing discrepancies between the two are typically less than 2% on real benchmarks. This gives us confidence that the results obtained for the modified SEMi (simulating the specialized hardware in addition to the regular CPU) are reasonably close to what could be achieved with real hardware, since the specialized hardware is relatively simple compared to the CPU and other MANNA hardware.

Four systems are simulated in this study using SEMi. EARTH-MANNA-D and EARTH-

[4]120 nodes is the maximum currently supported on EARTH-MANNA. Adding more nodes will require modifying the virtual memory system. Such an effort is currently in progress.

Parameter	Single-processor	Dual-processor	External SU	Internal SU
Latency (ns)	2450	4091	1414	1200
Latency (cycles)	122.5	204.5	70.7	60.0
Bandwidth (MB/s)	28.8	42.0	44.4	44.4
Bandwidth (% of peak)	57.5	83.9	88.8	88.8

Table 1: Latency and Bandwidth on EARTH-MANNA

MANNA-S, described above, are completely off-the-shelf. Two more systems are simulated to measure the benefits of adding a hardware-based SU. The first assumes an external SU on the system bus, as in the second point in the evolutionary path in Figure 5. The second assumes the same SU placed on the same chip with an EU core, but with no additional integration of the logic. The EU still needs to communicate with the SU through loads and stores to and from memory-mapped I/O, but the internal memory-access logic will recognize these as internal addresses and send them to the internal SU without putting them on the main bus.

To perform the latter two experiments, a module simulating the SU functions was written and added to SEMi. To maintain the credibility of the simulation, the SU's two interfaces (the bus interface and the network interface) were copied from the existing module simulating the link chip, and run at the same speed. Therefore, in our simulations the EU communicates with the SU at the same speed as the link chip, a piece of real hardware.

The MANNA is several years old, and therefore slow by current standards. The processors, system bus and network all run at 50MHz. To confirm that our findings are also valid for newer processors, we modified SEMi to simulate a MANNA with performance parameters as near as possible to its successor, the PowerMANNA, which is based on PowerPC 620 processors. Results from these experiments are presented in the appendix.

5.2 Performance of Low-Level Operations

The first two experiments compare the performance of individual low-level operations on the four systems. The first test (Table 1) is a simple measurement of raw latency and bandwidth. Latency is measured by invoking a simple procedure on two nodes connected through a single crossbar. Each procedure has a small fiber, which sends a synchronization signal to the other node, which starts the fiber on that node. The nodes have no other work during this time, and are therefore idle when not running these fibers. The table lists the one-way latency. This measurement includes the time it takes for the receiving node to start up the fiber. Thus, it is a practical measure of latency as seen by the program. Bandwidth is measured by sending large blocks from one node to another.

To put this in perspective, a survey of commercial parallel machines [8] found only two machines (both shared-memory) with latencies under 10μs. The fastest is the Convex SPP1200 (2.2μs). This machine runs at 120MHz, so its latency is 264 processor cycles, more than the

Operation	Single CPU		Dual CPU		External SU		Internal SU	
	Loc.	Rem.	Loc.	Rem.	Loc.	Rem.	Loc.	Rem.
Synchronization	300	588	504	504	40	60	40	60
Spawn new fiber	323	640	721	580	40	100	40	100
Switch fibers	441	—	530	—	282	—	142	—
Send data and sync.	480	660	580	606	80	100	80	100
Fetch data and sync.	620	700	580	620	100	180	80	140
Invoke procedure (1 arg)	479	806	760	620	198	200	139	140
Terminate procedure (1 arg)	760	—	794	—	287	—	148	—

Table 2: EU Costs (nsec.) of Various Operations on EARTH-MANNA

slowest EARTH-MANNA implementation.

Though the COTS-based systems already have low latency, there is a significant improvement when a hardware SU is introduced. The improvement is entirely due to faster communication within each node, because the network is the same.

A more dramatic demonstration of the benefit of a hardware SU can be seen by measuring the costs, on the EU, of performing various EARTH operations. Table 2 lists the costs to the EU of various operations, such as sending a synchronization to another fiber (in accordance with the EARTH fiber model), involving either local or remote nodes. For the single-CPU version, 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). For all other versions, this is the cost of sending the request to the SU (CPU or special hardware). This is not the total time required for the operation to be completed, but merely the time for the EU to finish making the request.

5.3 Performance on Real Programs

The final experiments measure the speedups of actual programs. Five benchmarks were used in this study. Table 3 lists the sequential running time of each benchmark running on one i860XP processor of the MANNA, i.e., without the execution of multithreading instructions (and their overheads).

The *Fibonacci* program is the code used for illustration in Section 3. 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 fiber linkage. Since many of the overheads in parallel programs involve interactions between different fibers, this benchmark gives a good upper bound on the overheads encountered by a parallel system.

