

Developing a reference implementation for a microgrid of microthreaded microprocessors

Mike Lankamp



Universiteit van Amsterdam

Developing a reference implementation for a
microgrid of microthreaded microprocessors

by

Mike Lankamp

Submitted to the faculty of science in partial fulfilment of the requirements for the degree of

Master of Science

at the

University of Amsterdam

August 2007

Supervisor.....

Prof. Dr. Chris R. Jesshope

Board of Examiners.....

Dr. Peter van Emde Boas

Prof. Dr. Chris R. Jesshope

Dr. Andy D. Pimentel

Abstract

The goal of this master's project was to implement a cycle-accurate software simulator for a new processor architecture model called "Microthreading". Microthreading aims to provide scalable latency-tolerant concurrency on a microgrid of microprocessors. The simulator was required to accurately simulate a microthreaded architecture at the cycle level so detailed performance analysis and eventually power usage analysis could be undertaken on the model and pieces of code.

The result of the project is a simulator based on the Alpha ISA able to load and simulate compiled code and allow for the state of the components of the virtual processors to be examined at every cycle. During development of the simulator, various issues with the microthread model were uncovered and fixed. As part of the verification of this simulator, it is shown that the microthreading alterations to a generic RISC architecture cause speedup when using multiple processors.

Contents

1. INTRODUCTION	9
2. RELATED WORK	11
2.1 OUT-OF-ORDER EXECUTION	11
2.2 VLIW.....	11
2.3 EPIC	12
2.4 MULTISCALAR	12
2.5 INTHREADS	13
2.6 TRIPS & EDGE.....	14
2.7 WAVESCALAR	14
3. MICROTHREADING	15
3.1 FAMILIES.....	15
3.2 THREADS.....	16
3.3 THREAD SYNCHRONIZATION.....	19
3.4 FAMILY SYNCHRONIZATION.....	21
3.5 FAMILY TERMINATION AND PRE-EMPTION	22
4. THE REFERENCE MODEL	25
4.1 THE FAMILY TABLE.....	25
4.2 THE THREAD TABLE.....	28
4.3 THREAD STATES.....	29
4.4 THREAD LISTS	30
4.5 FAMILY AND THREAD ALLOCATION	31
4.6 MEMORY AND CACHES.....	35
4.7 INTER-PROCESSOR SHARED EXCHANGE.....	37
4.8 FAMILY TERMINATION, SYNCHRONIZATION AND CLEANUP	40
5. THE SIMULATOR	43
5.1 GENERAL STRUCTURE.....	43
5.2 CYCLE PHASES	44
5.3 PORTS AND FUNCTIONS.....	45
5.4 ATOMICITY.....	45
5.5 VERIFICATION AND VALIDATION	47
6. RESULTS	49
6.1 SETUP.....	49
6.2 MATRIX MULTIPLICATION #1	50
6.3 MATRIX MULTIPLICATION #2.....	51
6.4 MATRIX MULTIPLICATION #3.....	52
7. CONCLUSION	55
REFERENCES	57

1. Introduction

“To perform the world's fastest computation, I divided and evenly distributed the calculations among the 65,536 processors and then squeezed the most performance from the each processor.”

- Philip Emeagwali

Moore's law describes the growth in chip transistor density, with a doubling every 18 to 24 months and, according to the semiconductor industry's predictions, looks set to do so for at least the coming decade and possibly longer [1]. This growth will pose major problems for computer architecture in this time frame. The problems arise from current approaches, which do not scale well and have used clock speed rather than concurrency to increase performance. This has given rise to excessive power dissipation and circuit complexity. The reason that processor developers relied on clock speed was so their processors could maintain source code and binary compatibility whereas switching to concurrency techniques would break this compatibility. Only recently are continued clock speed improvements no longer trivial to achieve and, consequently, processor developers have been forced to look into other directions.

Modern desktop processors attempt to extract concurrency from a sequential instruction stream through mechanisms such as out-of-order execution which requires scoreboards, re-order buffers and complex data-dependency analysis logic in hardware. Complex branch prediction logic is required to avoid the cost of branches due to the increasing pipeline lengths. All of this logic consumes significant surface area and power and means that microprocessors have become large, monolithic entities. Combined with increasing clock speeds, this leads to situations where chip designers are faced with a physical bound, where a clock signal cannot propagate across the entire chip in a single cycle.

