McGill University
School of Computer Science
ACAPS Laboratory

Advanced Compilers, Architectures and Parallel Systems

Building Multithreaded Architectures with Off-the-Shelf Microprocessors

Herbert H.J. Hum†
Kevin B. Theobald
Guang R. Gao

ACAPS Technical Memo 68
October 1, 1993

A condensed version of this report appears in the Proceedings of the 8th IEEE International Parallel Processing Symposium (IPPS '94), Cancún, Mexico, April 23–26, 1994.

e-mail: herbert@ece.concordia.ca, theobald@cs.mcgill.ca, gao@cs.mcgill.ca

†Dept. of Electrical and Computer Engineering, Concordia University, Montréal, Québec

Abstract

Current strategies for supporting high-performance parallel computing often face the problem of large software overheads for process switching and interprocessor communication. This document presents the design of the Multi-Threaded Architecture (MTA) model, a multiprocessor architecture designed for the efficient parallel execution of both numerical and non-numerical programs. The basic MTA design begins with a conventional processor, and adds what we believe to be minimal external hardware necessary for efficient support of multithreaded programs.

The presentation begins with the basic processor design and the program execution model. The latter includes a description of *activation frames* and *thread synchronization*. This is followed by a detailed description of the instruction set. Major features of the MTA include the Register-Use Cache for exploiting temporal locality in multiple register set microprocessors more effectively, support for programs requiring non-determinism and speculation, and local function invocations which can utilize registers for parameter passing.

Contents

List of Figures

List of Tables

1 Introduction

Current strategies for supporting high-performance parallel computing often face the problem of large software overheads for process switching and interprocessor communication. These large overheads limit the range of applications which can be effectively mapped to these architectures. For example, if there are high overheads on process switching, then parallelism must be coarse-grain so that the overhead for process-switching does not greatly impact the total execution time. While some architectures have been proposed to deal with the problems of overheads, most of them require unusual hardware designs that are far removed from conventional processor technology. Since such experimental ventures generally don't have the same resources as the designers of mainstream processors, it is nearly impossible to build prototypes that can compete with the mainstream architectures and thus gain acceptance. In our work, we are attempting to address these issues by proposing a design for a parallel multithreaded architecture which can execute a wide range of applications efficiently, yet can be implemented with a conventional high-end microprocessor augmented by simple external hardware, rather than a modified microprocessor or completely new processor design.

In this paper, we describe the Multi-Threaded Architecture (MTA) project. There are two main goals to this project:

1. To describe architectural features supporting *scalable parallel computing*, that is, features to support the effective use of the physical resources of a parallel computer such that the performance gains are proportional to the number of processors (if the application has enough parallelism to exploit). Such features include mechanisms for:
   (a) tolerating variable latencies in interprocessor communications, i.e., mechanisms for simultaneous and efficient exploitation of both fine-grain and coarse-grain parallelism;
   (b) fast and efficient interprocessor synchronization and communications; and
   (c) effective exploitation of spatial and temporal locality.

   Ideally, these features will be added to the processor node design as special I/O devices or memory subsystems. In doing so, we can build multithreaded architectures with off-the-shelf microprocessors.

2. To provide a simple and sound program execution model as an interface between compiler writers and the system implementation. To this end, a set of extensions to a typical RISC instruction set will be defined, as well as a model describing how an implementation of the MTA will function when presented with code compiled according to the program execution model.

As indicated by the name of the model, our approach to scalable parallel processing is based on a multithreaded execution model. A multithreaded architecture supports multiple "threads of control" within one node. Conceptually, a thread of control is similar to the notion of a *process* in multiprogramming computer systems, except that threads communicate with each other much more frequently than processes, because they are generally cooperating on one problem, and, in most machines, the threads are much shorter than a typical process. Multithreaded architectures are reviewed in Appendix A.

Rapid switching between threads is needed in order to minimize the impact of variable latencies in interprocessor communications on the overall execution time. For example, when a thread is performing a remote send or receive, the processor can quickly switch to another thread which is available for execution, thereby hiding the communication latency. Moreover, communication mechanisms can be smoothly integrated into a multithreaded processor to yield low sending/receiving overheads. Therefore, thread switching generally uses hardware assist in order to lower the associated costs of switching far below the costs of software overheads in conventional parallel processors.

In our previous work, some initial thoughts on the MTA model were presented [14]. The intent of this report is to formally define the basic MTA model and present various features which we believe are beneficial to the efficient and effective support of both scientific numerical computations and non-numerical applications. In particular, the MTA model is to specify features which ensure high performance even for programs where the computation and communication patterns are irregular.

The main characteristics of the MTA model are:

- Division of the processor into a separate Execution Unit, programmed like a RISC processor, and a Synchronization Unit which can be hardwired to support dataflow-like synchronization of threads.

- Use of thread prioritizing and a Register Use Cache to improve exploitation of data locality. The Register Use Cache is used to determine which function activations have their contexts in a multiple register set file such that the loading and unloading of contexts can be diminished.

- Use of a tree of activation frames to represent the program state.

A primary difference between the MTA and most other multithreaded architectures is that the MTA processor node can be built around an unmodified RISC processor. Special multithreading support can be provided by external hardware. The tasks performed by the external hardware are many, but the individual tasks are fairly simple. Therefore, we believe the hardware support can be implemented with simple ASIC chips.

1.1 Synopsis

The next section covers the basic MTA architecture and execution model. It descibes the organization of the processor node, the use of activation frames as the basis of program execution, and the MTA memory hierarchy. Section 3 gives more details about the architecture of the MTA node, including the network message format and the instruction set of the Execution Unit. In Section 4, we present some advanced features of the MTA for supporting improved scheduling and speculative execution.

The MTA is in the early stages of development, and much work needs to be completed. Section 5 briefly discusses our plans for further work on the MTA. In Section 6, we compare the MTA to other multithreaded machines. The last section summarizes our paper. In addition, we present a short overview of multithreaded architectures in an appendix.

2 The Base Multi-Threaded Architecture Model

In this section, we describe the structure and operations of the MTA model.

2.1 Processor Node Organization

Each MTA node contains an *Execution Unit* (EU) and a *Synchronization Unit* (SU) linked together by buffers (see Figure 1). In simple terms, the EU executes the instructions that comprise the program, while the SU tells the EU which instructions to execute. This organization is derived from the *Argument-Fetch Dataflow Processor* [6], in which the analogous units are called the *Pipelined Instruction Processing Unit* (PIPU) and the *Dataflow Instruction Scheduling Unit* (DISU), respectively.