The *N-Queens* problem is a familiar benchmark that typifies recursive searching problems. Two versions are used in this study. *N-Queens-P* is maximally parallel, invoking new threaded

Benchmark	Input	T_{seq} (sec.)	Description
Fibonacci	30	0.969	Recursive; high overheads
N-Queens-P	10 queens	0.541	Fully para. recursive enumeration
N-Queens-T	10 queens	"	Partially sequentialized
Paraffins	$N = 20$	0.228	Recursive enumeration
Tomcatv	$N = 257$	48.6	Regular, data-parallel, barrier

Table 3: Benchmarks and Sequential Performance

procedures at each recursion. Over 60,000 procedures are generated, with 2–3 fibers executed each. *N-Queens-T* is a modified form of N-Queens-P in which the parallelism is "throttled" algorithmically. When the search depth 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," considered challenging to parallelize [24]. This application enumerates all distinct isomers of each paraffin (molecule of the form C_nH_{2n+2}) up to a given size. Parallelism is exploited by invoking functions on all the processors to compute the radicals (basic reaction units to form larger molecules) and then invoking procedures to compute the paraffins for the required sizes.

Tomcatv (SPEC92), is representative of traditional coarse-grain applications and was selected to show that one can use EARTH without sacrificing the good performance obtained on such applications with conventional parallel programming. This code uses static work distribution rather than the automatic load-balancer. Each iteration updates a pair of 257×257 meshes, using calculations with horizontal loop-carry dependences. Iterations continue until the change drops below a threshold; this involves a global reduction and barrier. Separate rows synchronize with each other using a producer-consumer paradigm illustrated in Figure 6.

For comparing multithreaded implementations, the programs were rewritten in Threaded-C, an explicitly threaded language in which EARTH operations are added to standard ANSI C [14, 30]. All four platforms tested used the same compiler as the sequential programs, but provisions were made (using custom-written pre- and post-processors) for translating the EARTH operations into native instructions.

Results for the five benchmarks are shown in Figures 7–11. Each graph shows both the *absolute* and *relative* speedups for all four systems. Absolute speedups (speedup relative to the *sequential* running time in Table 3) should be used for judging the overall performance of a system. From the absolute speedup, one can see the combined losses due to both fiber overheads and an imbalanced load. The relative speedup (speedup relative to the running time of the *threaded* program on 1 node) is primarily an indication of the effectiveness of the

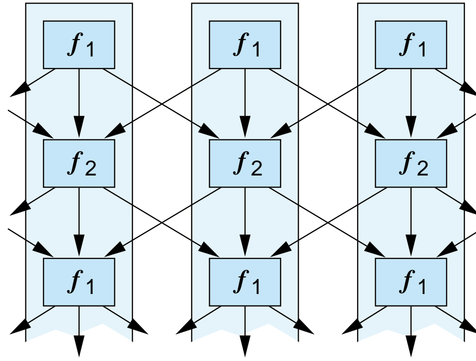

Figure 6: Producer-Consumer Synchronization

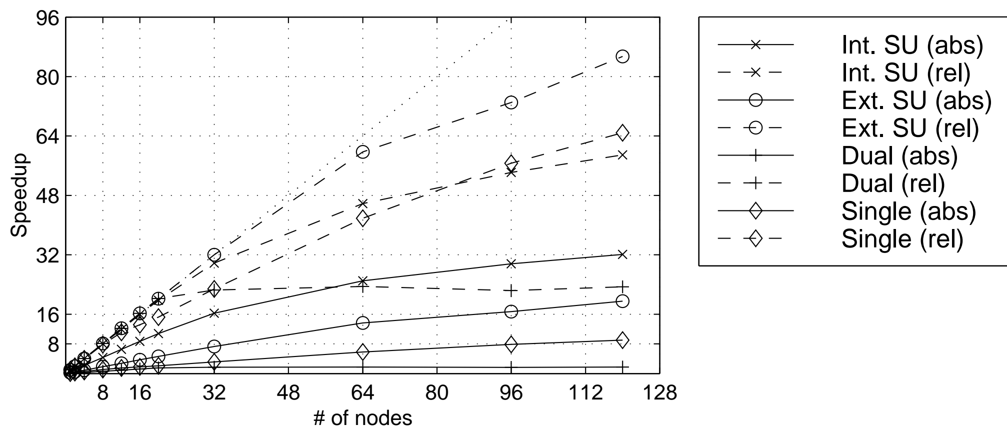

Figure 7: Speedups for Fibonacci (30)