For these reasons, new microprocessor designs need to emphasize locality, concurrency, scalability and power consumption.

Multi-threaded approaches to microprocessors have been studied and it was shown that Chip Multiprocessors show promise as long as they are provided with enough parallelism [2]. However, this creates the problem of presenting enough parallelism to the processors for them to be effective. This is a combined software and hardware problem. The software, typically the compiler, must extract parallelism from a sequential code stream and encode it explicitly in the instruction stream or the programming language must allow the programmer to specify concurrency. A combination of both methods is also not uncommon. The hardware must be able to efficiently execute the concurrency encoded in the instruction stream.

Microthreading aims to provide a solution to the hardware issues by allowing the software to explicitly encoded loops or other concurrent fragments of code in the instruction stream in a high-density form. A microthreaded processor executes instructions from multiple threads interleaved on a single-issue pipeline. The software does not directly create threads, but instead creates a group of threads called a *family*, whose properties, such as the number of threads are stored in the instruction stream by the software. Because the properties of a family are independent of the hardware configuration, scalability is easily achieved by distributing

the set of threads in a family across the number of available processors. This distribution of threads is determined by the hardware at run time based on the static properties of the family and the number and configuration of available processors, enabling scalable speedup with binary code compatibility. On a single microthreaded processor, the pipeline can switch between its assigned threads without delay, thus keeping the pipeline at full effectiveness. Long-latency operations such as memory reads are efficiently supported by decoupling the operation into an *issue* and a *writeback* part. This allows the pipeline to continue executing from the same or another thread while the data is being fetched from memory.

2. Related Work

“You have to know the past to understand the present.”

- Carl Sagan

Many approaches have been and are being used to increase the performance of microprocessors, each with their benefits and downsides. This chapter describes the most successful techniques and highlights the principle differences when compared with microthreading.

2.1 Out-of-Order Execution

Out-of-order (OoO) execution is a technique used by most modern superscalar processors to extract Instruction Level Parallelism (ILP) from a sequential instruction stream by taking advantage of the fact that multiple instruction in a certain instruction window are independent and can be issued in parallel. It works by fetching multiple instructions at once and executing independent instructions concurrently by having multiple functional units in the pipeline. To ensure sequential semantics on the instruction stream, the processor has to take care that no instruction overwrites the results from instructions which appear in the instruction stream. This requires some form of data dependency analysis on the instructions and a way to delay the instructions or order the results when they have completed.

Although OoO offers speedup for a sequential instruction stream while remaining completely backwards compatible with binary code, this comes at great cost in the form of complex hardware and power consumption [3]. These costs, combined with the limited parallelism available in sequential instruction streams [4], make OoO a highly non-scalable technique. Microthreading is fundamentally differently from OoO in that it can use a simple in-order pipeline with straightforward issue logic because concurrency is not extracted from a single instance of a sequential code stream, but from multiple independent instances of the code stream which can be interleaved on a single pipeline or run concurrently across multiple pipelines in multiple processors.

2.2 VLIW

Very Large Instruction Word is a mechanism to exploit ILP in cooperation with the compiler. Instead of each instruction word representing a single instruction operating on a single piece of data, instruction words in a VLIW architecture contain a fixed number of multiple instructions that each operate on single piece of data, thereby performing multiple operations in parallel. Since each instruction in an instruction word has to be independent of the others, the compiler is required to identify which operations can be performed in parallel and construct the instruction stream accordingly. If not enough instructions can be bundled into an instruction word, it must be padded with no-ops.

Traditionally, VLIW programs are statically scheduled on the processors, made possible by the compiler having full knowledge of the latencies of the instructions on the processor,

something that severely restricts binary compatibility of VLIW programs. Furthermore, being completely dependent on the latency of the processor, VLIW cannot efficiently deal with the large and unknown latencies involved in memory accesses.

Even though VLIW can be combined with superscalar processor techniques such as pipelining and OoO execution to achieve even greater performance, these improvements also come at the cost of significantly more hardware [5]. And despite this, VLIW itself is still bound to a maximum level of parallelism as defined by the architecture as the width of the instruction word. This makes VLIW unsuitable for scalable performance; when new systems with an increased number of functional units become available, old code, which has been compiled for a lower number of functional units, cannot be run on these systems.