The SU and EU share the node's local memory, which is cached for better performance. Accessing data in a remote node requires explicit 'request' and 'send' messages; thus hardware-wise, the MTA multiprocessor is a distributed memory machine. However, this does not preclude the use of software and/or limited hardware machanisms for projecting a global address space to the compiler; which would make the MTA a *distributed shared memory* machine.[1]

The EU consists of a register file and a RISC execution pipeline. The register file contains registers for storing temporary values, and special registers such as a frame pointer register (%fp) and a status register (%sr). The execution pipeline processes instructions in an active thread, using standard RISC register-to-register and load/store operations. Moreover, it can inject messages into the interconnection network and issue synchronization signals to the local SU.

[1] A version of the MTA in [14] specifies one such mechanism.

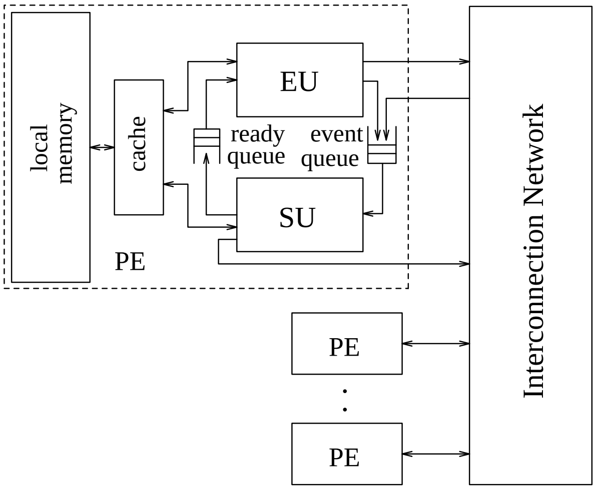

Figure 1: The Multi-Threaded Architecture

The buffer between the SU output and EU input is called the *ready queue*, and holds the threads that are waiting for execution. A thread becomes *active* (initiated for execution) when the EU fetches its *ready thread identifier* (or *ready thread id* for short) from the ready queue. The execution pipeline of the EU processes an active thread until the thread relinquishes control of the pipe with a special instruction which fetches the next available ready thread id from the ready queue and context-switches to the new thread.

It is the responsibility of the SU to generate ready thread ids of threads to be executed and deposit them into the ready queue. The SU is also responsible for processing messages in the *event queue*. Events in the event queue include local *synchronization* signals emitted by the EU of the local processor, as well as remote synchronization signals and communication messages delivered from the interconnection network. Lastly, the SU is solely responsible for handling remote requests for data.

We believe that a good high-end RISC processor can be used for the EU. The SU will most likely require special hardware. The tasks required by the SU, however, are generally simple, and can be implemented using ASIC designs.

2.2 Thread Synchronization

The method of determining the order of execution of threads on the MTA is derived from the dataflow model [5]. In this model, an *actor* (instruction) is *enabled* (becomes eligible for execution) as soon as all inputs required by the actor are available. The processor has to know when all inputs needed by the actor have arrived. Therefore, there is an "enable" count associated with each actor, which counts how many inputs remain to be filled. As the input values are produced, the count is decremented. When it reaches 0, the actor is enabled and can be executed as soon as processing resources become free.

The MTA model is similar to the dataflow model, except that each actor is a thread that can contain more than one operation. A thread can either be directly activated by another thread, or it can keep a count of how many inputs it has received. For each thread of the latter type, there is a *synchronization slot* containing three fields: a synchronization count, 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. The thread pointer indicates which thread is to be activated when the sync count reaches 0, i.e., enough synchronization events have occurred. The complete synchronization method, described in a later section, is very simple, yet can support the dataflow model of programming, and can also easily support additional features such as non-determinism and mutual exclusion. These are not possible under the classic dataflow model, but can be useful for exploiting certain types of parallelism such as OR-search.

2.3 Activation Frames

Conventional sequential processors keep the local context of each function instance in a frame, and these frames are kept on a stack. This is an efficient way to use memory when there is a single thread of control, but it eliminates most opportunities for executing functions in parallel [16]. Therefore, our runtime model will support a *tree of activation frames*, in which each activation frame is dynamically linked to its caller. This was first proposed for dataflow machines [18] and later used in multithreaded systems [4, 16],

When a function is invoked in the MTA, an *activation frame* is allocated to represent that function instance. A frame pointer *fp* points to the base of the frame and is used to identify the frame uniquely. A function will usually contain many threads. A given thread in a function activation is considered distinct from the corresponding thread in another activation of the same function, even though the two threads share the same code. Therefore, under the MTA runtime model, a thread can be *uniquely* identified by a *thread pointer* <*fp, ip*> consisting of a frame pointer (*fp*) and an instruction pointer (*ip*). The former points to the activation frame and the latter is the address of the first instruction of the thread to be executed.

Each activation frame is structured as in Figure 2. An activation frame has three main components. The first links a frame to its caller, i.e., its parent 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 is a synchronization slot (see Section 2.2) for each thread that requires synchronization. Note that sync slots are dynamic and specific; if there are two activation frames corresponding to two invocations of the same function, the sync slots for corresponding copies of the same thread are distinct.

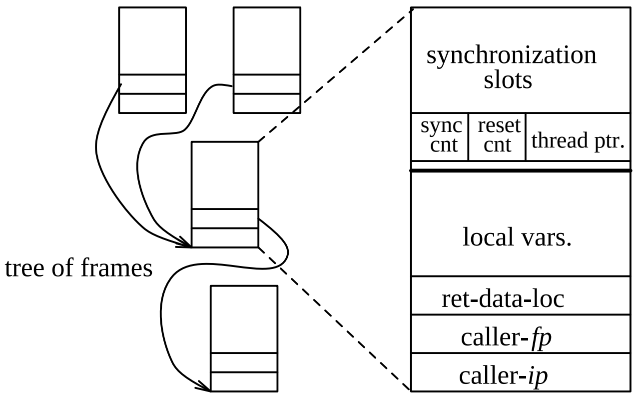

Figure 2: Layout of an activation frame

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 (this is only useful when the function was invoked from a remote processor, as we will see in a later example). When the callee terminates, i.e., it does a return instruction, the callee's activation frame is deallocated.

At runtime, a thread can be activated in three ways:

1. If it is the first thread of a function, it will be activated when the function is invoked.

2. An explicit *spawn* instruction is issued by a thread which activates another thread.

3. When the sync count in a synchronization slot reaches zero, the thread indicated by the thread pointer in the slot is activated.

