McGill University
School of Computer Science
ACAPS Laboratory

Advanced Compilers, Architectures
and Parallel Systems

The Multi-Threaded Architecture 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 Nemawarkar*
Prakash Panangaden *Xun Xue* *Yingchun Zhu*

ACAPS Technical Memo 88
December 9, 1994

A version of this paper appears in the Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT 95), Limassol, Cyprus, June 27–29, 1995.

†Dept. of Electrical and Computer Engineering, Concordia University,
1455 de Maisonneuve W., Montreal, Canada, H3G 1M8.

ACAPS • School of Computer Science • 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 the synchronization and communication latencies, without intruding on the performance of sequentially-executed code? And 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 the MTA (Multi-Threaded Architecture) model which attempts to address the above issues. We have implemented the MTA model on an experimental high-performance multiprocessor testbed called MANNA 2.0. The MTA-MANNA platform has for each processing node, an Execution Unit (EU) for performing the computations and is programmed as a standard superscalar processor, and a Synchronization Unit (SU) supporting dataflow-like thread synchronizations and scheduling, as well as processing remote memory requests. We demonstrate that multithreading support can be efficiently implemented into 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, performance measurements, MTA.

Contents

List of Figures

List of Tables

1 Introduction

There are two general approaches to designing parallel processing architectures. The first is to 'retrofit' an existing uniprocessor system with parallel processing mechanisms such as interprocessor communication devices and the second approach is to design a parallel system from the ground up, making such parallel processing mechanisms an inherent part of the architecture. The first approach has led to quick product releases in the marketplace, but users often fail to achieve the speedups they were expecting because the system designs do not adequately address the inherent latencies of parallel processing, such as interprocessor communication and synchronization overheads. On the other hand, 'retrofitted' machines retain the efficiency of uniprocessors for sequential computations, an efficiency which is often sacrificed in the second approach. Achieving good performance on general applications will require an architecture with both characteristics: it should be capable of tolerating the latencies due to synchronization and communication, without intruding on the performance of sequentially-executed code.

Multithreaded architectures [1, 2, 4, 14, 19] 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.

On another note, in most multithreaded architecture projects known to us, the focus has been on the demonstration of their advantages in terms of speedup, scalability, etc. in multiprocessing, but with less attention to the *quantitative benefits of multithreading support.* That is, how much can interprocessor communication/synchronization latencies be hidden while still attaining the expected speedups?

In this paper, we address both the issue of non-intrusiveness of multithreading support, and the benefits of hiding communication/synchronization latencies, by presenting our *Multi-Threaded Architecture (MTA)* model [11, 13] and its implementations. Each node in an MTA 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 designed using ASIC devices, and can be optimized for synchronization and communication tasks, relieving the RISC processor of these responsibilities. The main features of the MTA which support multithreading are:

- Augmenting the existing RISC instruction set with MTA 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 MTA instructions are implemented directly using native instructions, so that existing processors can be used.)

- A dedicated coprocessor 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.

Thus, we are taking the 'retrofitting' approach to parallel processing, but in a way which gives good support to synchronization and latency without losing uniprocessor performance. As we demonstrate later, the retrofitting of a uniprocessor system goes beyond simply supplying an interprocessor communications chip and an SU—it requires a fast means of communications between the EU and SU. For example, in our implementation of the MTA model on an experimental high-performance multiprocessor testbed called MANNA 2.0 (to be detailed in section 3), communications from the EU to SU take 780ns (a software queue is implemented in memory to buffer the communications) while simple MTA operations only require 200 to 300ns.

In this MTA-MANNA platform (emulation system), each MANNA node has two Intel i860XP processors apiece, which we dedicate to the EU and SU tasks, respectively. Due to the excellent match between the MANNA 2.0 architecture and the MTA model, and because MTA operations are directly supported by our compiler, the MTA emulation on MANNA can be regarded as a multithreaded system. As the results in section 3 and section 4 demonstrate. This system permits us to run large benchmark programs efficiently, which is essential for our quantitative study. To measure the benefits of the basic multithreading support provided by the SU in the MTA, we have instrumented the MTA-MANNA emulator to count the number of occurrences of various multithreading operations. By measuring the number of remote communications/synchronizations performed, we can determine the latency hiding ability of our MTA-MANNA platform.

We have developed a set of extensions to the C language which allow programmers to express the MTA functions directly in the code. Using this system, we have implemented several benchmarks. We performed an experiment to gauge the 'absorption capability' of increased communication/synchronization demands in the MTA-MANNA without affecting the overall execution time of benchmarks. From the preliminary data we have collected on a couple of our benchmark programs, we show that the SU can handle about a 10-fold increase in interprocessor communications/synchronizations over the original amount required with only a small degradation in performance (<10%)—the amount of tolerance obviously depends on the parallelism in the program as well as the original amount of communications performed. Moreover, we 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 processor node of the MTA as compared to programs optimized for sequential execution running on one processor. For the benchmark programs we examined, we obtain a range of USE values from 72% to 107%.[1] 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 MTA-MANNA platform. Lastly, 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.

1.1 Synopsis

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

2 The Multi-Threaded Architecture Model

In the MTA model, a multiprocessor consists of multiple MTA nodes and an interconnection network. The MTA node architecture is derived from the McGill Data Flow Architecture [6], which was based on the idea that synchronization of operations and the execution of the operations themselves could be performed more efficiently in separate units rather than in the same processor. Each MTA node consists of an *Execution Unit* (EU) and a *Synchronization Unit* (SU), linked together by buffers (see Figure 1). The SU and EU share a local memory, which is part of a *distributed shared memory* architecture in which the aggregate of the local memories of all the nodes represents a global memory address space.

The EU processes instructions in an active thread, where an active thread 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. It interacts with the SU and the network by placing messages in the 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.

[1]The super unity USE value phenomena is a subject of another paper.

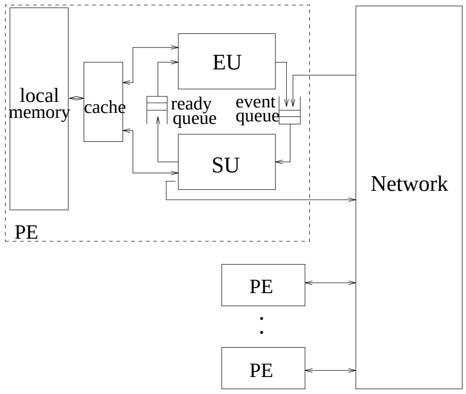

Figure 1: The Multi-Threaded Architecture.

