A Design Study of the EARTH Multiprocessor

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

Advanced Compilers, Architectures, and Parallel Systems (http://www-acaps.cs.mcgill.ca)

School of Computer Science, McGill University, 3480 University St., Montréal, Canada, H3A 2A7

† Dept. of Elec. and Comp. Eng., Concordia University, 1455 de Maisonneuve W., Montréal, Canada, H3G 1M8

‡ Dept. of Electrical Eng., McGill University, 3480 University St., Montréal, Canada, H3A 2A7

Abstract

Multithreaded node architectures have been proposed for future multiprocessor systems. However, some open issues remain: can efficient multithreading support be provided in a multiprocessor machine such that it is capable of tolerating synchronization and communication latencies, with little intrusion on the performance of sequentially-executed code? Also, how much (quantitatively) does such non-intrusive multithreading support contribute to the scalable parallel performance in the presence of increasing interprocessor communication and synchronization demands?

In this paper, we describe the design of EARTH *(Efficient Architecture for Running THreads), which attempts to address the above issues. Each processor in EARTH has an off-the-shelf RISC processor for executing threads, and an ASIC Synchronization Unit (SU) supporting dataflow-like thread synchronizations, scheduling, and remote memory requests. In preparation for an implementation of the SU, we have emulated a* basic *EARTH model on MANNA 2.0, an existing multiprocessor whose hardware configuration closely matches EARTH. This EARTH-MANNA emulation testbed has been fully functional, enabling us to experiment with large-scale benchmarks with impressive speed.*

With this platform, we demonstrate that multithreading support can be efficiently implemented (with little emulation overhead) in a multiprocessor without a major impact on uniprocessor performance. Also, we give our first quantitative indications of how much the basic multithreading support can help in tolerating increasing communication/synchronization demands.

Keywords: *multithreaded architectures, multiprocessors, performance measurements, EARTH.*

1 Introduction

Multithreaded architectures [1, 2, 3, 4, 5, 14, 15, 16, 18] have been promoted as potential processing nodes for future parallel systems due to their toleration of inherent parallel execution latencies. As long as there is enough parallelism in an application, a multithreaded architecture can hide the latencies inherent in parallel processing by quickly switching to another task when it encounters a long-latency operation. Although this point is well taken, there has been much reservation regarding the *non-intrusiveness of multithreading support.* Many architects of both uniprocessor and multiprocessor systems question whether multithreading support can be made transparent to sequentially-executing code and still be useful, or if users will always have to sacrifice efficient sequential execution for good multithreaded performance. Moreover, if multiprocessing support is transparent to sequential code, how much can interprocessor communication/synchronization latencies be hidden while still attaining the expected speedups? To the best of our knowledge, there are no experimental results which directly answer these questions.

In this paper, we address both the issue of non-intrusiveness of multithreading support, and the benefits of hiding communication/synchronization latencies, by performing experiments on an emulation platform of our *Efficient Architecture for Running THreads (EARTH)*[1] [11, 13]. Each node in an EARTH computer consists of an Execution Unit (EU), which is an off-the-shelf high-end RISC processor for executing threads sequentially, and a Synchronization Unit (SU) supporting dataflow-like thread synchronizations and communication with remote processors. The SU can be optimized for synchronization and communication tasks, relieving the EU of these responsibilities.

In our opinion, dividing the execution and synchronization into separate units is preferable to performing both tasks within a single processor. In the latter approach:

- Incoming synchronization requests need to interrupt the processor. Even if interrupts could be avoided by keeping synchronization events in a special queue, there are still the costs of context-switching and the sharing of registers with the main execution tasks.
- Long-latency operations, such as block moves, can tie up the processor and prevent it from performing com-

In the Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'95), Limassol, Cyprus, June 27–29, 1995, pages 59–68.

[1] In previous papers, this architecture was called the Multi-Threaded Architecture (MTA). However, this name is currently used by Tera Corporation (see section 6). Private discussions with Burton Smith revealed that we both first used 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).

putations, especially when the network is congested and the block move is stalled.

- Synchronization events involve only simple ALU tasks, so mechanisms such as floating point logic are not utilized for multithreading operations.

It may seem as though creating a separate synchronization unit would simply be doubling the processing power (and cost!), and would be equivalent to leaving synchronization tasks in the execution processor and doubling the number of processors, but this is not the case. Synchronization tasks are specific, thus they can be eventually implemented in a small amount of external hardware (or better yet, integrated into a RISC processor) which should be far less expensive than a second RISC processor.

To summarize, the main features of EARTH which support multithreading are:

- Augmenting the existing RISC instruction set with EARTH instructions which realize thread creations and terminations, and thread-level synchronizations and communications. Therefore, sequential code can take full advantage of the RISC processor as if it were run on a uniprocessor. (The EARTH instructions can be implemented directly using native instructions of existing processors.)

- A dedicated hardware unit (SU) for handling split-phase transactions and performing dynamic runtime scheduling and synchronizations.

- Support for dynamic load balancing (via work stealing) in which the overhead is incurred by the SU and not by the EU. Therefore, such overhead has minimal effects on the overall execution times.

There have been other proposals for multithreaded systems [1, 2, 3, 4, 5, 15, 16, 18] and some of them have considered a similar EU-SU configuration [3, 4, 16]. Instead of waiting for a complete implementation to begin answering our questions about multithreaded systems, we have chosen to implement an emulation system. The system is based on an existing multiprocessor called the MANNA 2.0 [4], which has a hardware configuration similar to the EARTH design. In this EARTH-MANNA platform, each MANNA node has two Intel i860XP processors, which we dedicate to the EU and SU tasks, respectively. This emulation system has given us the following benefits:

- The emulation system is very efficient, as the performance results in section 3 and section 4 demonstrate. Thus, we can develop and run applications far larger than would be possible with a simulator.

- We were able to develop a base system supporting basic multithreading tasks in less then three months, which is far less time than required by a hardware development effort.

- We were able to use a commercial-grade compiler as the basis of our compilation software.

- We have instrumented the EARTH-MANNA emulator to gather statistics on various multithreading operations. Such information will guide us in further refining EARTH.

- Because the SU in the emulator is programmed, we can experiment with more advanced features (as discussed in section 5).