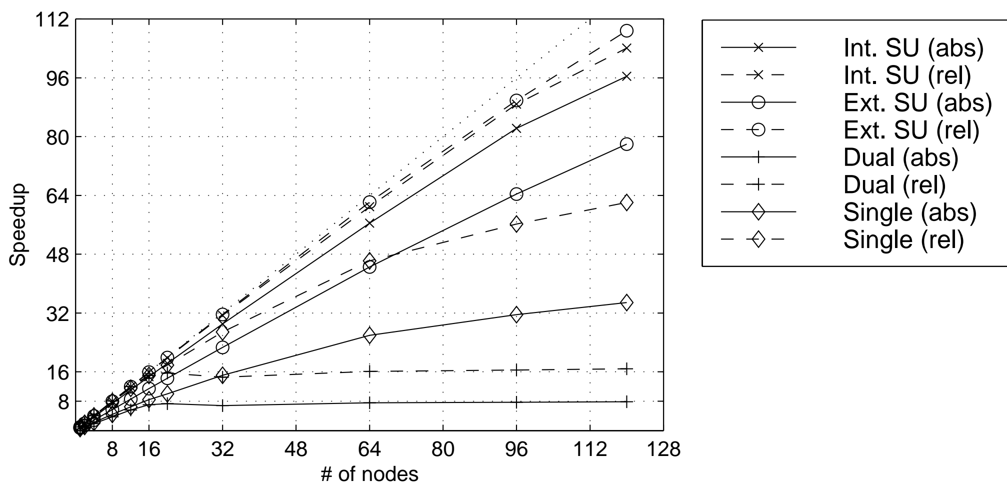

Figure 8: Speedups for N-Queens-P (10)

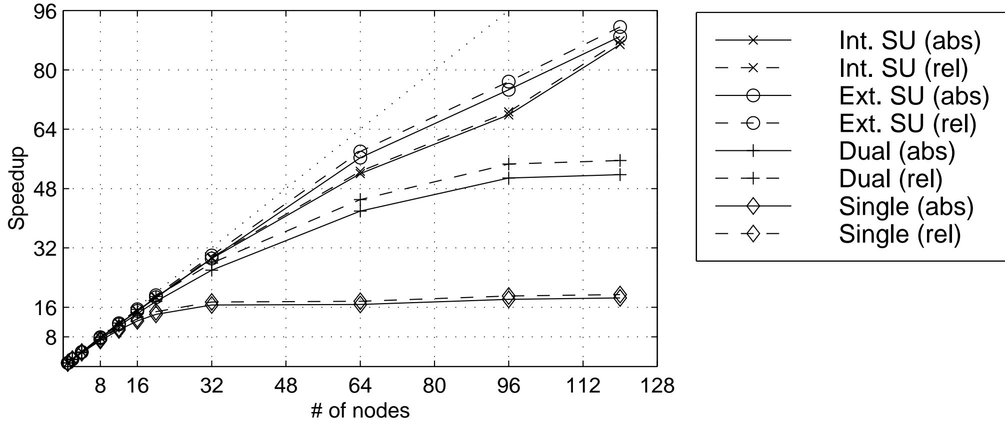

Figure 9: Speedups for N-Queens-T (10)

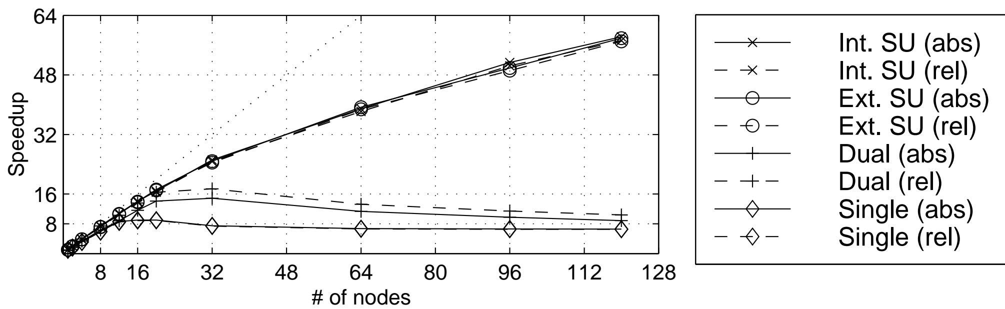

Figure 10: Speedups for Paraffins (20)

load balancer (this was verified using SEMi [30]), while the ratio between absolute and relative speedup mostly shows the effects of local fiber and communication overheads.

As expected, Fibonacci has the worst performance. It represents a worst-case measurement, since there are many multithreading operations and very little actual computations. Analysis with SEMi showed that in all four systems, the local buses were saturated by the traffic between the EU and SU or the EU and the link, while the sequential code's stack, 30 frames deep, fits entirely in the L1 cache. In such a case it is crucial to eliminate overheads as much as possible. The data in Table 2 show a substantial reduction in overheads, and this is reflected dramatically in the results for Fibonacci; a hardware SU doubles single-node performance compared to the best off-the-shelf implementation. Removing the EU-SU traffic from the bus (with an internal SU) doubles performance again, so that overheads are less than half the total execution cost (compared to sequential code).