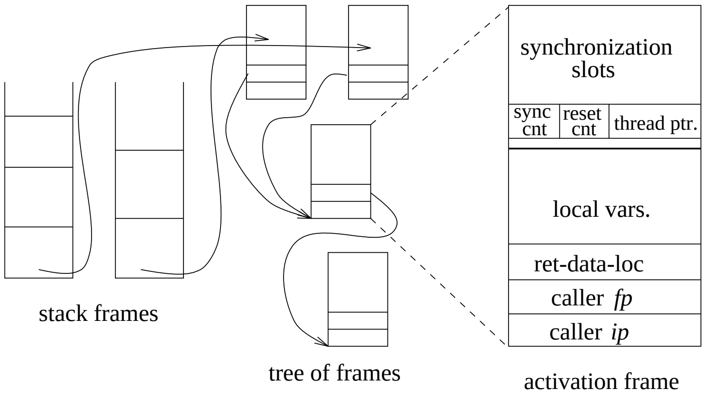

Figure 2: Stack and activation frames.

2.1 Runtime Model

The MTA runtime model is similar to the runtime models of some other multithreaded machines in that it uses a *tree of activation frames* to represent the program state. Each frame, which is dynamically allocated from a heap, represents the instantiation of one function, and contains the local state of that instantiation, including local variables. A frame pointer *fp* points to the base of the frame and is used to identify the frame uniquely. For efficiency, sequential, single-threaded functions that only call other single-threaded functions can use conventional stacks for function invocation, which are integrated into the tree structure as shown in figure 2.

A function containing one or more threads, each of which is a section of code synchronized as an atomic unit. Dataflow-like synchronization is achieved using a mechanism described below. A thread can be identified by a *thread id* <*fp, ip*> consisting of a frame pointer and an instruction pointer (ip). The latter is the address of the first executed instruction of the thread.

An activation frame has three main components. The first links an activation frame to its caller, i.e., its pparent in the activation tree. Another group of slots in the frame is used to store values local to the activation, e.g., input parameters for the function call, values to be passed between two threads of the function instance, etc. Lastly, there are *synchronization slots* which are used to control the enabling of threads in the current function invocation. Three slots in the frame are reserved for function linkage: the *caller IP*, *caller FP*, and *ret-data-loc* slots. The caller IP is used to indicate which thread in the caller function to initiate upon returning, and the caller FP identifies the activation frame of the caller function instance. The ret-data-loc slot contains a pointer to a slot in the caller's activation frame where a return value is to be deposited When the callee terminates, i.e., it does a 'return' instruction, the callee's activation frame is deallocated.

Each synchronization slot contains three fields: a synchronization count (*sync cnt*), a reset count, and a thread pointer. The sync count indicates the current number of synchronization events required to enable the thread, the reset count gives the *total* number of synchronization events, and the thread pointer indicates which thread is to be activated when the sync count reaches 0, i.e., enough synchronization events have occurred.

2.2 Instruction Set of the MTA

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 instructions, register-register operations, and control operations, augmented with instructions for multithreading (fig. 3). In order for the EU to be implemented with an off-the-shelf processor, the multithreading instructions will actually be encoded using instructions native to the RISC processor. They will probably be encoded as memory-mapped I/O operations that will be recognized and processed by the SU hardware.

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 will be used to fetch the first instruction of the next thread.

The EU processes **spawn**, **sync**, and **incr_sync** instructions by generating spawn, sync and incr_sync signals, respectively, to the SU. Signals to the SU contain the parameter values as specified in the instructions. If these signals were destined for a remote processor—by checking the processor id in *fp*—then the signals would be packaged as messages and sent out on the net. 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 at (*fp* + *ss_off*). If the decrement produces a 0, then <*fp*, *ip*> is put onto the ready queue where *ip* is the thread pointer in the synchronization slot, and the sync count is replaced with the reset count value in the slot—-resetting the sync count prepares the thread for its next activation. An incr_sync signal instructs the SU to add

Instruction Set
Arithmetic, Logic and Branching
typical RISC instructions, e.g., those from the i860 architecture
Thread Switching
fetch_next
Synchronization
spawn *fp*, *ip* **sync** *fp*, *ss_off* **incr_sync** *fp*, *ss_off*, *value*
Data Transfer & Synchronization
data_spawn *addr_val*, *value*, *fp*, *ip* **data_sync** *addr_val*, *value*, *fp*, *ss_off* **blockdata_spawn** *addr_val*, *size*, *values*, *fp*, *ip* **blockdata_sync** *addr_val*, *size*, *values*, *fp*, *ss_off*
Split-Phase Data Requests
get_spawn *addr_val*, *dest_addr*, *ip* **get_sync** *addr_val*, *dest_addr*, *ss_off* **get_block_spawn** *addr_val*, *dest_addr*, *size,ip* **get_block_sync** *addr_val*, *dest_addr*, *size*, *ip*
Function Invocation
invoke *dest_PE*, *f_name*, *f_size*, *caller_fp*, *caller_ip*, *ret_data_loc*, *no_params*, *params* **token** *f_name*, *f_size*, *caller_fp*, *caller_ip*, *ret-data-loc*, *no_params*, *params*

Figure 3: Instruction set of MTA.

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 the case where the number of synchronization signals destined for a sync count can only be determined at runtime, such as 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. However, they can also be used locally, which is useful in cases where both local and remote copies of the same sync slot can be accessed from the same function. The destination processor id of a message is extracted from *fp*. For remote destinations, the SU forms appropriate messages containing the parameters, and sends them over the network. These instructions have block-transfer versions, which are used when a large block of data is to be transferred and synchronized as a single unit.

The **get_spawn** and **get_sync** instructions are used to retrieve data (usually remote) in a *split-phase manner.* If the address parameter *addr_val* indicates a remote accesses, 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 operation is performed within the frame that issued the original instruction (thus, *fp* is not needed as a parameter, though it is added to messages by the SU). 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 right node. To process an 'invoke' command or message, the SU calls a frame allocator to get a free frame of size *f_size* from local memory and copies *caller_fp*, *caller_ip*, and *ret_data_loc* into the first three slots of a new frame pointed to by *fp'*. Then the parameters (*params*)—*no_params* specifies the number of parameters—are copied into the next successive slots following the function linkage information. Lastly, the ready thread id <*fp'*, *f_name*> is put onto the ready queue; the function name *f_name* is simply the address of the first instruction of the function's first thread.