We have developed a set of extensions to the C language which allow programmers to express EARTH operations directly in the code. Using this system, we have implemented several benchmarks and measured the non-intrusiveness of multithreading support on sequential code. For this, we introduce the *Uni-node Support Efficiency (USE)* measure to indicate the efficiency of programs generated for multithreading running on a single processing node of EARTH as compared to programs optimized for sequential execution running on one processor. For the benchmark programs we examined, we obtained a range of USE values from 72% to 107%.[2] In other words, the penalty paid for one of the multithreaded programs is less than 28% (while others are much less) when run on a single processor node of the EARTH-MANNA platform. Furthermore, this is for the case where the single node version executes the same amount of multithreading operations as the multinode versions. Moreover, we show that the same code which obtains high USE values can also attain good to near-linear speedups without requiring alterations to the threaded code.

Lastly, we performed an experiment to gauge the capability of EARTH-MANNA in 'absorbing' increased communication/synchronization demands without diminishing the speedups. For the Ray Tracing program, we show that the SU can handle about a 2500-fold increase of interprocessor communications/synchronizations over the original amount required with only a small degradation in performance (<10%). For N-Queens, that figure is a 10-fold increase before the performance degraded by 10%.

1.1 Synopsis

In the next section, we introduce the EARTH model which serves as a basis for implementations and as a target for our compiler work. In section 3, we describe the EARTH-MANNA platform used in our studies. We also detail the costs of various EARTH operations on the platform. In section 4, we show results from benchmark programs running on EARTH-MANNA. In section 5, we give an overview of our ongoing efforts in defining a more powerful EARTH, constructing an EARTH simulator, compiling C source programs for EARTH, and implementing EARTH in hardware. Lastly, we review related work in section 6 and summarize our work in section 7.

2 The EARTH Model

EARTH evolved from the dataflow model of computation. However, it differs from its dataflow predecessors in that instructions are not synchronized and scheduled individually, but are combined into larger units called *threads*. A thread is a segment of code which is executed sequentially (although control-transfer instructions, such as branches, are allowed), and synchronized externally.

Each EARTH node consists of an *Execution Unit* (EU) and a *Synchronization Unit* (SU), linked together by buffers (see figure 1). This has its roots in the McGill Data Flow Architecture [7] and the Super-Actor Machine [10]; these evolved from an earlier dataflow design [6]. The EU executes an active thread, which is initiated for execution when the EU fetches its *thread id* from the *ready queue*. The EU executes a thread to completion before moving to another thread, i.e., threads are non-preemptable. The EU interacts with the SU and the network by placing messages in the

[2] The super-unity USE value phenomenon is briefly discussed later and is a subject of another paper.

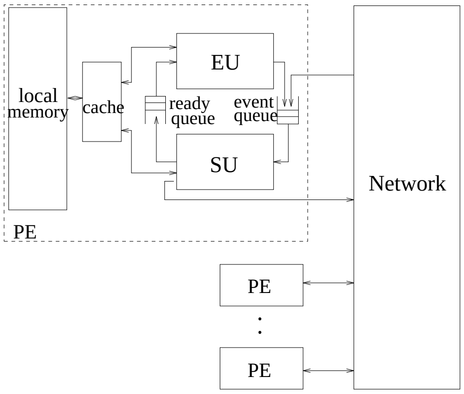

Figure 1: The EARTH multiprocessor

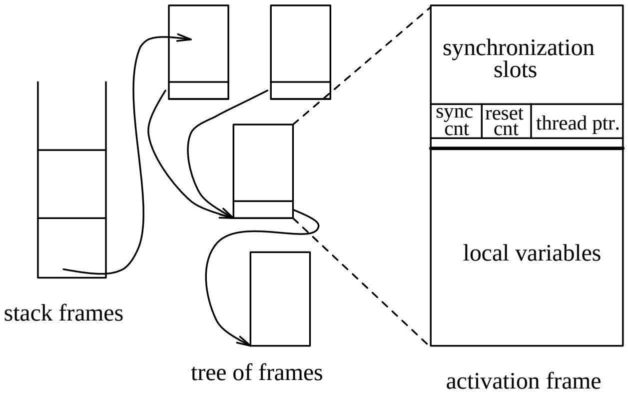

Figure 2: Stack and activation frames.

event queue. The SU fetches these messages, plus messages coming from remote processors via the network. The SU responds to remote synchronization commands and requests for data, and also determines which threads are to be run and adds their thread ids to the ready queue. The SU and EU share a local memory in which the aggregate of the local memories of all the nodes represents a global memory address space.

2.1 Runtime Model

In EARTH, threads are not ordered sequentially, but are scheduled according to data and control dependences. In general, a thread is *enabled* (ready for execution) when all data required by the thread is available. The data needed by the thread may be produced by other threads, whose ordering might not be determinate. Therefore, it is the SU's job to determine when all the requisite data is ready, and to place the thread id in the ready queue when this occurs.

This is done using *synchronization signals* and *synchronization slots.* A sync signal is used by the producer of a datum to tell consumers of that datum that the datum is ready. A sync slot is used to coordinate the incoming sync signals, so that a consumer knows when all required data is ready. Each sync signal is directed to a specific sync slot.

A sync slot contains three fields: a *reset count,* a *sync count,* and a thread id. The reset count, which is constant, gives the total number of sync signals required to enable a thread. The sync count indicates the *current* number of sync signals needed to enable the thread. When a sync signal is received by the SU, the sync count in the specified slot is decremented. If the count reaches 0, it is set back to the reset count, and the thread id is placed in the ready queue.

Because multithreaded machines allow different parts of a program to be active at the same time, procedure frames cannot be placed on a stack as in conventional processors. EARTH is similar to other multithreaded and dataflow machines in that frames are dynamically allocated from a heap [17]. Each frame represents the instantiation of one function, and contains the local state of that instantiation, including local variables and sync slots. A frame pointer *fp* points to the base of the frame and serves as a unique id. A thread must reside entirely within one function, but a function may contain more than one thread. A thread id represents a *specific* thread within *one* instantiation of a function; therefore, a thread id is a pair <*fp, ip*>, consisting of a frame pointer and a pointer to the first instruction of the thread.

Frames are not automatically linked, but the code can explicitly link frames to one another by passing frame pointers from one function to another. Any arbitrary graph structure is possible, although the simplest for the programmer to construct is a tree, as illustrated in figure 2. Section 2.3 contains an example of code that generates such a tree.