The other, more realistic benchmarks show better performance by all the multithreaded systems, with a steady improvement seen along the evolutionary path (software SU $\longrightarrow$ external hardware SU $\longrightarrow$ internal SU). The overheads, as indicated by the ratio between absolute and

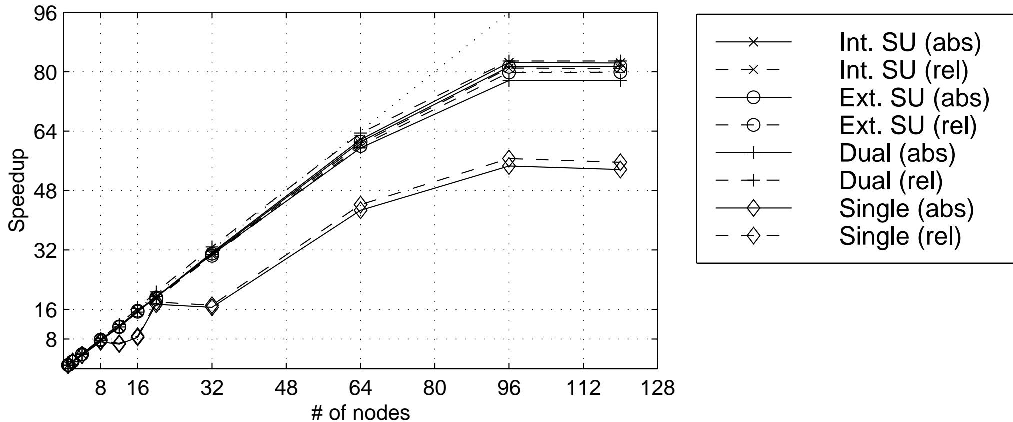

Figure 11: Speedups for Tomcatv (257)

relative speedups, are modest for the fully-parallel N-Queens program and very small for the other, coarser-grain programs. For N-Queens, the difference in performance (almost a factor of 3) along the evolutionary path is due both to the reduction in overheads and the greater efficacy of the load-balancer. For the other benchmarks, the performance gains are almost entirely due to improved load balancing. Thus, it can be seen that hardware support for fibers contributes both to their cheapness (reducing overheads) and their balance.

A comparison of Figures 8 and 9 shows the potential benefits of having many fibers, and how hardware support can contribute to this abundance. The former benchmark generates 67,150 threaded procedures, each of which generates two or three fibers. The latter benchmark was produced by limiting parallelism in the code, as a programmer may want to do in a coarse-grain system where overheads are large. While this throttling was relatively simple in this program, it can be far more difficult in complex programs. The figures show that given adequate hardware support, program throttling is unnecessary (in fact, sometimes the fully-parallel version has a faster *absolute* speedup). This is a benefit to the programmer, for it allows highly-parallel programs to be expressed in a more natural form expressing their inherent parallelism.

Also, abundance of fibers also helps to keep the code scalable into the realm of massively-parallel processors. In this example, since the fully-parallel code was able to produce parallelism more than 2 orders of magnitude beyond what was required (at 120 nodes), and the speedup curve is nearly linear at 120 nodes with little sign of tapering, this would suggest that, if the simulator were modified to allow larger simulations, speedups could be extended well into the hundreds.

6 Related Work

A number of articles have been published on multithreaded execution and architecture models and an introduction to the subject can be found in a survey article by Dennis and Gao [7].

Principal projects and representative work before 1995 have been discussed in two monographs [17, 12].

Multithreading models with fine-grain fiber models similar to EARTH include Cilk [11] (discussed in Section 3.3), Iannucci's hybrid multithreaded model [16], P-RISC [23], StarT [5], Berkeley TAM [6], and several projects in Japan [18, 25]. Many have been implemented on COTS-based systems using either one or two CPUs per node, and some have advocated the use of specialized hardware to support their threading models. We believe that these systems would benefit from such hardware, much like EARTH.

An entirely different form of multithreading is the interleaving approach, discussed in Section 2 [1, 2, 10, 32]. While these processors currently are not "off-the-shelf" and available for use in an EARTH-like system, there is considerable interest in them as they have demonstrated an ability to increase processor performance. Should this type of design become mainstream, it would open up exciting possibilities combining their benefits with fine-grain fiber models as described in this paper [30].