It is the responsibility of the compiler/programmer to use the latter two thread activation methods properly.

For data structures which have lifetimes independent of the lifetimes of activation frames, a global heap can be used for their storage. Therefore, an activation frame may contain pointers (in the local variable slots) to data structures in the global heap.

An MTA frame must reside entirely on one processor. Since a processor can only execute one thread at a time, it follows that there can be no parallel execution within a function call. This is not a problem for programs with highly-recursive functions, because they derive their parallelism by scattering function calls across many processors. On the other hand, it would appear to be a disaster for large, regular scientific computations such as data-parallel applications. For example, a matrix computation would be written in the program as a single function call, yet one would want to distribute the matrix elements in a data-parallel fashion.

However, the function boundaries in the MTA code do not have to correspond to function boundaries in the source code. For instance, one of the easiest data-parallel applications to program on a parallel machine is a mesh simulation [8], in which a physical region is represented as a grid of points, and at each time step, each grid point is updated by performing a function on its previous state and the state of its nearest neighbors. Each time step could be done in the source code with a single function call or series of function calls. Yet the compiler (possibly with the help of programmer annotations) could implement it as a collection of function activations that send their outputs to each other, rather than to the calling frame. Furthermore, since setting up this pattern of function activations each time would be time-consuming, the program could create function activations that never return, but remain static throughout the execution of the program. This mode of execution can be viewed as similar to the SPMD model of execution of data parallel programs on conventional distributed memory machines. It has the added advantage that the multithreading in each processor can tolerate latency by overlapping the thread execution with the communication. Thus, the MTA model provides a potential for efficient execution of scientific code as well as non-numeric code.

2.4 The MTA Memory Hierarchy

The set of memory locations visible to an active thread in the EU may be viewed as a memory hierarchy:

1. Internal data registers
2. Memory local to the thread's activation frame
3. Memory local to the processing node
4. Global memory (i.e., belonging to other processors)

The first two together represent all values local to an activation frame, while the last two represent all values outside the frame.

2.4.1 Data Within a Frame

The first level of the hierarchy, the internal EU registers, are used for storing intermediate results produced and consumed within the same thread. Indeed, this is one major advantage of multithreading over pure dataflow; processors can exploit such locality for faster execution.

The second level of the hierarchy is used for inter-thread communication. In general, when values are to be passed from one thread to the next, they can't be passed via registers. This is because the EU cannot guarantee that the next thread executed will not be from a different activation. Therefore, the thread must "spill" its live registers to local activation slots before terminating.

These spills, and the load operations at the beginning of threads, use the regular load/store RISC instructions, generally using offsets from the %fp register. These operations could take a long time due to cache misses. However, if a thread is active, its activation frame is likely to have been recently used, and therefore resident in the cache. Indeed, it should be possible to design the SU to ensure that the relevant frame slots are in the cache before the thread is activated. Therefore, to avoid adding complexity to the EU, we will simply have it stall in the unlikely event of a cache miss. (Out-of-order execution may ameliorate this somewhat, but this is not an essential part of the MTA model.)

The need for register spills and reloads can be reduced by taking advantage of temporal locality in the scheduling of threads. This means that the SU will favor (prioritize) threads in the ready queue which are immediately related to the currently executing thread. In TAM [4] terminology, this would refer to threads in a *quantum*—an execution of a function instance may include multiple quanta.

Quantum boundaries, in other words, threads belonging to a quantum, are difficult to identify unless the compiler partitioning strategy produces simple threads where the time between a thread and the spawning of its successor threads can be determined statically at compile time. In those cases, it should be straightforward to eliminate such extraneous spawning and create larger threads instead. However, allocating registers across the larger threads appears difficult since the thread boundaries are now demarcated with variable latency instructions, e.g., split-phase operations. In this light, we propose that registers be allocated for an entire function activation instead of for a quantum. That is, values which are used by many threads in a function—the decision of how many threads is 'many' should be left to the compiler writers when the costs of loading/storing of registers are determined—should be kept in registers across thread boundaries, whereas values which are solely produced by one thread and consumed by another should probably not be in registers upon thread switching.[2] Let us classify the former as *global function values* and the latter, *temporary values*. What is hoped is that global function values need not be loaded, stored, and re-loaded upon thread switching.

Identifying threads belonging to the same function activation as the currently executing thread in the EU, i.e., prioritizing ready thread ids, can be easily performed by comparing the *fp* value of the ready thread id with the %fp register. The SU can put such recently created ready thread ids on top of the ready queue. Since threads will usually spawn or synchronize on threads within their own activation, this scheduling policy will increase the chances that the next thread is in the same frame and hence shares some registers.

[2]This decision can also be affected by how many registers are in the register set.

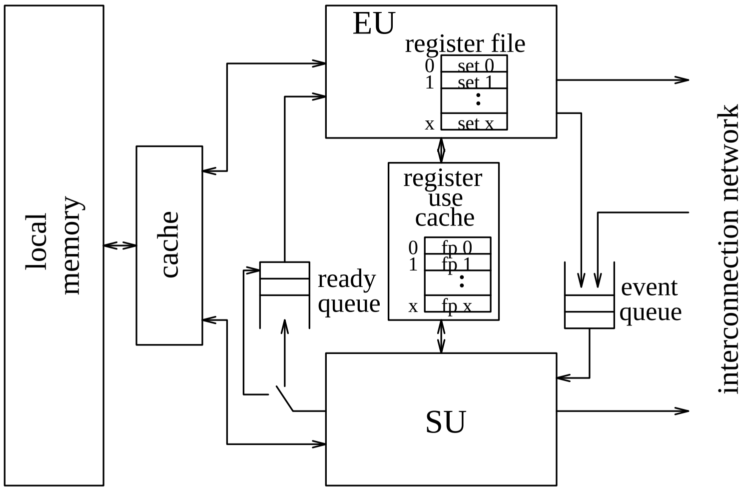

Figure 3: The Register Use Cache in an MTA node

If the EU determines that the next ready thread is from the same frame, it can bypass the instructions at the start of the thread for loading global function values. Similarly, if the EU knows that the incoming thread is from the same frame, it can skip the global spill instructions at the end of the outgoing thread. The checking of incoming ready threads can be readily performed in a software routine if, by convention, one of the general-purpose registers is reserved for indicating the identity of the recently terminated thread.

A more ambitious solution to the problem of register spilling can be found in microprocessors with large register files such as the SPARC and AMD29050. Those register files can be partitioned into multiple register sets such that each register set can service one function activation. The question is: how can these multiple register sets be used effectively in the MTA model?