For efficiency, single-threaded functions that only call other single-threaded ('non-threaded') functions can use a conventional stack (one per processor) for function invocation. That is, once a thread in a 'threaded' (multiple threads) function calls a non-threaded function, the activation frame of the non-threaded function will be created on the runtime stack of the processor. Such non-threaded functions can only call other non-threaded functions. When the first non-threaded function terminates, control is returned to the thread in the threaded function which invoked the non-threaded function. Figure 2 shows the integration of a stack and an activation tree.

2.2 Instruction Set of EARTH

Instructions as seen by a programmer are those executed by the EU; the SU is not user-programmable. The EU instructions include standard RISC load/store, register-register, and control operations, augmented with instructions for multithreading (see table 1). In order for the EU to be implemented with an off-the-shelf processor, the multithreading instructions will actually be encoded using memory-mapped accesses, i.e., loads and stores to special addresses which are recognized and processed by the SU.

The **fetch_next** instruction is used to fetch the next thread id from the ready queue. The ready thread id's *fp* value is stored in a dedicated register (%fp) for subsequent use and the instruction pointer *ip* of the thread id is used to fetch the first instruction of the next thread.

The EU processes **spawn**, **sync**, **init_sync**, and **incr_sync** instructions by generating spawn, sync, init_sync and incr_sync signals to the SU, respectively. Signals to the SU contain the parameter values as specified in the instructions. If a signal is destined for a remote processor (detected by checking the node id field in *fp*), then the SU would package the signal and send it to a remote SU. A spawn signal instructs the SU to put the thread id <*fp, ip*> on the ready queue. A sync signal requires the SU to decrement the sync count of the specified slot, as described in section 2.2. The

Instruction Set
Arithmetic, Logic and Branching
typical RISC instructions, e.g., those from the i860
Thread Switching
fetch_next
Synchronization
spawn *fp*, *ip* **sync** *fp*, *ss_off* **init_sync** *ss_off*, *sync_cnt*, *reset_cnt*, *ip* **incr_sync** *fp*, *ss_off*, *value*
Data Transfer & Synchronization
data_spawn *value*, *dest_addr*, *fp*, *ip* **data_sync** *value*, *dest_addr*, *fp*, *ss_off* **blockdata_spawn** *src_addr*, *dest_addr*, *size*, *fp*, *ip* **blockdata_sync** *src_addr*, *dest_addr*, *size*, *fp*, *ss_off*
Split-Phase Data Requests
get_spawn *src_addr*, *dest_addr*, *fp*, *ip* **get_sync** *src_addr*, *dest_addr*, *fp*, *ss_off* **get_block_spawn** *src_addr*, *dest_addr*, *size*, *fp*, *ip* **get_block_sync** *src_addr*, *dest_addr*, *size*, *fp*, *ip*
Function Invocation
invoke *dest_PE*, *f_name*, *no_params*, *params* **token** *f_name*, *no_params*, *params* **end_function**

Table 1: EARTH instruction set

init_sync instruction is used to set a sync slot within the current frame. An incr_sync signal instructs the SU to add *value* to the sync count at (*fp* + *ss_off*). If the result is 0, then <*fp*, *ip*> is put onto the ready queue, where *ip* is the thread pointer in the slot. The incr_sync instruction is primarily used in cases where the number of synchronization signals destined for a slot can only be determined at runtime, such as when performing a barrier synchronization for all the threads spawned in a while loop.

Data transfer & synchronization instructions are used to write data to a specified address and to perform either a spawn or sync operation *after* the data has been written. Combining these two parts into an atomic operation is important because of the potential for race conditions in the network. For an instruction destined for a remote processor,[3] the SU forms a message containing the parameters, and sends it to the processor identified in the *fp* parameter. These instructions have block-transfer versions, which are used when a large block of data is to be transferred and synchronized as a single operation.

The **get_spawn** and **get_sync** instructions are used to retrieve data (usually remote) in a *split-phase manner*. If the address parameter *src_addr* indicates a remote access, the SU forms a get_spawn or get_sync message, and injects it into the net. The SU at the receiving end would respond by sending a data_spawn or data_sync message, respectively. The *dest_addr* parameter specifies where the retrieved data should be stored. After the data transfer is completed, the appropriate spawn or sync signal is sent to the specified thread id or sync slot. Again, there are block-transfer versions of these instructions.

The **invoke** instruction is used to perform a function invocation. If *dest_PE* indicates a different node, the SU will form a message and forward the request to the specified node. The receiving SU routes an 'invoke' signal or message directly to the EU, and the EU calls a frame allocator to get a free frame (whose size is determined by the function) from local memory and copies the parameters *params* (*no_params* specifies how many) into the frame. Lastly, the thread <*fp*, *f_name*> is executed; *fp* points to the new frame, and *f_name* is the address of the first instruction of the function's first thread. The **end_function** instruction deallocates the frame of the current function.

One concern we seek to address is that of automatic load balancing for applications in which a good task distribution cannot be determined statically at compile time. For such applications, EARTH provides an instruction in which the programmer can simply encapsulate a function invocation as a *token*. A token is sent to the SU, which puts the token on top of the local token queue. When there are no more ready thread ids in the ready queue, the SU removes a token from the top of the token queue and invokes the function as specified locally. Note that by 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.[4] When both the ready queue and token queue 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, extracts one from the bottom of its queue and 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.

2.3 The EARTH Threaded-C Language

Currently, programs running on EARTH are written in *EARTH Threaded-C*, which extends the C language with multithreading instructions. Most of these multithreading extensions directly correspond to EARTH instructions as described in section 2. Ultimately, EARTH Threaded-C will be used as an intermediate language for the McCAT-EARTH C compiler (see section 5), but it is also clean and powerful enough to be used as a user-level, explicitly parallel programming language. Figure 3 shows a simple example of a Threaded-C function that computes Fibonacci numbers using binary recursion.

The keyword THREADED specifies that this function's frame should be allocated from the heap and that the function may contain multiple threads.[5] This function contains two threads. Each thread, except for the first, begins with a label THREAD_*n*; the first thread begins at the start of the function, and is automatically executed when the function is invoked using an **invoke** or **token** instruction. Each thread ends with either END_THREAD(), which simply executes a

[3] Local destinations are permitted though they are less efficient than a memory copy operation. However, using a data transfer & synchronizations instruction even for local writes allows one to use the same code without knowing whether the destination is local or remote.