Other recent multithreaded architecture models include SPSM [9], Multiscalar [28], Superthreads [31], Trace Processor [27], and STAMPede [29]. Some of those new models, such as Multiscalar, Superthreads, and STAMPede, apply aggressive *control* and *data* (memory) speculation techniques to improve single-thread program performance. IRAM technology [26] permits memory-intensive data-movement operations to be placed directly next to the memories where full use of the local bandwidth can be achieved. Again, the possibility of a marriage between these processor techniques and a fine-grain fiber model opens up interesting research areas.

7 Conclusion

More and more, supercomputers are being built from off-the-shelf processors. These processors, being designed for the uniprocessor mass-market, can exploit local ILP, but are not equipped to support fine-grain multithreading. On the other hand, many general-purpose parallel applications, especially irregular problems, are difficult to program well using coarse-grain programming. We argued that such applications would have much better performance on a system in which threads are abundant, balanced and cheap (have low overheads).

We presented a general model for a thread hierarchy based on *fibers* and *threaded procedures*. The former are executed non-preemptively, which allows them to run efficiently on off-the-shelf processors. Such a model can run on a multiprocessor built entirely of commodity parts, but fine-grain multithreading can be much more efficient if a small amount of hardware is added to handle the tasks not supported well by the main processor, such as creating, switching, and synchronizing fibers. This hardware support can be added incrementally, along an evolutionary path, so that a quantum leap in hardware design is not mandatory.

An accurate, cycle-by-cycle simulation of four systems, based on the same processor and running the same programs, but differing in the amount of hardware support for multithread-

ing, demonstrates empirically that such hardware support indeed improves performance. The improvement is most significant for the more fine-grain applications.

References

[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 *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 2–13, Santa Margherita Ligure, Italy, June 22–24, 1995. ACM SIGARCH and IEEE Computer Society. *Computer Architecture News,* 23(2), May 1995.

[2] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. In *Conference Proceedings, 1990 International Conference on Supercomputing*, pages 1–6, Amsterdam, The Netherlands, June 11–15, 1990. ACM. *Computer Architecture News,* 18(3), September 1990.

[3] 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 *Proceedings of the 1995 International Conference on Parallel Processing*, volume I, Oconomowoc, Wisconson, August 14–18, 1995.

[4] Jehoshua Bruck, Danny Dolev, Ching-Tien Ho, Marcel-Cătălin Roşu, and Ray Strong. Efficient message passing interface (MPI) for parallel computing on clusters of workstations. In *Proceedings of the 7th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 64–73, Santa Barbara, California, July 17–19, 1995. SIGACT/SIGARCH and EATCS.

[5] Derek Chiou, Boon S. Ang, Arvind, Michael J. Beckerle, Andy Boughton, Robert Greiner, James E. Hicks, and James C. Hoe. StarT-NG: Delivering seamless parallel computing. CSG Memo 371, Computation Structures Group, MIT Laboratory for Computer Science, Cambridge, Massachusetts, February 1995.

[6] 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 *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 164–175, Santa Clara, California, April 8–11, 1991. ACM SIGARCH, SIGPLAN, SIGOPS, and the IEEE Computer Society. *Computer Architecture News,* 19(2), April 1991; *Operating Systems Review*, 25, April 1991; *SIGPLAN Notices,* 26(4), April 1991.

[7] Jack B. Dennis and Guang R. Gao. Multithreaded architectures: Principles, projects, and issues. ACAPS Technical Memo 29, School of Computer Science, McGill University, Montréal, Québec, February 1994. In ftp://ftp-acaps.cs.mcgill.ca/pub/doc/memos.

[8] Jack J. Dongarra, Iain S. Duff, Danny C. Sorensen, and Henk A. van der Vorst. *Numerical Linear Algebra for High-Performance Computers.* Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania, 1998.

[9] Pradeep K. Dubey, Kevin O'Brien, Kathryn O'Brien, and Charles Barton. Single-program speculative multithreading (SPSM) architecture: Compiler-assisted fine-grained multithreading. In Lubomir Bic, Wim Böhm, Paraskevas Evripidou, and Jean-Luc Gaudiot, editors, *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT '95*, pages 109–121, Limassol, Cyprus, June 27–29, 1995. ACM Press.

[10] Marco Fillo, Stephen W. Keckler, William J. Dally, Nicholas P. Carter, Andrew Chang, Yevgeny Gurevich, and Whay S. Lee. The M-Machine multicomputer. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 146–156, Ann Arbor, Michigan, November 29–December1, 1995. IEEE-CS TC-MICRO and ACM SIGMICRO.

[11] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 212–223, Montréal, Québec, June 17–19, 1998. *SIGPLAN Notices,* 33(6), June 1998.