To this end, we propose a *Register Use Cache* (RU cache) which keeps track of which register set is assigned to which function instance. For example, a register file with n register sets will have a register use cache of n entries. Each entry is numbered and corresponds to a register set, e.g., entry i would correspond to register set i and would contain the frame pointer of the function instance to which register set i is assigned (see Figure 3). The idea is to have the RU-cache be visible to the SU such that when the SU is about to deposit a ready thread id into the ready queue, the RU-cache is associatively searched for a frame pointer which matches the *fp* value of the ready thread id. A match in the RU-cache would indicate that that ready thread id should be prioritized. Moreover, the register set which it is assigned to should be included in the ready thread id such that when the ready thread id does enter the EU, the proper register set will be used. In effect, this is a simpler form of the *registering process* as outlined in the register-cache of the Super-Actor Machine [12][3]. Complexity arises when a ready thread id

[3]This simpler form of the registering process is also referred to as "dynamic binding" in [23].

which is not prioritized enters the EU. in that case, a software routine is executed which tries to find an unallocated register set with the help of the RU-cache. If that is not possible, then the RU-cache will use some policy to determine which allocated register set should be spilled to make way for the new ready thread.

2.4.2 Data Outside a Frame

Accesses to remote memory locations require explicit messages to be sent to the remote SU. These are clearly long-latency operations that would cause huge stalls if they were in the middle of threads. Therefore, we will use *split-phase transactions* [4] to avoid this situation. A remote load operation will be performed in one thread, and the values produced will be used in *other* threads. The other thread(s) will be activated by the messages returning from the remote SU. This allows the EU to run other waiting threads during the interim.

Accesses to memory locations in the local memory, but external to the activation frame, are a borderline case. Such accesses are more likely to cause cache misses than references within the frame, especially if the external references are to the heap. These, therefore, should also be constructed as split-phase transactions. In other words, to fetch a heap location, the read request should be issued by one thread to the local SU (via the event queue). The SU will read the location, copy it into a local frame slot, and activate the appropriate thread. That thread will then load the frame slot into a register.

This would appear to be wasteful in cases where the data value is already in cache, because it is simply being copied from one memory location to another before finally being copied into a register. However, the reduction in cache-miss stalls in the EU will outweigh the extra overhead required in the cache-hit cases. (The extra overhead in cache-hit cases is low anyway, since it usually involves copying from one cache line to another, and a suitably-designed cache should be able to do this efficiently).

It may also be possible for the compiler to determine whether or not a split-phase transaction is really needed. For instance, if a thread is to read contiguous parts of a record structure, it can access the first component using a split-phase transaction, and the thread at the other end of the transaction can load the other components directly into registers, on the assumption that they will be in the same cache line. (This requires that the compiler know about the cache line size, though.)

3 Implementation of the MTA Model

The following section gives more details about the MTA architecture. It lists the types of messages that can be processed by the SU and the types of instructions that can be executed by the EU. Since many of the non-standard EU instructions are for sending messages to the SU, there is some duality between the SU messages and EU instructions.

LOCAL SYNCHRONIZATION
<‘**spawn**’, *fp*, *ip*>
<‘**sync**’, *fp*, *ss_off*>
REMOTE SYNCHRONIZATION
<‘**rspawn**’, *fp*, *ip*>
<‘**rsync**’, *fp*, *ss_off*>
DATA TRANSFER
<‘**data&spawn**’, *addr_val*, *value*, *fp*, *ip*>
<‘**data&sync**’, *addr_val*, *value*, *fp*, *ss_off*>
DATA REQUEST
<‘**get-spawn**’, *addr_val*, *addr_ret*, *fp*, *ip*>
<‘**get-sync**’, *addr_val*, *addr_ret*, *fp*, *ss_off*>
REMOTE FUNCTION INVOCATION
<‘**invoke**’, *f_name*, *f_size*, *caller_fp*, *caller_ip*, *ret_data_loc*, *no_params*, *params*>

Table 1: Communication and synchronization messages processed by the SU

3.1 The Synchronization Unit

The SU is responsible for issuing *ready thread ids* to the EU. The ready thread id <*fp*,*ip*> is actually put into the *ready thread queue* by the SU and the EU will fetch the ready thread ids from the queue.[4] The SU determines which threads are ready to be executed by processing incoming *synchronization signals* issued by the EU and *communication messages* from the interconnection network. These signals and messages are listed in Table 1 and explained in the following subsections.

3.1.1 Local Synchronization Messages

The local synchronization signals processed by the SU (first section of Table 1) are **spawn** and **sync**. They can only be issued by the local EU. When the SU encounters a **spawn** signal, it immediately puts the ready thread id <*fp*, *ip*> into the ready queue. As for the **sync** signal, (*fp* + *ss_off*) points to the synchronization slot to be operated upon. The sync count in the slot is decremented; if it reaches zero, the reset count is copied into the sync count, thus preparing the thread to receive new **sync** signals, and the thread pointer in the sync slot is put into the ready queue.

If the thread corresponding to the sync slot affected by the **sync** message requires an input value produced by the thread that generates the **sync**, then the thread should store the

[4] Note that the form of a ready thread id is the same as that of a thread pointer.

data in the proper place *before* sending the **sync** signal to the SU. Note that this doesn't require a special message, since this operation merely requires a write to local memory.

3.1.2 Remote Synchronization Messages

The SU also must process communication and synchronization messages received from the interconnection network. There are four types of messages: remote synchronization, data transfer, data request, and remote function invocation (see Table 1).

The messages **rspawn** and **rsync** are used to activate a thread residing on a remote processor, and to synchronize with a thread residing on another processor, respectively. They are analogous to the **spawn** and **sync** messages.

Data transfer messages combine data transfer and synchronization, in order to guarantee that the incoming data is written before the thread which consumes it can be activated. When the SU encounters a **data&spawn** (**data&sync**), it writes the data as specified by *value* into local memory at address *addr_val*. The SU then performs the appropriate spawn (sync) operation using the last two arguments in the message. (Note that there is no equivalent local signal, since the program order can guarantee that a data write will occur before a spawn or sync operation.)