A 'return' instruction functions is simply a macro consisting of a **data_spawn** instruction and a call to the frame deallocator routine.

One concern we seek to address is that of automatic load balancing for applications in which a good task distribution cannot be determined statically. For such applications, MTA provides an instruction in which the programmer can simply encapsulate a function invocation as a *token.* A token message 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 would remove a token from the top of the token queue and invoke 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. When both the ready queue and token queue are empty, the SU would send a message to a neighbouring processor requesting a token, in effect performing *work stealing.* The neighbouring processor, if it has tokens in its queue, would extract one from the bottom of its queue and send 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.

3 The MTA-MANNA Platform

In this section we describe our emulation of the MTA 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. Each processing node contains two 50-MHz i860XP processors, 32M bytes of DRAM, and a fast network interface. This dual-processor architecture closely matches the MTA model, permitting an efficient emulation of our model on the MANNA system.

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 16×16 crossbar switches, which support the full link bandwidth. For machines with more than 16 nodes, the network can be organized as a hierarchy of crossbars. Communication within a node, which is necessary in order to implement the ready and event queues of the MTA model, goes through the local memory, which is shared by both CPUs.

3.1 The MTA Threaded-C Language

Currently, programs run on MTA-MANNA are written in *MTA Threaded-C*, which extends the C language with multithreading instructions. Most of these multithreading extensions directly correspond to MTA instructions as described in section 2. Ultimately, MTA Threaded-C will be used as an intermediate language for the McCAT-MTA C compiler, but it is clean and powerful enough to be used also as a user-level, explicitly parallel programming language. As an example, figure 4 shows a vector addition subroutine written in Threaded-C. This code is purely sequential and is not tuned for performance, but it illustrates several distinctive features of MTA Threaded-C.

The keyword THREADED specifies that this function contains multiple threads. At the beginning of the function, the synchronization slots are initialized by the 'INIT_SYNC's. The first slot (0) is initialized to start thread 1 after 2 synchronization signals have been received, while the second slot is set to start thread 2 after *size* signals have arrived, in other words, a barrier synchronization. At the end of the function, the SYNC operation signals to a synchronization slot address provided by the caller that the function has terminated. The END_FUNCTION function deallocates the activation frame and fetches the next ready thread for execution.[2]

A feature of MTA Threaded-C is that multiple threads can be specified within the same function. This makes it easy to share information among such threads and makes the code more

[2]For efficiency, the automatic passing of function linkage information as specified in the MTA abstract model is not implemented since not all functions send the same amount of data back to the callee. Also, values are not returned automatically within END_FUNCTION, but is left to the programmer to perform a DATA_SYNC.

```
THREADED vadd (SLOT *done, int size,
               double *a, double *b, double *res)
     {
         SLOTS SYNC_SLOTS [2];
         int i;
         double la, lb;

         INIT_SYNC (0, 2, 2, 1);
         INIT_SYNC (1, size, size, 2);
         for (i = 0; i < size; i++) {
             GET_SYNC_D (a++, &la, 0);
             GET_SYNC_D (b++, &lb, 0);
             END_THREAD ();
     THREAD_1:
             DATA_SYNC_D (la + lb, res++, 1);
         }
         END_THREAD ();
     THREAD_2:
         SYNC (done);
         END_FUNCTION ();
     }
```

Figure 4: Threaded-C Example

readable, Also, thread boundaries can be located inside any C compound statement, e.g., a for-loop. In this particular example, execution of the function stops after the two GET_SYNC_D operations, which fetch remote data into a local variable (the '_D' suffix specifies a double word to be fetched). When the two values have arrived, the DATA_SYNC_D operation is performed, which stores the result to the remote location. The processor then directly executes the next loop iteration, switching context again after the two GET_SYNC_D's.

3.2 MTA-MANNA Performance

The main goal of the MTA-MANNA implementation is its use as a tool to study the MTA model and an execution platform for code generated by the McCAT-MTA 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 MTA-MANNA implementation achieves nearly linear speedups on many of the benchmarks we ran. A key to these good results is the close interaction between the MTA Threaded-C compiler and the run-time system.

In order to cut down development time without compromising on code quality, the Threaded-C compiler is based on an existing commercial compiler for the i860 written by the

MTA Operation	*Sequential Local*	*Sequential Remote*	*Pipelined Local*	*Pipelined Remote*
SYNC	2424 ns	4104 ns	841 ns	994 ns
SPAWN	2354 ns	4286 ns	2003 ns	1570 ns
GET_SYNC	2648 ns	7109 ns	1137 ns	1908 ns
DATA_SYNC	2560 ns	6458 ns	1060 ns	1749 ns
INVOKE	5296 ns	9216 ns	3188 ns	2792 ns

Table 1: Execution time of some MTA operations

Portland Group. Their C compiler is used in conjunction with a pre- and post-processor which handle all the constructs specific to Threaded-C. The bulk of the work is performed by the post-processor, which handles the thread-specific register saving conventions, dynamic frame allocation, context switching, etc. In particular, an analysis is performed to determine exactly which registers are needed or modified by each thread and only these registers are then saved or restored at thread boundaries.

The second processor of each MANNA node implements the functionality of the SU. The complete communication kernel which runs on the second processor (SU) is written in assembly language in order to take full advantage of all processor resources. Table 1 shows the time needed on the average to perform some of the more important MTA operations.

The data shown in table 1 distinguishes between four different cases. Either the operation can be performed locally, e.g., when a function is invoked on the same node, or it is remote and therefore necessitates communication with another node. The sequential times represent the case where a single operation is issued and the Execution Unit then waits on its completion. While in the pipelined case, a large number of operations are issued without waiting for their termination. This results in lower delays per operation, as several operations can be in progress at the same time.

Even though these numbers are quite good compared to current state-of-the-art multiprocessors, it is obvious that performing a remote read operation, or more importantly using an MTA operation to perform a local memory read is still about an order of magnitude slower than an actual DRAM access ($\approx$ 200ns). This is mainly due to the overhead associated with communication between EU and SU.

In table 2, we list the relevant MTA operations and their costs on the MTA-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 MTA routines; a cache miss takes approximately 200ns 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

MTA Operation	local EU overhead (ns)	local SU costs (ns)	remote SU costs (ns)
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: Costs of MTA operations in the MTA-MANNA platform.

are deemed overhead because in an MTA implementation on a system in which both the EU and SU functionalities are performed on 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 200ns). The 'local SU costs' are the actual costs of the instructions used to perform the MTA 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 MTA 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 for efficient MTA operations. Nevertheless, the SU is still very beneficial as we illustrate in the next section.