[12] Guang R. Gao, Lubomir Bic, and Jean-Luc Gaudiot, editors. *Advanced Topics in Dataflow Computing and Multithreading.* IEEE Computer Society Press, 1995. Book contains papers presented at the Second International Workshop on Dataflow Computers, Hamilton Island, Australia, May 24–26, 1992, held in conjunction with the 19th Annual International Symposium on Computer Architecture.

[13] Gerd Heber, Rupak Biswas, and Guang R. Gao. A new approach to parallel dynamic partitioning for adaptive unstructured meshes. In *Proceedings of the 13th International Parallel Processing Symposium and the 10th Symposium on Parallel and Distributed Processing*, pages 360–364, San Juan, Puerto Rico, April 12–16, 1999. IEEE Computer Society and ACM SIGARCH.

[14] 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. *International Journal of Parallel Programming*, 24(4):319–347, August 1996.

[15] Herbert H. J. Hum, Kevin B. Theobald, and Guang R. Gao. Building multithreaded architectures with off-the-shelf microprocessors. In *Proceedings of the 8th International Parallel Processing Symposium*, pages 288–294, Cancún, Mexico, April 26–29, 1994. IEEE Computer Society.

[16] Robert A. Iannucci. Toward a dataflow/von Neumann hybrid architecture. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 131–140, Honolulu, Hawaii, May 30–June 2, 1988. IEEE Computer Society and ACM SIGARCH. *Computer Architecture News,* 16(2), May 1988.

[17] 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 Publishers, Norwell, Massachusetts, 1994. Book contains papers presented at the Workshop on Multithreaded Computers, held in conjunction with Supercomputing '91 in Albuquerque, New Mexico, November 1991.

[18] Yuetsu Kodama, Hirohumi Sakane, Mitsuhisa Sato, Hayato Yamana, Shuichi Sakai, and Yoshinori Yamaguchi. The EM-X parallel computer: Architecture and basic performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 14–23, Santa Margherita Ligure, Italy, June 22–24, 1995. ACM SIGARCH and IEEE Computer Society. *Computer Architecture News,* 23(2), May 1995.

[19] 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.

[20] 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 *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 178–188, Philadelphia, Pennsylvania, May 22–24, 1996. ACM SIGARCH and IEEE Computer Society. *Computer Architecture News,* 24(2), May 1996.

[21] Olivier Maquelin, Guang R. Gao, Herbert H. J. Hum, Kevin B. Theobald, and Xinmin Tian. Polling watchdog: Combining polling and interrupts for efficient message handling. ACAPS Technical Memo 99, School of Computer Science, McGill University, Montréal, Québec, November 1995. In ftp://ftp-acaps.cs.mcgill.ca/pub/doc/memos.

[22] James R. McGraw. Petaflop computing: Planning ahead. In Thomas Sterling, editor, *Proceedings of the Petaflops-Systems Operations Working Review*, Bodega Bay, California, June 1–5, 1998. Slides; see http://www.cacr.caltech.edu/powr98.

[23] Rishiyur S. Nikhil and Arvind. Can dataflow subsume von Neumann computing? In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 262–272, Jerusalem, Israel, May 28–June 1, 1989. IEEE Computer Society and ACM SIGARCH. *Computer Architecture News,* 17(3), June 1989.

[24] Rishiyur S. Nikhil and Arvind. Id: a language with implicit parallelism. CSG Memo 305, Computation Structures Group, MIT Laboratory for Computer Science, Cambridge, Massachusetts, February 1990.

[25] Kazuaki Okamoto, Shuichi Sakai, Hiroshi Matsuoka, Takashi Yokota, and Hideo Hirono. Multithread execution mechanisms on RICA-1 for massively parallel computation. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques (PACT '96)*, pages 116–121, Boston, Massachusetts, October 20–23, 1996. IEEE Computer Society Press.

[26] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A case for intelligent ram: Iram. *IEEE Micro*, 1997.

[27] Eric Rotenberg, Quinn Jacobson, Yiannakis Sazeides, and Jim Smith. Trace processors. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 138–148, Research Triangle Park, North Carolina, December 1–3, 1997. IEEE-CS TC-MICRO and ACM SIGMICRO.

[28] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 414–425, Santa Margherita Ligure, Italy, June 22–24, 1995. ACM SIGARCH and IEEE Computer Society. *Computer Architecture News,* 23(2), May 1995.

[29] J. Gregory Steffan and Todd C. Mowry. The potential for using thread-level data speculation to faciliatte automatic parallelization. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, Las Vegas, Nevada, January 31–February 4, 1998. IEEE Computer Society.

[30] Kevin Bryan Theobald. *EARTH: An Efficient Architecture for Running Threads.* PhD thesis, McGill University, Montréal, Québec, May 1999.