2.3 EPIC

Explicitly Parallel Instruction Computing (EPIC) is a mechanism researched in the 1990s and has been implemented by Intel and HP in the IA-64™, Itanium™ and Itanium 2™ processors [6]. It was designed to increase parallelism by cooperating with the compiler but improve upon the disadvantages of VLIW. An EPIC processor executes statically scheduled multiple-operation instruction words in a pipeline just like VLIW. EPIC deals with branch delays by using features found in superscalar processors and performs speculative or predicated execution for the instructions after the branch. Memory latencies are ‘tolerated’ by delegating control of caching to the compiler. It can specify what cache a datum in a memory load should end up in as well as issue data-prefetch instructions. This control allows the compiler to make reasonable estimates for the latency of memory loads and adjust the static schedule accordingly.

The problems with EPIC all stem from its VLIW roots and are mainly code-related. Firstly, although older EPIC architectures are compatible with new code, this is not the case with old code. In Intel’s case, the IA-64 ISA is completely incompatible with its previous IA-32 ISA, used in all of its previous processors [7]. The processor also has to detect when new code makes use of code (e.g., libraries) written for an older EPIC ISA and switch modes on the fly. Secondly, if code does not contain enough parallelism to fill an instruction word with independent operations, space in the instruction stream is wasted and the code size will grow. Furthermore, EPIC’s and the compiler’s usage of predication and speculative execution and loading increases hardware and software complexity, can result in unpredicted performance and dissipates energy.

2.4 Multiscalar

A Multiscalar processor [8] works by having the compiler split up a program into chunks called *tasks* and executing the tasks concurrently on different processing elements. The processing elements, which are connected with a unidirectional ring network, operate like simple in-order pipelined processors. The tasks can be different pieces of code or different instances of the same piece of code, such as iterations in a loop. The main instruction stream is annotated with a secondary instruction stream, which contains information on register synchronization and task termination for every instruction in the main instruction stream. The register synchronization bits are set by the compiler to inform the processor when to send register values from one task to the next. The process element containing the task that reads a

shared register will stall until that register has been written by the previous thread. Synchronization between tasks via memory to ensure sequential semantics is done by monitoring past and outstanding loads and stores, and squashing a thread if it turns out it loaded a value before a thread preceding it had written it. Tasks that represent iterations are speculatively dispatched to processing elements and excess tasks are squashed when the loop exit condition becomes known.

Multiscalar has, at a first glance, many similarities with micro-threading. In both architectures the compiler must separate the program into blocks of annotated code which can run concurrently on multiple simple, in-order pipelined processors. The difference, however, lies in the fact that with Multiscalar, only a single task can execute on a processing element at a time. If, because of a data dependency, the task is forced to stall, the entire processing element is wasting cycles. Micro-threading, on the other hand, can continue executing instruction from other threads mapped to the same processor. This allows micro-threading to take a much more conservative synchronization approach by suspending a thread when its data is not yet available. In the event that all threads are suspended, parts of the microthreaded processor can be powered down to conserve energy. Multiscalar has no explicit synchronization on memory and its run-time per-address data dependency analysis is a form of speculative execution, wasting time and energy when it squashes tasks. Micro-threading can bulk-synchronize on families, ensuring that memory loads from subsequent threads always use the memory written by the preceding threads.

2.5 Inthreads

Inthreads [9] is an architecture that provides explicit synchronization primitives and instructions in the form of binary semaphores and *wait*, *signal* and *clear* instruction that act upon them. The multiple-issue pipeline dedicates each fetch stage to a single thread and has a special stage that stalls threads when it waits on a semaphore that isn't signaled. Once another thread signals the semaphore the pipeline resumes executing from the stalled thread. Once a thread has passed this wait stage, it is processed by a normal out-of-order pipeline with a shared register space and possibly shared functional units.

The compiler has to explicitly create threads, designate thread identifiers and map the thread's registers in the shared register space such that private registers of concurrent threads don't overlap. Communication between threads is done by the compiler mapping shared registers to the same location in the shared register file and inserting proper semaphore instructions to effect synchronization.