4 Performance Results

In this section we first describe the five benchmark programs coded in MTA Threaded-C running on the MTA-MANNA platform. Next, we show that the benchmark programs exhibit good performance on a single node and also show good speedups up to sixteen processors without any changes to the Threaded-C versions of the programs. Lastly, we give some preliminary results on the quantitative benefits of the SU in the MTA-MANNA platform.

4.1 Benchmark Description

Benchmark programs used in our study range from 70 to 20,000 lines of C code. Each benchmark was transformed from C to MTA Threaded-C by hand.

Ray Tracing—The Persistence of Vision Raytracer creates a three-dimensional, photo-realistic image using the ray tracing rendering technique. It reads in a text file that describes the objects and lighting in a scene and generates an image of that scene from the view point of a camera also described in the text file. We generated 512×512-pixel images to generate the speedup curves and 256×256 images for collecting other statistics. This program has at its core a doubly-nested loop which iterates over each pixel. The MTA Threaded-C implementation performs this loop in parallel. As the amount of work necessary for the computation of each pixel can vary widely, this implementation takes advantage of the MTA dynamic load balancing mechanism to get good processor utilization. The token operation is employed to encapsulate function calls which compute pixel values in one scan line of the image, and such *tokenized* functions are distributed using the work-stealing algorithm.

Protein Folding—This chemistry application finds all the possible polymers of an $n \times n \times n$ cube (n is the number of points on an edge of the cube, 3 in our experiments), where a polymer is defined as a chain of monomers and each monomer may represent a number of amino acids. It is assumed that each monomer is situated on a lattice point (i.e., a point on a 3D grid). Each possible folding of the polymer can then be described as some path along the set of lattice points. Two kinds of parallelism are exploited in this program. The first is is the loop-level parallelism exploited for all possible start paths provided in the original data-set using a dynamic token-driven mechanism. Another is the token-level medium-grained parallelism exploited at each lattice point for finding possible foldings using tail recursive and token mechanisms.

Tomcatv—This floating-point SPEC89 benchmark was chosen to demonstrate that the MTA could handle large data-parallel applications. From an initial pair of 257×257 meshes, each iteration updates the meshes, first using independent nearest-neighbor calculations, then by performing calculations with horizontal loop-carry dependencies. Iterations continue until the maximum change drops below a threshold. Since the MTA doesn't allow functions to be distributed across multiple processors, a separate function activation is created for each row. Static partitioning is done using a simple algorithm. Separate rows synchronize with each other and the top-level function using a pure dataflow paradigm.

Paraffins—The purpose of Paraffins is to enumerate all distinct isomers of each paraffin of size up to n. Paraffins are molecules with chemical formula C_nH_{2n+2} ($n > 0$). The number of paraffins grows exponentially with n ($n = 20$ in our experiments). A paraffin is essentially an unrooted 4-ary tree; thus the problem of generating distinct paraffins is the same as the problem of detecting isomorphism in labeled free trees. The program generates lists of paraffins in a pointer data structure and finally returns an array filled with the number of distinct paraffins of each size up to and including the maximum size specified by the user. The parallelism is exploited by invoking functions on all the processors to compute the radicals (basic reaction units to form larger molecules) and then forming tokenized functions which compute the paraffins for the required sizes.

N-Queens—This familiar benchmark typifies searching problems and is used to determine the number of ways in which N queens can be placed on an $N \times N$ chess board so that no queen may attack another. N for our runs is set to 12. Our implementation attacks this problem by expanding the number of boards first in a breadth-first manner, then switching to a depth-first traversal of the rest of the state space. The amount of breadth-first expansion depends on the number of processors and problem size. In the MTA Threaded-C implementation of N-Queens, both breadth-first and depth-first parallelisms are exploited with a threshold controlling mechanism on the breadth and depth of the search for generating medium-grained tokens.

4.2 Uni-Node Support and Multi-Node Speedups

To show the non-intrusiveness of multithreading support in the MTA-MANNA platform (and thus the MTA 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} denotes the execution time of a program optimized for sequential execution running on one processor node having no multithreading support, and $T^1_{threaded}$ denotes the execution time of a program written in MTA Threaded-C running on one processor node with multithreading support.

Benchmarks	T_{seq}	$T^1_{threaded}$	USE
Ray Tracing	273.9s	277.75s	0.99
Protein Folding	7.43s	7.86s	0.95
Paraffins	0.283s	0.356s	0.80
Tomcatv	55.14s	51.5s	1.07
N-Queens	37.75s	52.78s	0.72

Table 3: USE values.

In our experiments, programs written in MTA Threaded-C can be run directly on multiple processor nodes of the MTA-MANNA without any code change. Granted one may produce code which checks the number of processors assigned to the task and then simply vector to a version of the program which does not use any multithreading operations. However, our versions of the programs do not, they utilize multithreading operations as if they were run on multiple nodes. A USE value close to unity for such programs would indicate that: 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 be minimal if the parallelism is coarse-grain enough such that few multithreading operations are performed), and 3) the support for multithreading operations is not intrusive on sequential code.

We ran the five benchmarks on one processor node of the MTA-MANNA platform and obtained USE results for each (table 3). The results show that even with multithreading instructions being executed in the uniprocessor version, the performance degradation is not overwhelming. One can say that this indicates that the Threaded-C versions of the programs were well-written to minimize the use of multithreading operations when running on a single node, but what it also indicates is that the multithreading support does not have a huge negative impact on sequential code 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 the table, we can see that the tomcatv benchmark obtained a super unity USE value. The increase in performance appears to be due to lower cache misses experienced by the EU. Such super-unity USE cases are subjects of another paper.

To show how much multithreading operations affected the overall execution time, we show the breakdown of execution time as experienced by the EU for the multithreaded version of the programs running on a single node (fig. 4). To derive this figure, we used the 'local EU overhead' costs for the overhead cost of MTA operations as seen by the EU; the overhead 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 computations / 100) should be equal to the USE value if there is no

Benchmarks	Computations (%)	Idles (%)	Overhead of MTA Operations (%)	No. of MTA Operations	USE
Ray Tracing	100.0	0.00	0.00	178,184	0.986
Protein Folding	98.2	0.05	1.75	177,150	0.95
Paraffins	99.4	0.3	0.3	1,759	0.80
Tomcatv	93.8	5.76	0.42	286,828	1.07
N-Queens	87.75	0.38	11.86	9,534,306	0.72

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

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 or equal to one.) As we can see, for raytracing, the intrusiveness of multithreading support is virtually non-existent. For the other benchmark codes (not including tomcatv), the sequential versions are benefitting from the global optimizations performed by the Portland Group compiler which the threaded versions cannot take advantage of; global optimizations cannot be performed across thread boundaries. Another reason for the difference is the increased cache misses due to the sharing of data between the EU and SU.