[4] A similar technique was first used in the Concert system [8].

[5] Non-threaded functions are specified as regular C functions.

```
THREADED fib (SLOT *done, long n, long *result)
{
    SLOTS SYNC_SLOTS[1];
    long sum1, sum2;

    if (n < 2) {
        DATA_RSYNC_L(1, result, done);
    } else {
        INIT_SYNC(0, 2, 2, 1);
        TOKEN(fib, SLOT_ADR(0), n-1, &sum1);
        TOKEN(fib, SLOT_ADR(0), n-2, &sum2);
        END_THREAD();
THREAD_1:
        DATA_RSYNC_L(sum1 + sum2, result, done);
    }
    END_FUNCTION();
}
```

Figure 3: Threaded-C example

fetch_next instruction, or END_FUNCTION(), which executes an **end_function** to deallocate the frame before executing a **fetch_next**.

The *fib* function has a single sync slot. Since *fib* uses simple binary recursion, it needs to invoke itself twice (if n>2) and add the results of both children. Therefore, the INIT_SYNC command sets the sync count to 2. The other arguments to this command are the slot number (0 in this case), the reset count (2) and the number of the thread to execute when the sync count reaches 0 (THREAD_1). Note that the sync count and reset count are the same. This is often the case, but is not required. The sync count could be initialized to a lower value, to indicate that some initial data is already available, or to a higher value, to force the thread to wait for additional events to occur during initialization.

The function takes two arguments in addition to n. These are pointers to the address where the result should be sent (result), and to the sync slot that receives the sync signal (done). (For coding efficiency, data and sync slot locations are represented in Threaded-C as single addresses rather than as <*fp*, *offset*> pairs.) Thus, frames will point back to their callers in a tree-like fashion, as in figure 2.

If n<2, the function sends a 1 to *result and signals the sync slot *done (atomically, using a **data_sync**).[6] Otherwise, it invokes two instantiations of itself, telling each to signal sync slot 0 (the macro SLOT_ADR(0) generates a pointer to the correct slot). The TOKEN command is used to invoke the load-balancing mechanism. The initial thread then terminates, and THREAD_1 will not be executed until both functions return their values and signal slot 0. Other threads in the ready queue can be executed in the interim. When THREAD_1 is activated, the addition is performed, the data is sent to this function's caller, and the frame is deallocated.

Note that thread boundaries can be located inside any compound statement, e.g., the if-statement. In this example, if n<2, the program will jump to the middle of THREAD_1 and execute the END_FUNCTION() command. Another example is shown in figure 4. This shows a fragment of a program in which elements of vectors stored on a remote processor are fetched using **get_sync** instructions. This code effectively performs split-phase transactions, because THREAD_2 will not start until both values have returned (assuming that slot 0 has sync and reset counts of 2). This code is purely sequential, but the time waiting for remote accesses can be put to use executing another thread.

[6] The suffix indicates the type of value being transferred, e.g., _L for long or _D for double.

```
THREAD_1:
    for (i = 0; i < size; i++) {
        GET_SYNC_D(a++, &la, 0);
        GET_SYNC_D(b++, &lb, 0);
        END_THREAD();
THREAD_2:
        DATA_SYNC_D(la + lb, res++, 1);
    }
    END_THREAD();
```

Figure 4: Split-phase transactions in Threaded-C

3 The EARTH-MANNA Platform

In this section we describe our emulation of the EARTH model on the MANNA multiprocessor. The MANNA 2.0 (Massively parallel Architecture for Non-numerical and Numerical Applications) architecture was developed at GMD-FIRST in Berlin, Germany [4]. Each processing node contains two 50-MHz i860XP processors, 32M bytes of DRAM, and a fast network interface. The network interface supports bidirectional data transfers at 50M bytes per second, for a total bandwidth of 100M bytes per second. The network is based on 16 $\times$ 16 crossbar switches, which support the full link bandwidth (a total of 1.6G bytes per second). This dual-processor architecture closely matches the EARTH model, permitting an efficient emulation of our model on the MANNA system.

The main goal of the EARTH-MANNA implementation is its use as a tool to study the EARTH model and an execution platform for code generated by the McCAT-EARTH C compiler. However, the good performance achieved by this system also allows it to be used as a multithreaded system in its own right, even though it is not perfectly suited to the task. As the results in section 4 demonstrate, the EARTH-MANNA implementation achieves nearly linear speedups on most of the benchmarks we ran.

A key to these good results is the close interaction between the EARTH Threaded-C compiler and the run-time system. In order to reduce development time without compromising on code quality, the Threaded-C compiler was constructed by adding a pre-processor and a post-processor to an existing commercial compiler for the i860 written by the Portland Group. The pre- and post-processor handle all the constructs specific to Threaded-C. The Threaded-C compiler only generates code for the EU; the second processor of each MANNA node is coded to implement the functionality of the SU, and this code is the same regardless of what code runs on the EU.

In table 2, we list the relevant EARTH operations and their costs on the EARTH-MANNA platform. Most of the costs were measured, but for the ones which weren't, they were estimated from the number of instructions required in the routines. The costs in the table do not take into account any cache misses as a result of executing the EARTH routines; a cache miss takes approximately 140ns to be serviced. The 'local EU overhead' is the cost for the EU to generate a signal packet and insert it into the event queue implemented in main memory. Such costs are deemed over-

EARTH Operation	local EU overhead	local SU costs	remote SU costs
Communications			
GET_SYNC local	780	300	0
GET_SYNC remote	780	500	800
DATA_SYNC local	780	300	0
DATA_SYNC remote	780	500	800
BLKMOV local	800	400+15/B	0
BLKMOV remote	800	900+25/B	700+25/B
BLKMOV request	0	500	500
Synchronizations and scheduling			
SPAWN local	780	200	0
SPAWN remote	780	500	700
SYNC local	700	200	0
SYNC remote	700	500	700
END_FUNCTION	750	0	0
END_THREAD	720	0	0
Function invocations and load balancing			
INVOKE local	800-1500	300-1000	0
INVOKE remote	800-1500	500-2000	500-2000
TOKEN	800-1500	300-1000	0
token request	0	500	500
token response	0	500-2000	500-2000
token expand	0	300-1000	0

Table 2: EARTH operation costs (ns.) on EARTH-MANNA

head because in an EARTH implementation which performs both the EU and SU tasks in one processor, such costs will not be incurred. The overhead is quite expensive since it requires a few main memory accesses (each memory access takes 140ns). The 'local SU costs' are the actual costs of the instructions used to perform the EARTH operations. 'Remote SU costs' are the costs to process the operation at a remote SU, where applicable. An operation labeled with 'local' implies that the operation can be fulfilled entirely on the local node, while ones labeled with 'remote' indicate an operation requiring interprocessor communications and work done by a remote SU.