The data request messages **get-spawn** and **get-sync** are used to retrieve data when programming using a shared-memory paradigm. Each is sent from the requesting processor to the processor owning the requested data. The SU of the owner processor fetches the data from memory address *addr_val*, then constructs a **data&spawn** or **data&sync** message with the appropriate field values (the *addr_ret* will be used as the *addr_val* of the return message), and injects the just-created data transfer message into the interconnection network. This form of retrieving data is generally referred to as a *split-phase transaction* where the requesting processor for data does not wait for the response but continues with useful processing (of other active threads). One may remark that the **get-sync** and **data&sync** message pair is a more powerful version of a simple split-phase transaction; this message pair can be used to fetch individual data efficiently from various processors before a thread can execute, i.e., a *split-phase gather operation.*

The **invoke** message is used to perform a function invocation. To process the **invoke** message, the SU calls a frame allocator (see Section 3.3) to get a free frame of size *f_size* from local memory, pointed to by a new frame pointer fp', and copies the caller frame pointer (*caller_fp*) and caller instruction pointer (*caller_ip*) into the first three slots of the new frame. Then the parameters (*params*) — *no_params* specifies the number of parameters — are copied into assigned slots of the new frame, perhaps the next successive slots following the function linkage information. Lastly, the ready thread id <fp', *f_name*> is put into the ready thread queue; the function name *f_name* is simply the address of the first instruction of the function's first thread.

3.1.3 Introducing Non-Determinism and Mutual Exclusion

To facilitate the support of programs with non-determinism (such as OR-parallelism in search programs), or critical regions protected by mutual exclusion mechanisms, we can extend the semantics of the synchronization operation as performed by the SU. The modification is as follows. If a **data&sync** is received by the SU, and the sync count of the specified synchronization slot is already 0, then the data is *not* written into the specified address. This implies that if some other thread had already deposited its value and sync'ed the thread, then subsequent threads which were "racing" with the previous thread are simply ignored; thus allowing for non-deterministic behavior. Ordinarily, when a sync count reaches 0, the SU will set it back to the reset count, and it is up to the compiler, following proper construction rules, to ensure that no more data will come in until the thread has finished reading its inputs [13]. However, if the reset count *itself* is 0, then when the sync count reaches 0, it will *remain* at 0; thus disabling subsequent attempts to write data into the synchronized address or to spawn the thread.

In order for the thread to be triggered again (e.g., for another parallel-OR search), it is necessary to set the sync count back to a non-zero number so that new incoming messages are not ignored. Since synchronization slots are in the activation frame, the sync count can be written directly by an instruction in the EU. Thus, the input to such a thread can be "unblocked" either by the thread itself or by a thread spawned by the "blocked" thread.

What is done with the data that is blocked when a sync count of 0 is encountered? That would depend on what kind of nondeterminism is desired. For parallel-OR, in which multiple paths are searched simultaneously and the first solution that is found from any path is accepted to the exclusion of all others, the processor can simply throw away the superfluous messages. If mutual exclusion is desired, then all messages should eventually lead to thread activations; thus the SU should put them into a special queue. Alternatively, the remote processors could vie for access by sending sync messages only, and send their data once they have received permission.

3.2 The Execution Unit

The instruction set of the MTA is based on a typical RISC instruction set, epitomized by the DLX instruction set [11]. The typical set of RISC instructions is augmented by special instructions for such operations as thread switching. Since our goal is to use an off-the-shelf microprocessor for the EU, we can't expect these instructions to be available in the EU hardware. Rather, these instructions will correspond to macros which will be translated to native processor code during the final assembly phase, or to software traps.[5] The interface between the EU and the SU will be designed so that the special multithreading instructions can be implemented with as few native instructions as possible.

[5] For instructions which access the ready and event queues and the RU-cache, regular load/store instructions to specified memory locations reserved for the specialized I/O devices can be used.

ARITHMETIC, LOGIC AND BRANCHING
Typical RISC instructions, e.g., DLX, SPARC, MIPS, i860XP, etc.
THREAD SWITCHING
fetch_next
SYNCHRONIZATION
spawn *ip* **sync** *ss_off* **xspawn** *fp*, *ip* **xsync** *fp*, *ss_off*
DATA TRANSFER
data&spawn *addr_val*, *value*, *fp*, *ip* **data&sync** *addr_val*, *value*, *fp*, *ss_off*
DATA REQUEST
get-spawn *addr_val*, *addr_ret*, *ip* **get-sync** *addr_val*, *addr_ret*, *ss_off*
REMOTE FUNCTION INVOCATION
invoke *dest_PE*, *f_name*, *f_size*, *caller_fp*, *caller_ip*, *ret-data-loc*, *no_params*, *params*

Table 2: Instruction Set of the MTA

We don't want to limit our design options by over-specifying the basic instruction set of our machine. Rather, our approach is to specify *only* the additional instructions needed for multithreading. Since instruction sets show differences in the mixtures of instructions offered, even among RISC chips, we would like to allow a multithreading compiler to make optimizations specific to a particular processor, rather than choosing one specific instruction set, compiling to that target, and then using a translator to produce possibly-suboptimal code for the real processor. However, in this paper, we use DLX code when illustrating sample code.

The MTA instruction set is listed in Table 2. The core arithmetic and branch instructions represent mainstream RISC technology, and need no further explanation. The rest of this section is devoted to the description of instructions for multithreading.

The **fetch_next** instruction is used to fetch the next ready thread id from the ready queue. The ready thread id's *fp* value is stored in register %nfp (for *next frame pointer*) and the instruction pointer *ip* of the ready thread id is stored in register %nip (*next instruction pointer*) after **fetch_next** has executed. Most threads will end with the following instructions (which can be turned into a macro):

```
fetch_next
addi %fp, %nfp, #0
jr %nip
```

Synchronization instructions are used to spawn another thread or perform a synchronization on a sync slot. **Spawn** and **sync** are used to spawn and synchronize threads within the same function activation. The *ip* in the **spawn** instruction is combined with the value in the %fp register to form a spawn signal to the local SU. The synchronization slot value (*ss_off*) in the **sync** instruction is combined with the %fp register value for a sync signal sent to the local SU. **Xspawn** and **xsync** (X stands for "external"), are instructions which can be used for explicitly spawning and synchronizing with threads belonging to another function instance; thus the need for accessing the frame pointer value *fp*. If the *fp* value points to a frame in local memory, then a **spawn** or **sync** signal is issued to the local SU, otherwise, an **rspawn** or **rsync** message is formed with the appropriate field values and sent to a remote processor node.

Data transfer instructions are used to send data to a remote processor and perform either a spawn or sync at the remote processor. Thus, the EU processes these instructions by forming **data&spawn** or **data&sync** messages. The destination processor id of a message is extracted from the frame pointer operand *fp*.