4.2.1 Speedups of the Multithreaded Programs

As we mentioned, the Threaded-C benchmark programs can be run directly on multiple nodes of the MTA-MANNA platform. In figures 5 and 6 we show the relative and absolute 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 the same codes attaining good USE values can also be scaled nicely to sixteen processors (the largest configuration of the MANNA currently available).

4.3 Quantitative Benefits of the SU

To measure the benefits of the basic multithreading support—SU—in the MTA-MANNA platform, we performed a series of experiments with a couple 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 modify them to include more split-phase communications (which includes synchronizations) and try to determine how much more communications the MTA-MANNA platform can absorb before the speedups attained for the original versions of the programs diminish. Figure 7 and 8, show the results of these experiments. It is clearly evident that the SU can handle a sizeable increase in interprocessor communications before affecting the processing rate of the EU. The

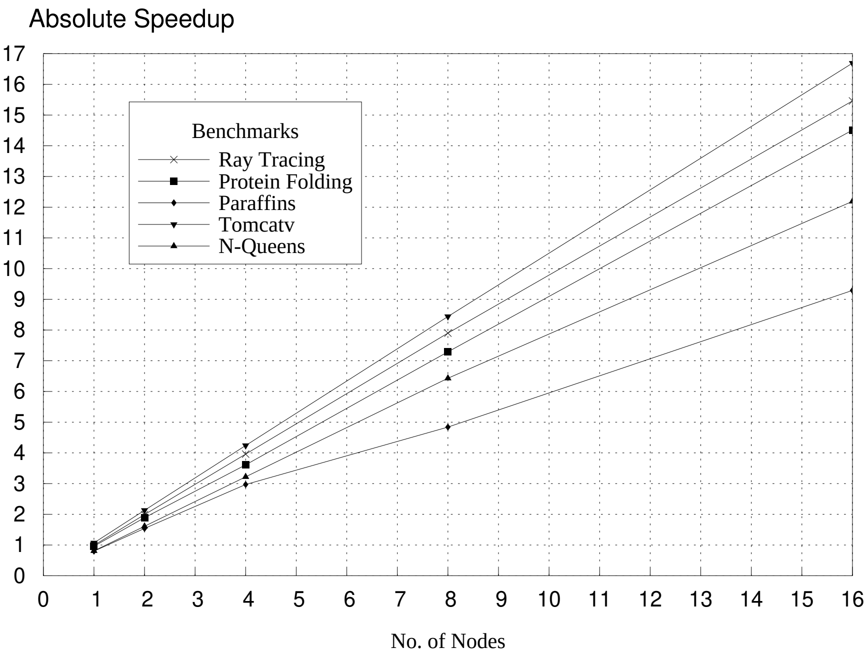

Figure 5: Absolute speedups.

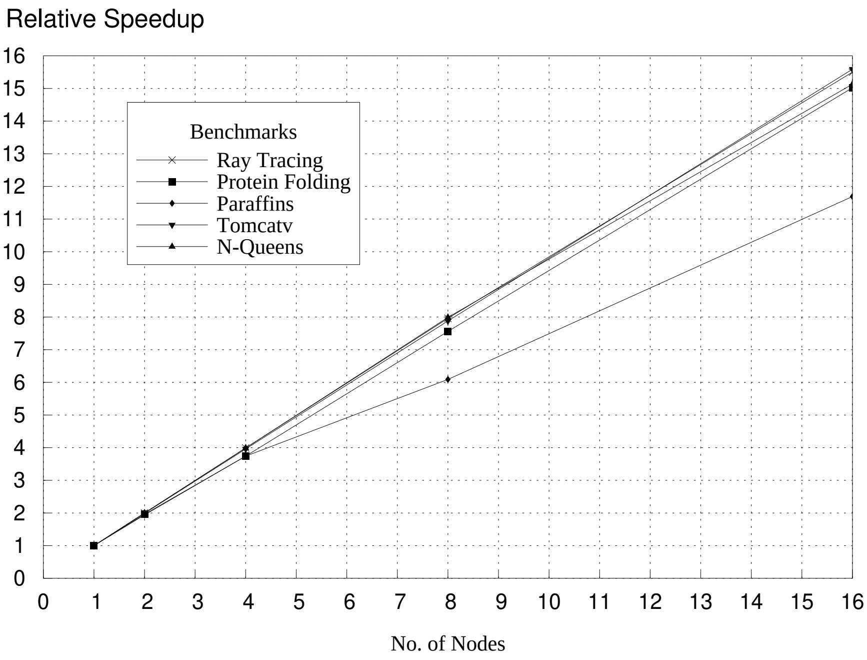

Figure 6: Relative speedups.

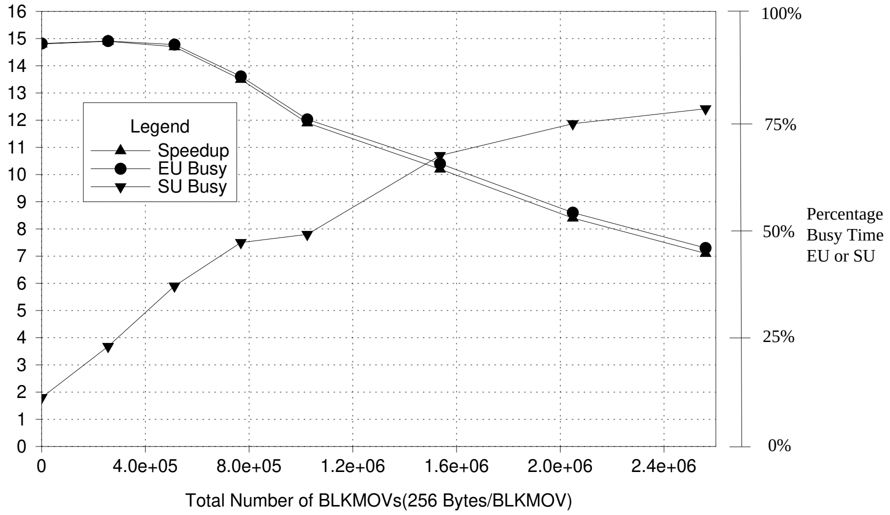

Figure 7: Benefits of the SU for raytracer.

amount of increased communications over the original amount is roughly 10-fold for both raytracing and N-queens before their speedup decreased by 10%. The original version of raytracing required 256 BLKMOVs of 256 bytes of data each, while N-queens required ≈6000 BLKMOVs of 48 bytes of data each. Note that the benefit level of the SU are dependent on the amount of parallelism in the program as well as the original amount communications/synchronizations performed. Our results imply that for parallel applications which have the same amount of parallelism as the benchmark programs, but are more communications intensive, the MTA-MANNA platform can produce the same speedups which we have attained.