For the BLKMOV costs, they are specified as some fixed cost plus x nsecs per byte in the block of data moved. For INVOKE and TOKEN costs, the smallest values are for a 1-parameter function while the largest values are for the maximum number of parameters (18 words). Token requests are issued by an SU when its EU is idle, token responses are responses to such requests, and token expand is the action of actually invoking the function within the token on the local node.

As we can see, the EU overhead for some simpler EARTH operations which can be fulfilled on the local node are more costly than the actual operations performed by the local SU. In other words, there is a penalty to use the SU for those operations. This points to the need for hardware support for the buffers between the EU and SU if efficient EARTH operations are to be realized. Nevertheless, the SU is still very beneficial as we illustrate in the next section.

4 Performance Results

In this section, we show that the EARTH-MANNA platform exhibits both good performance as a single node and good speedups (up to sixteen processors) without any changes to the Threaded-C versions of benchmark programs. We also give some preliminary results on the quantitative benefits of the SU in the EARTH-MANNA platform.

Benchmarks	Input size	T_{seq} (sec.)	Description
Tomcatv	$N = 257$	55.14	SPEC89; data-parallel
TSP	10 cities	38.2	Recursive search
Ray Tracing	512×512	273.9	Task-parallel
Protein Folding	$3 \times 3 \times 3$	7.43	Recursive search
N-Queens	12 queens	37.75	Recursive enumeration
Paraffins	$N = 22$	1.364	Recursive enumeration

Table 3: Benchmarks used

We used six common benchmarks, summarized in table 3. They range from 70 to 20,000 lines of EARTH Threaded-C. More detailed descriptions of the benchmarks and of their implementations in EARTH Threaded-C can be found in [12].

In table 3, the running time of each benchmark is listed 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).

4.1 Uni-Node Support and Multi-Node Speedups

To show the non-intrusiveness of multithreading support in the EARTH-MANNA platform (and thus the EARTH model), we propose to measure the **Uni-Node Support Efficiency (USE)** of the platform. USE indicates the efficiency of a program generated for multithreading running on a single processor node of a multithreaded machine.

$$\text{USE} = T_{seq}/T^1_{threaded}$$

where T_{seq}, as defined above, is the best *sequential* running time, and $T^1_{threaded}$ denotes the execution time of a program written in EARTH Threaded-C running on one processor node with multithreading support.

In our experiments, programs written in EARTH Threaded-C can be run directly on multiple processor nodes of the EARTH-MANNA without any code change. Of course, one may produce code which checks the number of processors assigned to the task and then simply branches to a version of the program which does not use any multithreading operations. However, our versions of the programs don't do this. They utilize multithreading operations as if they were run on multiple nodes; thus, the version running a single node would execute the same number of multithreading operations as if it were run on sixteen nodes. A USE value close to unity for such programs can be attributed to a combination of many possible reasons: 1) there are enough parallel threads to hide the latency of multithreading operations, i.e., there are enough threads for execution while multithreading operations in other threads 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); and 3) the support for multithreading operations is not intrusive on sequential code.

We ran the six benchmarks on one processing node of the EARTH-MANNA platform and obtained USE results for each (see table 4). The results show that even with multithreading instructions being executed in the uniprocessor version, the performance degradation is not overwhelming.

Benchmarks	USE	EU Cycles (%)		EARTH Operations	
		Compu-tation	Idle	Over-head (%)	Number
Tomcatv	1.07	93.8	5.76	0.42	286,828
TSP	0.97	99.0	0.20	0.76	761,376
Ray Tracing	0.99	100.0	0.00	0.00	178,184
Protein Folding	0.95	98.2	0.05	1.75	177,150
N-Queens	0.72	87.75	0.38	11.86	9,534,306
Paraffins	0.92	99.8	0.07	0.11	2,250

Table 4: Breakdown of uni-node execution times of the EU

One could say that this indicates 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 or in machines which require the processor in the node to handle all the multithreading chores as well as the computations.

In table 4, we can see that the Tomcatv benchmark obtained a super-unity USE value. This appears to be due to lower cache misses experienced by the EU. Such super-unity USE cases are subjects of another paper.

4.1.1 Analysis of Multithreading Overheads

To show how much multithreading operations affected the overall execution time, we show the breakdown of execution time (in cycles) as experienced by the EU for the multithreaded version of the programs running on a single node (table 4). To derive the % EU computations figure, we subtracted from 100% the percentage of EU idle time and the 'local EU overhead' costs. The overhead are the costs of EARTH operations as seen by the EU, which includes the time to perform context switches. As we can see, the multithreaded versions running on one node does not pay heavily for the multithreaded operations. Ideally, the fraction of computations performed by the EU (% EU computation cycles / 100) should be equal to the USE value if there is no intrusion on sequential computations within a thread, but due to the intrusion of multithreading it is generally more. (This does not apply when the USE value is greater than 1.)

For Ray Tracing, the intrusiveness of multithreading support is virtually non-existent; the ratio of EARTH operations to the total operations is virtually zero. For N-Queens, the large percentage of cycles attributed to EARTH operations explains why it has a low USE factor. However, it does not explain the large difference between the fraction of EU computations and the USE factor. On closer examination, we discovered that during the execution of N-Queens, 235 activation frames (containing instances of recursive functions), each of size 528 bytes, were simultaneously active at one time. Seeing that the on-chip data cache is 16kB, a 115+ kB working set size points to a high cache miss rate. We will confirm this when our simulator is fully functional.