Although Inthreads offers low-cost synchronization primitives, the maximum concurrency is still limited by the issue width of the pipeline and when a thread is stalled on a semaphore, no other thread can be run in its place. The compiler also needs intricate knowledge of the hardware, adjusting the number of threads to the issue width of the pipeline. This could hinder backwards or forward compatibility. The threads themselves are relatively long-lived, compared to the Micro-threading model. They are used mainly to easily balance the load of a loop across the width of the pipeline. This load balancing has to be hardcoded by the compiler and there seems no way to dynamically adjust this depending on available resources or hardware structure.

2.6 TRIPS & EDGE

Explicit Data Graph Execution (EDGE) is an architecture that has the compiler explicitly encode a data graph in the compiled code. The EDGE architecture has been implemented in TRIPS at the University of Texas [10]. The compiler segments sequential code into hyperblocks, which consist of 128 dataflow-executed instructions. The result of each instruction is sent directly to its consumers within the hyperblock, leaving the register file just to store results between hyperblocks. TRIPS features 16 execution cores per processor, with multiple processors on a chip. Hyperblocks are mapped to cores such that each core in a processor has at most 8 instructions mapped to it. TRIPS sequentially and speculatively executes hyperblocks on a chip's processor.

In effect, TRIPS, and the EDGE architecture, consists of multiple 16-wide out-of-order superscalar processors on a chip that can hold up to 1024 instructions in flight. However, EDGE is only a dataflow model inside of hyperblocks. The hyperblocks themselves are still sequentially and speculatively executed. As a result, this model is only scalable up to the size of the hyperblocks. For increased scalability, the hyperblocks must be enlarged, which prevents previously compiled code from being run on the new hardware, limiting binary compatibility.

2.7 Wavescalar

Wavescalar [11] is a pure data flow architecture that uses data tags, called *wave numbers*, to differentiate between the data from multiple loop instances. It also uses these wave numbers and explicitly annotated memory instructions to order memory requests and thus enforce sequential memory semantics. The instructions are cached on the processing elements themselves and send their results directly to their consumer instructions on the same or nearby processing elements.

Where instructions are placed on a grid of processing elements is worked out by both the compiler and hardware. The compiler groups dependent instructions into segments which will be placed on a single processing element by the hardware. Subsequent segments are placed on nearby processing-elements by the hardware with a simple fill algorithm.

The disadvantage of Wavescalar lies in the fact that it severely constrains memory operations into a sequential ordering. This constraint allows the processor to utilize only limited amounts of the concurrency encoded in the instruction stream.

3. Microthreading

“Any sufficiently advanced technology is indistinguishable from magic.”

- Arthur C. Clarke

Microthreading is a model originally devised in, and refined since, 1996 that defines hardware support for multiple, dynamically scheduled threads to increase concurrency in a generic RISC architecture [12,13]. It is a model that can be applied to any RISC-like architecture and instruction set and yields scalable speedup for the same binary code.

A microthreaded processor consists of threads and families. In regular RISC architectures, there is a single program counter and a handful of registers, typically 32, as this number is limited by the instruction size. In effect, the program counter and registers represent a single thread.

In the microthreading model there are multiple threads, each with its own state, including a program counter and registers. Threads are executed on a data flow basis, where instructions whose operands are not available cause the thread to suspend until the operands are available. Combined with single-cycle thread switch times, this avoids any unnecessary stalling of the pipeline resulting in a higher instruction throughput and improved power efficiency.

Families describe a group of homogeneous threads by a fixed set of parameters such as number of threads and each thread's properties. Families are defined and created by the program code and a family's threads are dynamically distributed among a group of processors, thus yielding the potential for increased concurrency and speedup when more processors are available.

Since the microthreading model is an extension, rather than a replacement, to the instruction set, binary compatibility with existing code remains, although without the same benefits of speedup. This compatibility is implemented by creating a single thread in a family for legacy code and executing the entire legacy program from that thread.

The components of a microthreaded processor will now be discussed in more detail.

3.1 Families

A family describes a collection of homogeneous threads; each thread has the same number and type of registers and starts executing from the same address. To be able to distinguish between threads in a family, each thread has an index in the family, which is written to a thread's first register when it is allocated on a processor.