5 Status of the MTA Project

Other efforts in our MTA project include the definition of a more sophisticated MTA to support caching of globally shared data and speculative execution, construction of an MTA simulator for gathering more detailed data, construction of the McCAT-MTA C compiler which can automatically generate threads from a regular C program or a program written in a dialect of C which we call *MTA-C*, and the VLSI implementation of the MTA machine. These efforts are described in this section.

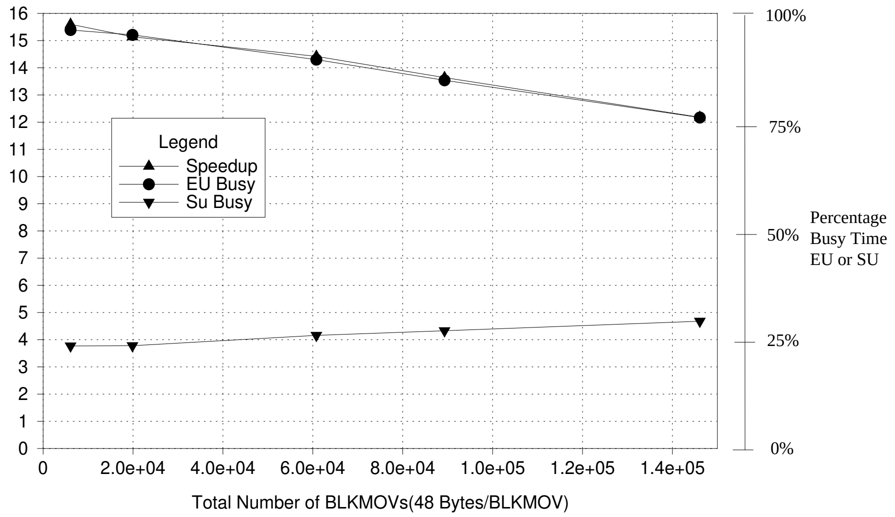

Figure 8: Benefits of the SU for N-queens.

5.1 Defining More Powerful Versions of the MTA

In this paper, we have described an MTA which has basic multithreading operations, block move operations, and an automatic load balancing mechanism. We are currently redefining MTA to include support of caching globally shared data. We intend to incorporate a version of the *global-cache* as defined in [11] which minimally extends a conventional cache to include information regarding which processor owns the data in the cache lines. This *globally-shared-data cache* contains only data which is shared by more than one node, and the on-chip cache of the EU will be used to data private to the node.

Also, we intend to have support for algorithmic speculative execution. In algorithmic speculative execution such as parallel OR-search commonly found in AI-type applications, we need to support non-deterministic arrivals of completion signals from parallel tasks which were spawned 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 the MTA is outlined in [12] 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.

5.2 The MTA Simulator

Although the MTA-MANNA platform runs many of our test applications quite efficiently, it suffers when running *fine-grained* multithreaded programs, because the hardware does not support rapid communication between the EU and the SU. Our experiences running multithreaded code on the MTA-MANNA platform, and the statistics it has gathered, are helping us to refine the MTA architecture and to design the SU hardware. However, many important statistics on memory use and scheduling performance are unavailable, and the platform cannot be programmed to collect them. Furthermore, our current MTA modeling is bound by the MANNA platform; it is difficult to experiment with fine-grained code or to model a system with more processors than available, and both of these will have a major impact on the future design of the MTA. Therefore, we are developing a multi-purpose MTA simulator to collect such data. The primary core of the simulator models the complete functionality of an i860XP processor. A complete architecture is modeled by combining one or more instantiations of the i860 with additional modules for simulating local memory systems and the interconnection network.

The simulator currently can model an n-node MANNA system, with a timing ratio of $5000n$:1 on a Sparc-10, and we will be using it to study the behavior of our MTA-MANNA platform. The simulator is currently in its 'shakedown' phase where it is put through the gamut of benchmark programs. We are presently adding a module to simulate SU hardware. This will allow us to predict the performance of a true MTA, to extract the statistics needed to select design parameters (such as the sizes of the queues), and to experiment with future enhancements to the MTA such as speculative execution [13]. Although MTA instructions such as data_sync will be invoked differently in the true MTA than in the MTA-MANNA platform, the Threaded-C post-processor requires only minor changes to generate true MTA code.

5.3 The McCAT-MTA C Compiler

A major component of our project involves the automatic generation of threads which can take advantage of the features in the MTA. Multithreaded efforts to generate threads of instructions from Sisal and Id for other multithreaded machines have been reported in [18, 5]. In these approaches, the algorithms for thread generation benefit from the declarative nature and implicit parallelism of these source languages. We are targeting a potentially more difficult problem in that we wish to generate threads for a wide range of applications written in the imperative programming language C.

As illustrated in figure 9, our compiler takes either ordinary sequential C programs or explicitly parallel programs written in MTA-C. The purpose of MTA-C is to provide some very simple extensions to C that allows the user to specify parallel statement sequences and parallel loops, and to expose some high-level notions of domain decomposition and block moves to the user. It is important to note that we don't expect the user to provide all of the low-level details about threads, but rather we want the user to concentrate on specifying the important points about the parallelism and domain decomposition of the program. The first version of MTA-C has been implemented and we are continuing to expand this language based on our experiences of hand-coding threaded programs in the lower-level MTA Threaded-C language.