For the other benchmark codes (not including Tomcatv), the sequential versions are benefitting from lack of interference from the SU. This interference is in the form of cache misses attributed to EARTH operations and the sharing of the single bus by the EU and SU. In fact, the sharing of the single bus caused Paraffins to have a noticeable difference between the USE and fraction of EU computations. We ran a multithreading version of Paraffins in which all multithreading operations were removed except for the declaration that the main function is THREADED. That is, the only overhead would simply be the SU periodically checking the LINK chip via the shared bus. That version produced a USE of 0.92. The reason is that Paraffins constantly writes to memory the viable paraffin structures and requires the full use of the shared bus. Therefore, bus accesses by the SU forces the EU to wait and slows down its computation rate. By examining how frequently the SU code accesses the bus to check the LINK chip, we determined that the SU bus bandwidth requirement is about 8%; thus explaining the difference. This implies that an actual EARTH implementation should consider separate buses for the EU and SU or a direct signal link between the SU and LINK chip.

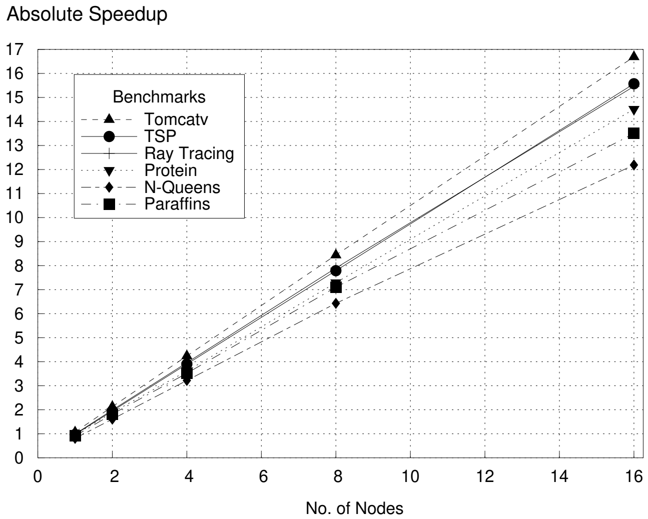

Figure 5: Absolute speedups

4.1.2 Speedups of the Multithreaded Programs

As we mentioned, the Threaded-C benchmark programs can be run directly on multiple nodes of the EARTH-MANNA platform. In figures 5 and 6 we show the absolute and relative speedups of the programs. The absolute speedup is defined as $(T_{seq}/T^{k}_{threaded})$ while relative speedup is $(T^{1}_{threaded}/T^{k}_{threaded})$, where k is the number of nodes. The results show that all the programs can scale nicely to sixteen processors (the largest configuration of the MANNA currently available), and that the speedups are good even when compared with sequential versions of the programs.

4.2 Quantitative Benefits of the SU

To measure the benefits of the basic multithreading support provided by the SU in the EARTH-MANNA platform, we performed a series of experiments with two of the benchmark programs running on sixteen nodes. For the benchmarks used, the original programs already perform certain amounts of split-phase data communications and remote synchronizations. In our experiments, we modified the programs to include more split-phase communications (which includes synchronizations) and tried to determine how much more communications the EARTH-MANNA platform could absorb before the speedups attained for the original versions of the programs diminished. Figures 7 and 8 show

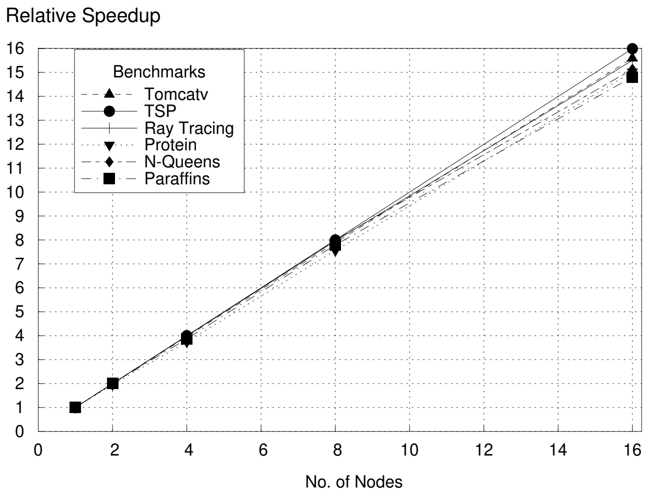

Figure 6: Relative speedups

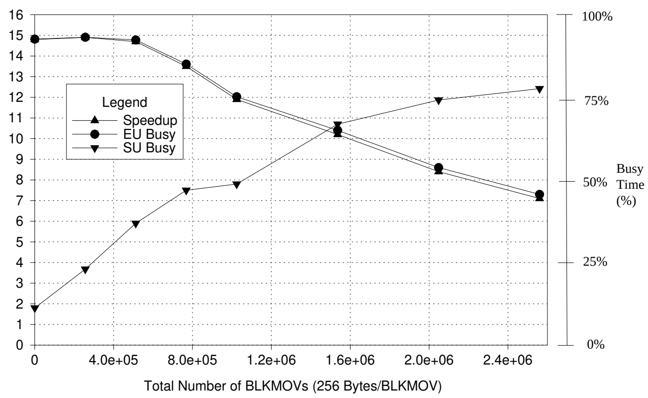

Figure 7: Benefits of the SU for Ray Tracing

the results of these experiments. It is clearly evident that the SU can handle a sizable increase in interprocessor communications before affecting the processing rate of the EU. The amount of increased communications over the original amount is roughly 2500-fold for Ray Tracing and about 10-fold for N-Queens before their speedup decrease by 10%. The original version of Ray Tracing requires 256 BLKMOVs of 256 bytes of data each, while N-Queens requires about 6000 BLKMOVs of 48 bytes of data each. Note that the benefit level of the SU is dependent on the amount of parallelism in the program as well as the original amount of communications and synchronizations performed. The huge absorption of increases in communications and synchronizations in Ray Tracing is due to a large amount of parallelism in the code, whereas the small amount of absorption in N-Queens can be attributed to the large overhead of EARTH operations (parallel processing overhead) in the code (see table 4). Our results imply that for parallel applications which have the same amount of parallelism as the benchmark programs, but are more communications-intensive, the EARTH-MANNA platform can produce the same speedups which we have attained. This is a good indication that architectures with a separate unit for handling communications and synchronizations will fare well.