The **get-spawn** and **get-sync** instructions are counterparts to the **get-spawn** and **get-sync** messages received by the SU. That is, the EU processes these instructions by forming **get-spawn** and **get-sync** messages and injecting them into the net. The id of the destination processor for a **get-spawn/get-sync** message is extracted from the address field *addr_val* of the respective **get-spawn/get-sync** instruction. The *fp* value of the **get-spawn/get-sync** message is simply the value in the %fp register when the **get-spawn/get-sync** instruction is processed. The *ip* operand of the **get-spawn/get-sync** instruction is used to fill the *ip* field of the **get-spawn/get-sync** message.

The **invoke** instruction is synonymous with the **invoke** message processed by the SU. The operand *dest_PE* specifies which processor node should execute the function activation and the rest of the operands are put into the appropriate fields of the invoke message.

3.3 Function Invocation

The aforementioned **invoke** instruction is used to invoke a function on a remote processor. The remote SU, upon receiving the **invoke** message, will allocate a new frame, load the parameters into the frame, and put the specified thread pointer into the ready queue. Thus, when the function begins executing, it will start with all its parameters in local frame slots. When the function returns, it must look up the *ip* of the caller and the ret-data-loc, and execute a **data&spawn** message.

The SU must allocate memory for the frame, since it can't simply add an offset to a stack pointer as in sequential machines. Since function invocation may happen frequently, software-based allocation routines would have too much overhead for quick activation of frames. A special mechanism for the quick allocation and deallocation of frames is crucial.

Therefore, the MTA will restrict the size of a frame to a small set of fixed sizes, such as powers of 2 or multiples of some number. The compiler will determine, for each frame allocation, the smallest number in this set that can contain the entire frame. For each size, the processor maintains a linked free-list, and a global pointer to the head of the list. Allocation and deallocation are done by removing blocks from, and adding blocks to, respectively, the head of the list. A special software trap can handle an empty free-list or a requirement for a frame that exceeds the largest fixed frame size.

Deallocation is done by the EU when the last thread in a function returns to its caller. This is also done by manipulating the free-lists. Since both the SU and the EU have access to these lists, lock variables are required to guard the free-lists so as to prevent erroneous memory management.

A local call may be carried out more efficiently, entirely within the EU, in order to take advantage of the registers (by using them for passing parameters) and the fact that no network messages need to be generated. To do the call, the caller loads certain registers with the parameters, and performs a **jal** (jump-and-link) instruction that jumps to the start of a thread in the callee. Control immediately passes to the callee, completely bypassing the synchronization mechanism. At this point, when the callee begins execution, the input parameters are in registers, the caller's return address is in %r31, and the caller's *fp* is in %fp. This is different from the situation that exists when a function is initiated by an SU that receives an invoke message.

Therefore, each function should have two different initiation threads, one for local invocations and the other for remote invocations. The local-invocation initiation thread should start with the following instructions (we assume that the free-lists are in addresses that are low enough to be loaded as an offset from %r0, which is hardwired to 0):

lw %r1, *free_n*(%r0)	;Global head of free-list of size n
bez %r1, *trap_n*	;What if free-list empty?
sw 4(%r1), %r31	;link new frame to caller-ip
lw %r31,0(%r1)	;next thing in free list
sw 0(%r1),%fp	;link new frame to caller-fp
sw *free_n*(%r0),%r31	;new free list head -> free list ptr
addi %fp,%r1,#0	;store new frame ptr into %fp

These seven instructions will create and link up the activation frame. At this point, the thread can either write the registers into local frame slots, or attempt to take advantage of the efficient register passing by doing some calculations with the input parameters. These instructions could be turned into a function setup macro, since they occur in the initialization thread of every function.

When a function returns, it must deallocate the activation frame and check whether the function was invoked locally or not. If it is, it simply changes the %fp register and jumps to the instruction pointed to by the caller-*ip* slot. Otherwise, it must issue a data&spawn message to a remote processor and send the proper return value. If the return value is in %r1, then the return would look like this (and could also be a macro):

```
        lw %r3, free_n(%r0)         ;Global head of free-list of size n
        lw %r2, 0(%fp)              ;retrieve caller-fp
        sw 0(%fp),%r3               ;link new block to free-list
        sw free_n(%r0),%fp          ;new free list head -> free list ptr
        andi %r3, %r2, local-mask   ;mask caller-fp with local-mask
        seqi %r3, %r3, local-pe     ;check if invoked locally
        bnez %r3, local
        lw %r3, 4(%fp)              ;retrieve caller-ip (bnez delay slot)
        lw %r4, 8(%fp)              ;retrieve return data loc.
        data&spawn %r4, %r1, %r2, %r3
        switch_thread
local:  addi %fp, %r2, #0
        jr %r3
```

If return values are always returned in the same register, then this sequence can be made into a macro.

4 Advanced Features of the MTA

There are a few features of the MTA which are not essential to basic multithreading, but which may be helpful in shortening the run time of certain programs, and in supporting nondeterministic parallelism. These are: explicit programmer/compiler control over thread scheduling, and support for killing threads that are scheduled or active. We briefly outline the mechanisms in this section.

4.1 Thread Priorities

In Section 2.4.1, it was mentioned that the SU will make certain threads jump to the front of the ready queue in order to take advantage of register locality. This can be expanded into a more general priority mechanism, in which each thread could be given a priority level. Thus, the compiler or programmer could identify critical paths in the code that need to be run quickly in order to maintain high parallelism, and parts that can be deferred until there is nothing else to do in the processor.

4.2 Support for Speculative Execution

Many non-numerical programs, and even some numerical programs, have the potential for big improvements in speed if some computation is performed *speculatively*, i.e., before the need for that computation has been determined [22]. There are also many applications in which the program can search down many paths in the search space simultaneously, but a success in any subsearch satisfies the search conditions, i.e., an OR condition. The subsearches can often be done in parallel. We would like to have the opportunity to explore the benefits of speculation and parallel search on a multithreaded machine, so a more advanced MTA will include features that make this efficient.

The generation of parallel searches and speculative threads is easy enough, but combining the results in a meaningful way is more complex, because nondeterminism is required. An efficient parallel search creates a race between the subsearches; the (temporally) first subpath to return a successful result is chosen as the solution to the next higher level of the search problem. The mechanisms discussed in section 3.1.3 can be used to support a nondeterministic merging of two results. For instance, a function can call many function instances on different processors, with all callee functions programmed to send a data&sync signal to the same slot in the caller frame. The reset count in the slot can be set to 0, so that only one result can get in, and all others will be ignored.