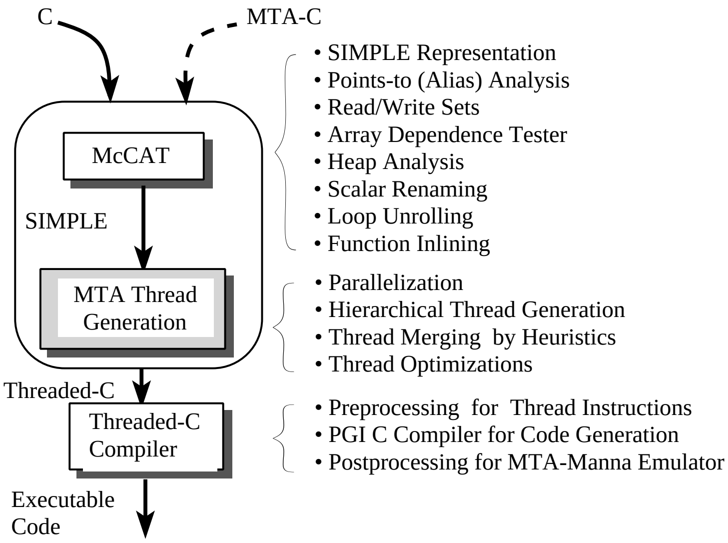

Figure 9: The McCAT/MTA C Compiler

The first component of our compiler comes from the McCAT (McGill Compiler Architecture Testbed) optimizing/parallelizing C compiler[10]. This component takes a C or MTA-C input program, and performs goto-elimination [8] and simplification to produce a structured AST representation of the program called SIMPLE [21]. In order to do a good job in parallelization and thread generation, the McCAT compiler provides sophisticated dependence analyses for stack-allocated pointers (points-to analysis)[7], heap-allocated pointers[9], and a practical array dependence tester[15]. Finally, the McCAT compiler provides several useful high-level transformations including scalar renaming, loop unrolling and function inlining.

The second component of the McCAT-MTA compiler performs the actual MTA thread generation, producing a lower-level Threaded-C program as output. The first phase performs parallelization at both the statement sequence and loop levels. The most interesting part of this parallelization is a process which hierarchically converts each sequential statement sequence of the form $s_1, s_2; ...; s_n$; into a sequence of parallel statements. This process is applied to each statement sequence where each statement, s_i, can be a basic statement or a compositional statement such as a loop or a conditional. Parallelization is performed using the Read/Write Sets that have been computed at the SIMPLE level. Two statements s_i and s_j can be scheduled in parallel if their Read/Write sets do not interfere. It should be noted that the user may have already specified some parallel statements using MTA-C. In this case the parallelization step retains that parallelization and adds any other parallelization that can be detected automatically. Thus, the coarse-grain parallelism can be given by the user, while the finer-grain parallelism will be detected by the compiler.

After parallelization, several heuristics are used to generate threads and parallel function invocations. There is always a trade-off between generating many threads on one processor, and generating parallel function invocations which correspond to threads on different processors. For example, for each parallelizable loop, we can either split and specialize the loop into several smaller threads on one processor, or we can generate a loop of function calls that distributes work among processors.

The first part of our general thread creation strategy is to break threads after remote memory accesses and at function call boundaries. The breaks after remote memory accesses are necessary since these are long-latency operations, while the breaks at function call boundaries are usually used to provide opportunities for parallel function invocation. Note that we do not necessarily introduce thread boundaries for each compositional statement (`for`, `while`, `if`). The second part of the strategy is to group parallel remote accesses into one thread so as to reduce synchronization costs. The third part of the strategy reorders parallel statements so the remote accesses are issued as early as possible to increase the latency hiding abilities.

After threads have been generated, the lower-level threaded-C program is produced, and this program is processed by the Threaded-C Compiler, the last component of figure 9.

5.4 The VLSI Implementation of MTA

As we discussed in section 3, the MANNA platform is a good experimental vehicle for our MTA model, however, it lacks the hardware support for efficient multithreading support.

In our current plans for the MTA machine, we are basing an MTA node on a dual-bus architecture with the EU and SU having their own buses to shared hardware resources so that bus contentions can be minimized (fig. 10). The EU will be an off-the-shelf processor such as the SuperSparc while the SU will be implemented as a customized VLSI chip. Main memory is connected to both processors via the access controller (AC). The main functionality of the access controller (AC) is to regulate access to main memory between the EU and SU and to merge the EU and SU buses on demand from the EU. In this way, the EU can inject messages into the net (via a communication engine in the SU) when the communication chip is free. Furthermore, the AC also contains status and communication registers which enable the EU to monitor SU activity and provide prioritized messages from the EU to SU. The memory subsystem is interleaved and should be capable of providing the necessary bandwidth required by the two units. The event and ready queues are implemented in hardware for fast buffering of messages between the EU and SU—a necessity as outlined in section 3. The ready queue will be implemented as a simple priority queue which accepts ready thread ids with priority, put on top of the queue, and without, put on the bottom. The communication chip and network chips will be off-the-shelf chips developed in research labs.

Within the SU are a RISC core (RC), a communication engine (CE), a register-use cache (RU-cache) [13], and other glue blocks. The RC is tailored to handle common RISC integer operations suitable for the SU as well as basic MTA operations (e.g., spawn, sync). The RC

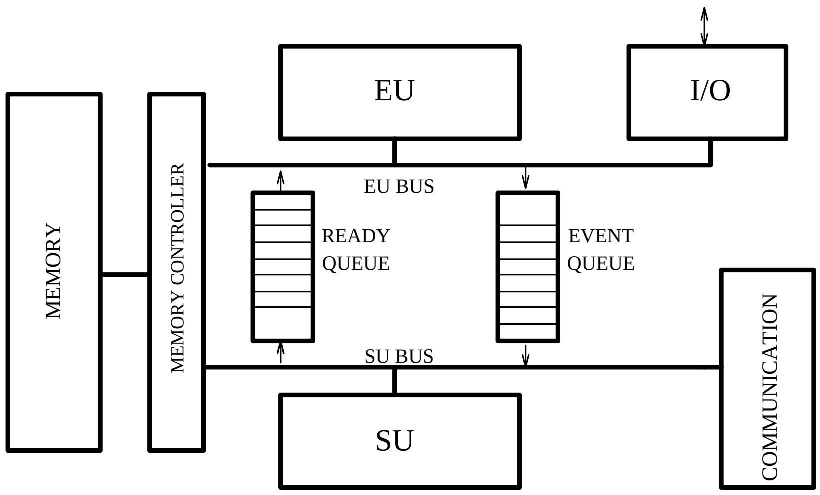

Figure 10: Node architecture of an MTA machine.

can also decide which ready thread ids to prioritize by consulting the RU-cache which keeps track of which function instance have their contexts in the multiple register sets of the EU. Complex/future MTA instructions can be specified as a sequence of instructions and traps will be provided for fast vectoring to the code. Messages are injected or extracted from the network through a communication chip by the CE. The CE is a low level communication handler which monitors the status of the communication chip and performs the necessary communications, and packing and unpacking of messages as instructed by the RC.