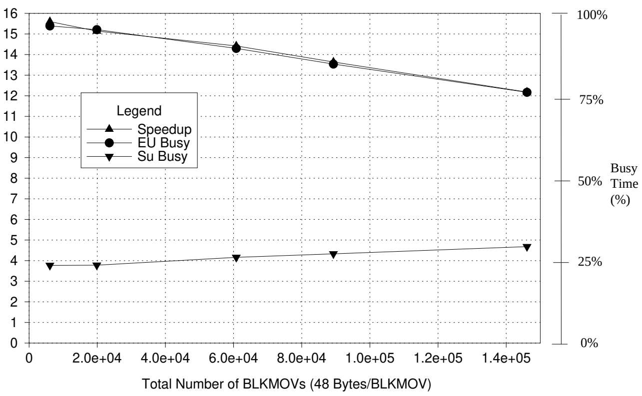

Figure 8: Benefits of the SU for N-Queens

5 Overview of the EARTH Project

Other efforts in our EARTH project include the definition of a more sophisticated EARTH to support caching of globally-shared data and speculative execution, construction of an EARTH simulator for gathering more detailed data, construction of the McCAT-EARTH C compiler which can automatically generate threads from a regular C program or a program written in a dialect of C which we call *EARTH-C*, and the VLSI implementation of EARTH. More details of these efforts can be found in [12].

Currently, the EARTH model does not support caching of globally-shared data in the nodes and must rely on a software layer to maintain coherence of shared data. For many applications, this may induce unnecessary synchronizations and communications which some hardware assist may circumvent. One of our recent efforts has been focused on providing a combined hardware/software approach to caching globally-shared data. We intend to add minimal hardware to conventional caches and employ the underlying split-phase transaction mechanisms in the maintenance of cached shared data [11]. Another effort is to provide prioritized ready thread ids to increase cache reuse [11].

Also, we intend to have support for algorithmic speculative execution. In speculative execution such as parallel OR-search, we need to support non-deterministic arrivals of completion signals from parallel tasks which were invoked to search for a solution. The arrival of the first completion signal signals the end of the search and the computation can progress to another phase. A preliminary solution for supporting algorithmic speculation in EARTH is outlined in [13] where the SU has the capability of enabling a thread by accepting the first of many signals destined for that thread. Subsequent signals are simply ignored.

Another effort on evaluation tools concerns the construction of a general-purpose EARTH simulator. The current EARTH-MANNA emulator can mimic the behavior of the SU with reasonable overheads, allowing experimentation with large benchmarks. Nevertheless, there are limitations. Some useful run-time statistics are impossible to gather without altering the timing behavior. Furthermore, the current configuration cannot accurately model the timing behavior of a hardware-based SU, making analysis of fine-grained programs difficult. Finally, the platform is limited by the number of available processors.

We are currently implementing a simulator which can model both the EARTH-MANNA platform and an EARTH with a specialized SU. We intend to collect more detailed information (e.g., cache misses, bus contentions, etc.), as well as create hypothetical SU mechanisms for experimentation. The simulator currently simulates EARTH-MANNA at 1/300 speed on a Sparc-20, and is presently being tested for correctness and timing accuracy.

A large and significant aspect of our project involves the automatic generation of EARTH threads from C programs. We are using the McCAT compiler [9] as the substrate for our McCAT-EARTH C compiler which can accept either C programs or programs written in a parallel dialect of C called *EARTH-C*. The EARTH-C language allows a programmer to specify high level (algorithmic) parallelism and data distributions while the compiler is still responsible for identifying and generating the low-level threads required in EARTH Threaded-C. A first version of the McCAT-EARTH C compiler, in which simple thread generation heuristics are employed, is currently operational, and we intend to expand its capabilities in producing threads with characteristics required by EARTH.

Lastly, our project involves an actual implementation of an EARTH machine in hardware. We are investigating the construction of EARTH nodes based on an off-the-shelf RISC processor which serves as the EU and a customized ASIC for the SU. Moreover, the node architecture is a two-bus system where the EU and SU each have separate memory buses to the shared local memory. Hardware queues will be implemented as well as an access controller which regulates accesses to local memory and which can tie the two buses together on demand. Off-the-shelf research chips will be used for the interprocessor networking. We are currently specifying the functionalities of the hardware devices in Verilog on Cadence and Synopsys.

6 Related Work

The RWC-1 project [18] 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.

Tera Corporation's MTA machine [2] interleaves the execution of multiple threads on one processor node, which EARTH does not do. Moreover, the Tera MTA has a separate memory subsystem for handling I-structure accesses. Tera Corp. intends to build custom processors to allow for the interleaving of threads.

In the Alewife Project [1] of MIT, each processor node is built around a modified Sparc processor, called Sparcle, which has special instructions for utilizing the register windows as if they were register sets dedicated to a function instance. A node in Alewife contains mechanisms for maintaining cache coherency and full/empty bits on memory locations.

The P-RISC [15] proposal explored the possibilities of constructing a multithreaded architecture around a RISC processor. The major difference is that no distinct synchronization unit was presented in that design; rather, augmentations to a RISC processor were proposed.

The division of the processor node into separate synchronization and execution units distinguishes our EARTH model from models such as TAM [5], in which all operations are performed on one processor. For instance, the EARTH model tries to isolate as much as possible the manipulation of synchronization information from the EU and provides special operations performed by the SU. Moreover, the TAM model relies solely on the tree of activation frames to represent multiply active function/loop instances, and that each activation frame has reserved space for maintaining a list of executable threads associated with the frame. The list of executable threads in EARTH is kept in hardware and software. Lastly, the TAM model allows inlets to be user programmable for flexibility, whereas the EARTH model restricts its 'inlets'—operations processed by the SU—to predefined functions. This 'inflexibility' in the EARTH model allows the SU to be readily specialized. Moreover, our SU will contain a simple RISC core so that some SU operations can be customized.

Spertus et al [19] analyzed the efficiency of fine-grain support mechanisms on both the J-Machine and a TAM implementation on Thinking Machine's CM-5. In the J-machine, like the TAM implementation, a processor handles computations, synchronizations, runtime scheduling, and processing split-phase requests. Their analysis hinted that a coprocessor for handling remote requests (such as our SU) may be interesting to examine.