[31] Jean-Yuan Tsai and Pen-Chung Yew. The superthreaded architecture: Thread pipelining with run-time data dependence checking and control speculation. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques (PACT '96)*, pages 35–46, Boston, Massachusetts, October 20–23, 1996. IEEE Computer Society Press.

[32] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403, Santa Margherita Ligure, Italy, June 22–24, 1995. ACM SIGARCH and IEEE Computer Society. *Computer Architecture News,* 23(2), May 1995.

A Experiments with a Faster CPU

This appendix presents the data from the enhanced-CPU experiments, first described in Section 5.1. Experiments from Section 5.3 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 4. The parameters for the faster "MANNA" are mostly taken from the new PowerMANNA, which is based on the PowerPC 620 processor. Also, multi-issue capability is included and some architectural limitations of the i860 are removed (e.g., the lack of a reservation table for FP registers). The miss penalty (in CPU cycles) is increased, but an L2 cache is added.

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 (Kibytes)	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 (Mibytes)	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
Network	Clock speed (MHz)	50	66.7
	Bandwidth (Mbytes/sec)	50	133.3

Table 4: Parameters of Original and Modified MANNA

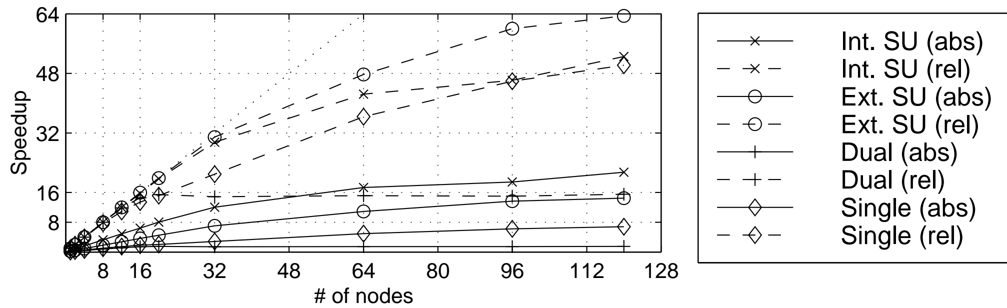

Figure 12: Speedups on Fast EARTH-MANNA for Fibonacci (30)

The following figures show absolute and relative speedups, and can be compared with the corresponding curves in Section 5.3. For the most part, the results here are consistent with the previous findings of improved performance from a hardware SU.

The results show some reductions in absolute speedups for Fibonacci and N-Queens. This is mainly due to the problems of bus saturation (mentioned in Section 5.3), which affect the parallel programs far more than the sequential programs because the working sets of the latter

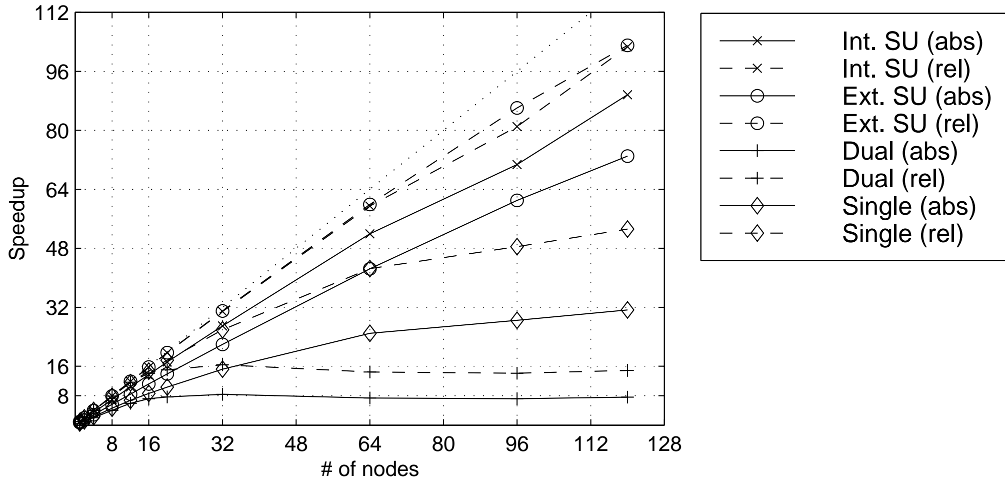

Figure 13: Speedups on Fast EARTH-MANNA for N-Queens-P (10)