This kind of parallel search, and speculative execution in general, would be far more efficient if there were a mechanism for killing the computations whose results are no longer needed (e.g., all other subsearches still in progress). A function that wished to kill some of the functions that it invoked could send a signal to those function instances, which in turn would have to kill any processes it created, propagating down the activation tree until all descendents are killed. Most of this could be done using the Level 0 mechanisms, e.g., using spawn or data&spawn signals and special "killing" threads. However, at least one extra support mechanism would be needed: a command to remove a thread, or multiple threads belonging to a single activation frame, from the ready queue.

5 Methodology for Developing the MTA

The MTA is still at an early developmental phase. Much work remains to be done before a complete working system is realized. This section outlines our plans for developing the MTA system.

To explore the architectural features of the MTA model, we will choose a multiprocessor whose hardware configuration closely matches the MTA, and program it to *emulate* the behavior of the MTA. The emulation overhead should be much smaller than the cost of software simulation, so that programs can run at a speed close to the speed of the desired machine. This will allow us to experiment with real applications, not just "toy" programs.

The MTA instruction set will contain a standard complement of RISC instructions augmented by special instructions to support multithreading (see Section 3.2). If the target multiprocessor is built out of RISC chips, it should be possible to configure the compiler to use instructions that come from the regular instruction set of the target RISC chip. The special multithreading instructions won't be supported in the RISC chip, and will have to be converted to native instructions, either in the compiler back-end, the assembler, or in a special translator. The support hardware will also need to be emulated in the processor. This will probably involve special traps or calls to emulator routines.

The early experiments with MTA will probably use code which is partially or completely hand-compiled. A multithreading compiler will be needed in order to complete the system. Such a compiler is presently being developed in parallel with the emulator. This compiler is based on the McCAT (McGill Compiler-Architecture Testbed) compiler [10].

There are still some important architectural issues to be settled. Cache coherency is one. In this level of the MTA model, we have advocated the use of a local cache in the processor node to cut down on the average memory latency. We have not suggested the use of any hardware cache coherency mechanisms such as cache directory schemes [21]. At this time, programmers/compiler writers must fend for themselves when it comes to cache coherency.

The fine-grain synchronization capabilities of the MTA model must be used to keep the local caches coherent. We expect that in the multithreaded style of programming described in this report, the most common mechanism for data transfer will be explicit message-passing, e.g., **data&sync** operations. However, there are applications for which these mechanisms are not effective, and we wish to support these kinds of applications.

At this time, we have not chosen a specific cache coherency scheme. We expect that the experience we gain from our first experiments with the MTA will give us the data we need to make a more informed design choice.

6 Comparisons with Other Multithreaded Machines

Multithreaded machines are increasingly being viewed as an efficient hybrid model that combines the von Neumann processor's efficient instruction scheduling and use of locality with the dataflow processor's runtime flexibility. This has led to a multitude of multithreading designs [1, 2, 12, 15]. This section briefly outlines some of the differences between the MTA and some of the other designs.

The Empire [15], RWC-1 [19], and P-RISC[16] are/were projects aimed at creating multithreaded processors. These three machines are different from the MTA in that there is no clear separation between the execution unit and synchronization unit.

The MTA does not interleave threads as on processors like the Tera [2]. Moreover, Tera does not rely on caching to exploit data locality, but on one very large register file with multiple register sets.

In the APRIL [1], there is no separate synchronization unit. Moreover, the decision of which thread context to swap out from the multiple register set file cannot be aided by hardware as is done in the MTA with the Register Use Cache.

Some other machines such as *T [17] have separate synchronization and execution units. A key difference between the MTA and *T is that the synchronization unit is user-programmable in the *T, but not in the MTA. The user has some control over synchronization in the MTA, since the sync slots are accessible by the EU, but the SU is not programmable, leaving open the option of a special hardware implementation rather than a general-purpose RISC processor. Nevertheless, the hardwired synchronization support is sufficiently flexible to support nondeterminism and mutual exclusion, as well as the regular dataflow paradigm.

Finally, the Register Use Cache offers a unique way to handle the difficult problem of sharing register contents across thread boundaries, especially when threads are demarcated by long-latency operations in which the latencies cannot be predicted at compile time.

7 Conclusions

In this document, we have detailed the execution model of the Multi-Threaded Architecture (MTA). One of the main characteristics of the MTA is the division of the processor into a separate Execution Unit, which is programmed like a standard RISC processor, and a Synchronization Unit which provides hardware support for the synchronization of threads. The EU can rapidly switch from a completed thread to the next thread in its scheduling queue, which will lead to high utilization of the EU. The EU uses local registers to take advantage of data locality, and the SU supports this further by adopting a scheduling policy that favours threads that are likely to use the same registers. The Register Use Cache provides additional support for exploiting locality. Synchronization is based on the Argument Fetching dataflow paradigm, but it has been extended to support nondeterminism and mutual-exclusion. We have carefuly considered the implementations of these features and believe that they can be realized as special I/O devices and/or memory subsystems to a conventional RISC processor.

We have outlined the instruction set of the MTA to serve as an interface between the compiler writers and system implementation. We have also provided details of the operation of the EU and the SU, and mentioned extensions for compiler- or programmer-controlled prioritizing of threads, and for the killing of threads to support efficient speculative execution and parallel search.

We believe the combination of features described in this paper provide a good foundation for the efficient exploitation of parallelism on a large-scale multiprocessor. Our next ongoing phase is to implement the MTA emulator on our chosen multiprocessor. By analyzing the emulation results, in relation to the performance that can be predicted for hardware implementations of the special MTA support features, we can predict the performance of a fully-implemented MTA machine, as well as determine which hardware mechanisms are most important for achieving that performance.

8 Acknowledgments

We would like to thank the Natural Sciences and Engineering Research Council (NSERC) and MICRONET for their support of this research. The first author would also like to acknowledge the support of a Concordia FRDP Grant. Many ideas presented in this paper resulted from discussions with G. Ramaswamy, S. Nemawarkar, V.C. Sreedhar, and H. Ahmed.

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 1990.

[2] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. In *Conference Proceedings, 1990 International Conference on Supercomputing*, pages 1–6, Amsterdam, The Netherlands, June 1990.

[3] Arvind and Robert A. Iannucci. Two fundamental issues in multiprocessing. In *Parallel Computing in Science and Engineering*, number 295 in Lecture Notes in Computer Science, pages 61–88. Springer-Verlag, 1987. Proceedings of the 4th International DFVLR Seminar on Foundations of Engineering Sciences, Bonn, West Germany, June 25–26, 1987.