We are currently specifying the functionalities of the SU, the queues and access controller in Verilog on Synopsys and Cadence. In the future, hardware support for caching globally shared data will also be incorporated.

6 Related Work

The RWC-1 project [19] 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 superscalar RISC core with embedded mechanisms for fork-join thread synchronizations. RICA does not have a separate unit for processing both synchronizations and remote split-phase requests.

Tera Corporation's MTA machine [2, 3] interleaves the execution of multiple threads on one processor node which our MTA does not do.[3] Moreover, the Tera MTA does not have a separate dedicated unit for handling synchronizations and remote memory requests.

[3]Private discussions with Burton Smith has revealed that we both first used the acronym 'MTA' at about the same time circa 1992.

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 for a function instance. A node in Alewife contains mechanisms for maintaining cache coherency, however, there is no separate unit dedicated for synchronizations and remote memory requests.

The P-RISC [17] of Nikhil and Arvind also explored the possibilities of constructing a multithreaded architecture around a RISC processor. The major difference is that there were no distinct synchronization unit proposed in that design.

The division of the processor node into separate synchronization and execution units distinguishes our MTA model from models such as TAM [5], in which all operations are performed on one processor. 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.

Spertus et al [20] analyzed the efficiency of fine-grain support mechanisms on both the J-Machine and the TAM model on Thinking Machine's CM-5. In the J-machine, like the TAM model, 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 the MTA and *T is that the synchronization unit is user-programmable in the *T—it is another microprocessor as in our MTA-MANNA platform—but not in the MTA. The user has some control over synchronization in the MTA, since the sync slots are accessable by the EU, but we opted for a non-user-programmable SU, thus removing the possibility of deadlock introduced by user code. A follow-on architecture called *T-NG [4] 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 the MTA and *T-NG is that we will build the synchronization unit as a custom ASIC while *T-NG uses an off-the-shelf microprocessor as is done in our MTA-MANNA platform.

In the MTA design, 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 MTA—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.

We have implemented a number of benchmark programs on the MTA-MANNA platform and our preliminary experimental results have demonstrated that efficient multithreading can be realized without major intrusions to the execution efficieny of sequential code. Furthermore, we have provided data from our first attempt at quantifying the latency tolerating capabilities in the MTA-MANNA.

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.

References

[1] Anant Agarwal, Beng-Hong Lim, David Kranz, and John Kubiatowicz. APRIL: A processor architecture for multiprocessing. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 104–114, Seattle, Washington, May 28–31, 1990. IEEE Computer Society and ACM SIGARCH. *Computer Architecture News,* 18(2), June 1990.

[2] Gail Alverson, Bob Alverson, David Callahan, Brian Koblenz, Allan Porterfield, and Burton Smith. Integrated support for heterogeneous parallelism. In 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*, chapter 11, pages 253–283. 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.

[3] 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.

[4] Boon Seong Ang, Arvind, and Derek Chiou. StarT the next generation: Integrating global caches and dataflow architecture. CSG Memo 354, Computation Structures Group, MIT Laboratory for Computer Science, Cambridge, Massachusetts, February 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 *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.

[6] Jack B. Dennis and Guang R. Gao. An efficient pipelined dataflow processor architecture. In *Proceedings of Supercomputing '88*, pages 368–373, Orlando, Florida, November 14–18, 1988. IEEE Computer Society and ACM SIGARCH.

[7] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 242–256, Orlando, Florida, June 20–24, 1994. ACM SIGPLAN. *SIGPLAN Notices,* 29(6), June 1994.

[8] Ana M. Erosa and Laurie J. Hendren. Taming control flow: A structured approach to eliminating goto statements. In *Proceedings of the 1994 International Conference on Computer Languages*, pages 229–240, Toulouse, France, May 16–19, 1994.

[9] Rakesh Ghiya. Practical techniques for interprocedural heap analysis. Master's thesis, McGill University, Montréal, Québec, Expected February 1995.

[10] 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 Uptal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, number 757 in Lecture Notes in Computer Science, pages 406–420, New Haven, Connecticut, August 3–5, 1992. Springer-Verlag. Published in 1993.

[11] Herbert H. J. Hum and Guang R. Gao. Supporting a dynamic SPMD model in a multi-threaded architecture. In *Digest of Papers, 38th IEEE Computer Society International Conference, COMPCON Spring '93*, pages 165–174, San Francisco, California, February 22–26, 1993. IEEE Computer Society Press.

[12] Herbert H. J. Hum, Kevin B. Theobald, and Guang R. Gao. Building multithreaded architectures with off-the-shelf microprocessors. ACAPS Technical Memo 68, School of Computer Science, McGill University, Montréal, Québec, October 1993.

[13] 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.

[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 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.

[15] Justiani and Laurie J. Hendren. Supporting array dependence testing for an optimizing/parallelizing C compiler. In Peter A. Fritzson, editor, *Proceedings of the 5th International Conference on Compiler Construction, CC '94*, number 786 in Lecture Notes in Computer Science, pages 309–323, Edinburgh, Scotland, April 7–9, 1994. Springer-Verlag.

[16] R. S. Nikhil, G. M. Papadopoulos, and Arvind. *T: A multithreaded massively parallel architecture. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 156–167, Gold Coast, Australia, May 19–21, 1992. ACM SIGARCH and IEEE Computer Society. *Computer Architecture News,* 20(2), May 1992.

[17] 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.

[18] L. Roh, W. A. Najjar, and A. P. W. Böhm. Generation and quantitative evaluation of dataflow clusters. In *Proceedings of Functional Programming Languages and Computer Architectures*, Copenhagen, Denmark, June 9–11, 1993. ACM SIGPLAN and SIGARCH.

[19] Shuichi Sakai, Kazuaki Okamoto, Hiroshi Matsuoka, Hideo Hirono, Yuetsu Kodama, and Mitsuhisa Sato. Super-threading: Architectural and software mechanisms for optimizing parallel computation. In *Conference Proceedings, 1993 International Conference on Supercomputing*, pages 251–260, Tokyo, Japan, July 20–22, 1993. ACM.

[20] 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 *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 302–313, San Diego, California, May 17–19, 1993. ACM SIGARCH and IEEE Computer Society. *Computer Architecture News,* 21(2), May 1993.

[21] Bhama Sridharan. An analysis framework for the McCAT compiler. Master's thesis, McGill University, Montréal, Québec, September 1992.