*T [16] is a proposed multithreaded machine which has separate synchronization and execution units in a processor node. A key difference between EARTH and *T is that the synchronization unit is user-programmable in the *T—it is another microprocessor as in our EARTH-MANNA platform. The user has some control over synchronization in EARTH, since the sync slots are definable by the EU, but we opted for a non-user-programmable SU, thus removing one source of deadlocks which can be introduced by user code. A follow-on architecture called *T-NG [3] 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 PowerPCs are used for cache consistency in a site, while the tasks for cache consistency between sites are relegated to one of the four PowerPCs in a site. The same processor dedicated to inter-site cache consistency is responsible for handling split-phase memory requests and synchronizations. A major difference between EARTH and *T-NG is that the latter, like *T, uses an off-the-shelf microprocessor for its synchronization unit. It will be interesting to compare *T-NG to EARTH-MANNA once they have experimental results.

The major difference between EARTH and most of the above architectures is that an off-the-shelf microprocessor is combined with an ASIC device for performing synchronizations and remote accesses. The division of labor between EU and SU allows greater processor utilization, and the use of a stock processor for the EU cuts down on design costs and enables us to exploit the latest in RISC technology.

7 Conclusion

We have presented the EARTH—a multithreaded architecture model where each node is constructed with an off-the-shelf high-performance microprocessor as the execution unit (EU) and a synchronization unit (SU) which supports (dataflow-like) thread synchronizations, dynamic scheduling and load balancing. The SU is designed to hide long and unpredictable latencies arising from interprocessor communications and synchronizations. Our EARTH-MANNA platform shows that an "emulation" methodology is quite effective in exploratory work since we can build a platform in three months for executing large applications quickly (e.g., a typical run of Ray Tracing takes 200 seconds).

We have implemented a number of benchmark programs to be executed on the EARTH-MANNA emulation platform, and our preliminary experimental results have demonstrated that efficient multithreading can be realized without major intrusions to the execution efficiency of sequential code within a thread. Moreover, we have collected USE factors demonstrating that a multithreading architecture is not necessarily inefficient in a uniprocessor context. Lastly, we have provided data of the latency tolerating capabilities in the EARTH-MANNA (and thus the EARTH design) which show that marked increases in communication and synchronization latencies are ably hidden.

8 Acknowledgment

We would like to thank the Natural Sciences and Engineering Research Council (NSERC) for their support of this research. Also, some authors received funding from FCAR, and the Concordia FRDP. Lastly, we would like to thank Jack Dennis for his valuable comments on an earlier draft.

References

[1] Anant Agarwal, Beng-Hong Lim, David Kranz, and John Kubiatowicz. APRIL: A processor architecture for multiprocessing. In *Proc. of ISCA-17*, pp. 104–114, Seattle, Wash., May 1990.

[2] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. In *Conf. Proc., Intl. Conf. on Supercomputing*, pp. 1–6, Amsterdam, The Netherlands, Jun. 1990.

[3] 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.

[4] U. Bruening, W. K. Giloi, and W. Schroeder-Preikschat. Latency hiding in message-passing architectures. In *Proc. of the 8th Intl. Parallel Processing Symp.*, pp. 704–709, Cancún, Mexico, Apr. 1994.

[5] 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 ASPLOS-IV*, pp. 164–175, Santa Clara, Calif., Apr. 1991.

[6] Jack B. Dennis and Guang R. Gao. An efficient pipelined dataflow processor architecture. In *Proc. of Supercomputing '88*, pp. 368–373, Orlando, Flor., Nov. 1988.

[7] Guang R. Gao, Herbert H. J. Hum, and Jean-Marc Monti. Towards an efficient hybrid dataflow architecture model. In *Proc. of PARLE '91*, volume I of *LNCS 505–506*, pp. 355–371, Eindhoven, The Netherlands, Jun. 1991. Springer-Verlag.

[8] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Trans. on Programming Languages and Systems*, 7(4):501–538, Oct. 1985.

[9] L. Hendren, C. Donawa, M. Emami, G. Gao, Justiani, and B. Sridharan. Designing the McCAT compiler based on a family of structured intermediate representations. In *Proc. of the 5th Intl. Work. on Languages and Compilers for Parallel Computing*, number 757 in LNCS, pp. 406–420, New Haven, Conn., Aug. 1992. Springer-Verlag. Published in 1993.

[10] H. H. J. Hum and G. R. Gao. A high-speed memory organization for hybrid dataflow/von Neumann computing. *Future Generation Computer Systems*, 8(4):287–301, Sep. 1992.

[11] Herbert H. J. Hum and Guang R. Gao. Supporting a dynamic SPMD model in a multi-threaded architecture. In *Digest of Papers, COMPCON Spring '93*, pp. 165–174, San Francisco, Calif., Feb. 1993.

[12] 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 Nemawarkar, Prakash Panangaden, Xun Xue, and Yingchun Zhu. The Multi-Threaded Architecture multiprocessor. ACAPS Tech. Memo 88, Sch. of Comp. Sci., McGill U., Montréal, Qué., Dec. 1994. Available via anonymous ftp://ftp-acaps.cs.mcgill.ca/pub/doc/memos.

[13] 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.*, pp. 288–294, Cancún, Mexico, Apr. 1994. Expanded version publ. as ACAPS Tech. Memo 68, Sch. of Comp. Sci., McGill U., Montréal, Qué., Oct. 1993. Available via anonymous ftp://ftp-acaps.cs.mcgill.ca/pub/doc/memos.

[14] 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.

[15] Rishiyur S. Nikhil and Arvind. Can dataflow subsume von Neumann computing? In *Proc. of ISCA-16*, pp. 262–272, Jerusalem, Israel, May–Jun. 1989.

[16] R. S. Nikhil, G. M. Papadopoulos, and Arvind. *T: A multithreaded massively parallel architecture. In *Proc. of ISCA-19*, pp. 156–167, Gold Coast, Australia, May 1992.

[17] James Rumbaugh. A data flow multiprocessor. *IEEE Trans. on Computers*, 26(2):138–146, Feb. 1977.

[18] 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., Intl. Conf. on Supercomputing*, pp. 251–260, Tokyo, Japan, Jul. 1993.

[19] 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 ISCA-20*, pp. 302–313, San Diego, Calif., May 1993.