[4] David E. Culler, Anurag Sah, Klaus Eric 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 1991.

[5] Jack B. Dennis. First version of a data-flow procedure language. In *Proceedings of the Colloque sur la Programmation*, number 19 in Lecture Notes in Computer Science, pages 362–376, Paris, France, April 9–11, 1974. Springer-Verlag.

[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 1988.

[7] Jack B. Dennis and Guang R. Gao. The evolution of multithreaded computers. In R. Iannucci et al., editors, *Multithreaded Computer Architecture: A Summary of the State of the Art*. Kluwer Academic Publishers, Norwell, Massachusetts, 1994.

[8] Jack B. Dennis, Guang-Rong Gao, and Kenneth W. Todd. Modeling the weather with a data flow supercomputer. *IEEE Transactions on Computers*, 33(7):592–603, July 1984.

[9] Robert H. Halstead, Jr. and Tetsuya Fujita. MASA: A multithreaded processor architecture for parallel symbolic computing. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 443–451, Honolulu, Hawaii, May–June 1988.

[10] Laurie J. Hendren, Guang R. Gao, Chandrika Mukerji, and Bhama Sridharan. Introducing McCAT—The McGill Compiler-Architecture Testbed. ACAPS Technical Memo 27, School of Computer Science, McGill University, Montréal, Québec, September 1991.

[11] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 1990.

[12] 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, September 1992.

[13] Herbert H.-J. Hum. *The Super-Actor Machine: a Hybrid Dataflow/von Neumann Architecture*. PhD thesis, McGill University, Montréal, Québec, May 1992.

[14] Herbert H. J. Hum and Guang R. Gao. Supporting a dynamic SPMD model in a multithreaded architecture. In *Digest of Papers, 38th IEEE Computer Society International Conference, COMPCON Spring '93*, pages 165–175, San Francisco, California, February 1993.

[15] Robert A. Iannucci. Toward a dataflow/von Neumann hybrid architecture. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 131–140, Honolulu, Hawaii, May–June 1988.

[16] 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–June 1989.

[17] Rishiyur S. Nikhil, Gregory 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 1992.

[18] J. E. Rumbaugh. *A Parallel Asynchronous Computer Architecture for Data Flow Programs*. PhD thesis, Massachusetts Institute of Technology, May 1975. Also published as Technical Report MIT/LCS/TR-150, MIT Laboratory for Computer Science.

[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 1993.

[20] Burton Smith. The architecture of HEP. In Janusz S. Kowalik, editor, *Parallel MIMD Computation: The HEP Supercomputer and its Applications*, pages 41–55. MIT Press, 1985.

[21] P. Stenström. A survey of cache coherence schemes for multiprocessors. *Computer*, 23(6):12–24, June 1990.

[22] Kevin B. Theobald, Guang R. Gao, and Laurie J. Hendren. Speculative execution and branch prediction on parallel machines. In *Conference Proceedings, 1993 International Conference on Supercomputing*, pages 77–86, Tokyo, Japan, July 1993.

[23] Carl A. Waldspurger and William E. Weihl. Register relocation: Flexible contexts for multithreading. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 120–130, San Diego, California, May 1993.

A Overview of Multithreaded Architectures

Multiprocessors based on conventional von Neumann processors have some inherent limitations which usually makes large-scale parallelism hard to achieve [3]. Dataflow architectures [8, 6] seek to maximize flexibility and opportunities for parallelism by removing all constraints on the ordering of instructions except those constraints representing true data and control dependencies. Unfortunately, dataflow machines generally have very high synchronization costs, because instructions have to perform synchronizations amongst themselves to maintain proper execution order. Furthermore, with few constraints on the order of instruction execution, a dataflow processor cannot take advantage of temporal locality in the user program, leading to ineffective use of the memory hierarchy. *Multithreaded* machines are seen as a middle ground combining the efficient scheduling of von Neumann processors with the runtime flexibility of dataflow processors.

In broad terms, a multithreaded node architecture [1, 2, 7, 9, 13, 12, 15, 17, 19, 20] is best characterized as an architecture designed to support the efficient concurrent execution of multiple threads of control. A *thread* is defined as a sequence of instructions in which the order of these instructions, relative to one another, is determined at compile time. In other words, if two operations are in the same thread, then the order in which they can be executed is fixed. On the other hand, if two operations are in *different* threads, and there are no direct or indirect data or control dependencies between them, then the processor may execute either one before the other. There may be instructions from several threads which are all candidates for execution, and the computation within the node is carried forward by the execution of the enabled instructions. The execution of instructions within a thread may activate other threads, and when there are no more active threads in the parallel processing system, the computation is deemed terminated.

This broad definition covers many existing multithreaded architectures, which differ in their concept of a thread and how threads are scheduled. For example, some multithreaded architectures emphasize efficient support for very long threads ($\gg 100$ instructions per thread) where a program counter and processor state are maintained for each thread, such as in the HEP[20]. The processor states usually remain valid across long latency operations so that when a long latency operation has been completed, the suspended thread can be quickly switched into the processor for execution. In other multithreaded machines, the emphasis is on supporting shorter threads (less than a few dozen instructions). These short threads will be frequently initiated and terminated, due to their short length, so it is necessary for the processor to perform efficient synchronization among them. Usually, this involves a paradigm similar to dataflow, in which a thread becomes eligible for execution as soon as its required input data becomes available.

Another design issue of multithreaded architectures is whether interleaving of thread execution is supported. Interleaved thread execution is intended to exploit greater amounts of parallelism with less complexity than other processor designs. Processing elements that support the interleaved execution of threads use instruction execution pipelines designed to achieve high resource utilization by sharing the computational resources (floating point units, for example) among several threads. Each pipeline stage may perform work for a different thread on each clock tick. In this way, latency of operations in one thread need not lead to low utilization of pipeline stages.

In multithreaded processors without interleaved instruction execution, the instruction execution mechanism is conventional and the novelty lies in the mechanism used to "switch context" among threads awaiting execution and the synchronization between such short threads. The objective is to reduce the time a thread must run to amortize the cost of switching control to a new thread, thereby permitting a finer grain of processing without incurring overwhelming overhead costs. This scheme has the advantage that it can be more easily implemented with conventional microprocessor technology.

The MTA model does not assume the hardware support of thread interleaving. This is in order to fulfill our basic objectives. However, the model does make use of hardware support for efficient thread switching.