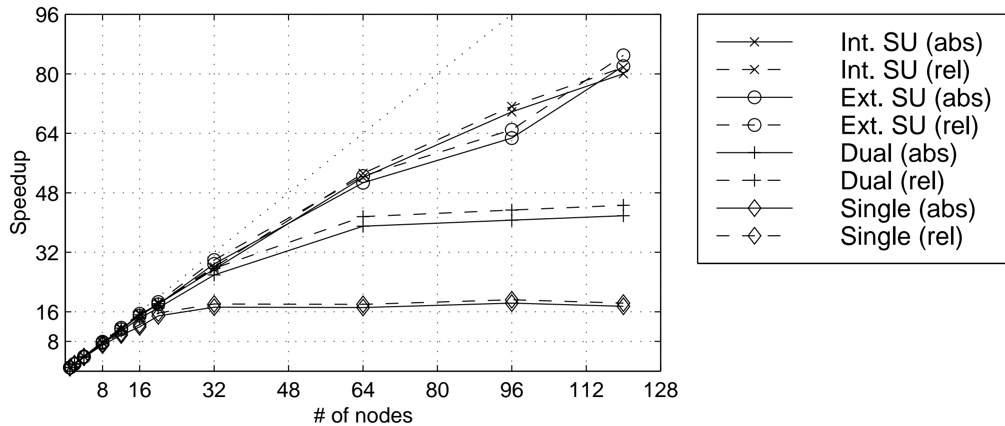

Figure 14: Speedups on Fast EARTH-MANNA for N-Queens-T (10)

fit entirely in the L1 cache. The difference between the two is likely to be far less with the types of applications intended for large parallel machines. In other words, if increasing the issue rate of the CPU causes the memory bus to become the bottleneck, this will probably be true for *any* parallel system, and here again, the overhead reductions offered by the hardware SU can still lead to an increase in performance.

Some anomolous results can also be seen with the last benchmark. For the software-SU systems, the absolute speedups are substantially lower than for the original MANNA (Figure 16), 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 data 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

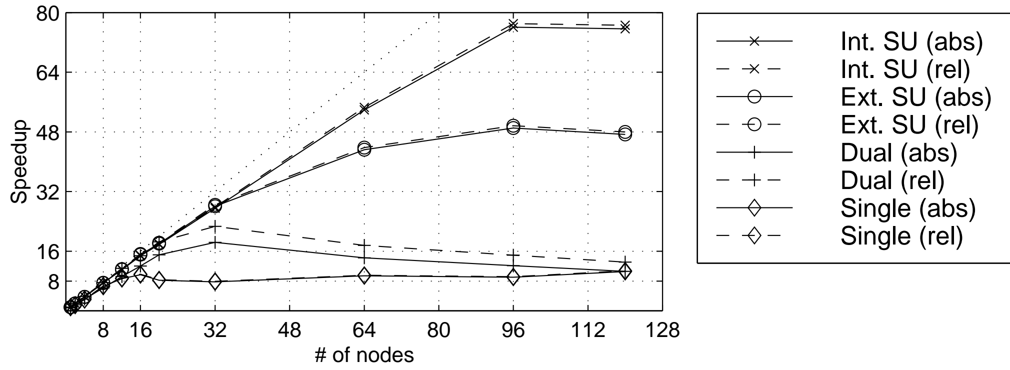

Figure 15: Speedups on Fast EARTH-MANNA for Paraffins (20)

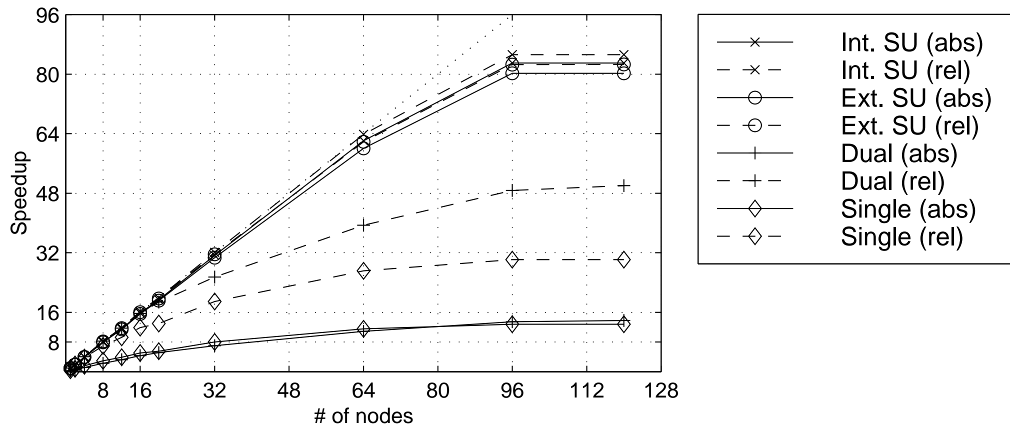

Figure 16: Speedups on Fast EARTH-MANNA for Tomcatv (257)

of cache-coherence traffic on the memory bus, exacerbated by the lower relative performance of the main memory. It is 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 paper, 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.