A typical example of a family is a for-loop with N iterations to perform some operation on an array. Here, the family defines N threads, where each thread executes a single iteration of the loop using the value written to their first register to calculate the address into the array to use for memory loads and stores.

A family is created by a thread, thereafter called the *parent thread* of that family, with a CREATE instruction, which takes the memory address of a structure called a *Thread Control Block* as an operand. This *TCB* describes the properties of the family, such as the number of threads to create, where to start the threads' execution and how many and what types of registers should be allocated for each thread.

The microthreading model makes a distinction between *local* families and *non-local* families. For the former type of families, all threads of the families are executed solely on the processor it was created on. The latter distributes the threads among a set of processors when the family is created by having each processor instantiate a subset copy of the family. This subset is determined deterministically from the family's parameters, the number of processors used and a processor's position in that set. A typical distribution might be block cyclic, where threads 0 to $N-1$ run on processor #0, threads N to $2N-1$ run on processor #1, and so on.

Created families are stored in a processor's *Family Table*, with each row representing a single family. The *Family Identifier*, or *FID*, of a family is mainly composed of its row-index in the Family Table and to ensure communication between processors can identify the same family, a non-local family has identical FIDs on each processor. Local families do not have copies on other processors to communicate with; therefore these families do not need to be placed at a globally unique FID. During family allocation, a protocol between processors must ensure that the copies of a family on other processors get allocated to the same row in the Family Table. The implementation of this protocol in the reference implementation will be discussed in the next chapter.

The parent thread can pass arguments to the threads of the family in two ways: using *shared* registers, which are explained in the next paragraph, or *global* registers. Global registers contain loop-invariant values accessible to each thread in the family as read-only registers mapped in their local register space. This allows the parent thread to pass array addresses and other such information to the newly created threads.

3.2 Threads

As described above, a thread has its own state, consisting of at most 32 registers and a program counter. Most threads in the microthreading model are generally short-lived, in the order of dozens of instructions at most. Threads of no more than 5 instructions are not uncommon as well.

Communication between threads occurs in two ways. Via conventional, asynchronous memory through memory reads and writes which are synchronized on family termination or via synchronizing memory, typically registers. The latter occurs by mapping a section of a thread's registers to another thread's registers. These mappings can be divided into three classes, each used to communicate between threads in a different way.

To achieve these communication-mappings, a thread's 32 registers are divided into 4 logical classes: the globals, shareds, locals and dependents. The rest of the register window is implicitly filled with read-as-zero registers. Figure 1 shows a thread's register space and the location of these classes within it. Note that there can be more than one read-as-zero registers, depending on how many registers the thread actually requires.

Of the four classes, the locals are the simplest; these registers are private to the thread itself and cannot normally be used for communication with other threads; each thread has its own set of locals that can be read and written to as often as desired. The other three classes are used for inter-thread communication and are slightly more complicated.

The first of these classes, the *shared* registers, or *shares*, are registers in a thread that map to registers in the register space of the next thread in the family. This creates a unidirectional communication channel between threads. The registers in the next thread that the shares are mapped to are called the *dependents*. For synchronization reasons, the shares should only be written once and the dependents should only be read once. This mapping between shares and dependents is deterministic and exists during the entire lifetime of both threads. The number of shared registers required for threads in a family is defined in the TCB.

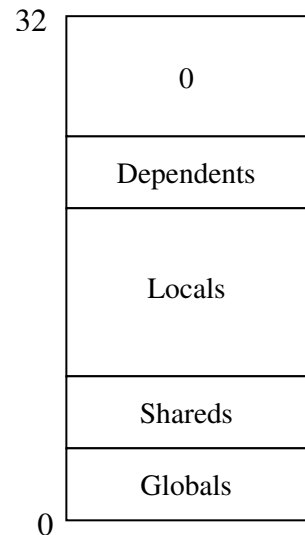


Figure 1 - Thread registers

With the shared-dependent mapping between threads, there is the issue of the first and last thread in a family, that have no previous and next thread to map from and to, respectively. Here, the second class of thread communication registers applies; the *child-shares*. This is a mapping similar to the shared-dependent mapping between threads in a family, only these registers map to and from registers in the family's parent thread's local registers. This allows the parent thread to initialize the 'chain' and fetch the result at the end. To the threads in the family, the distinction is completely transparent, so the first and last threads can read from its dependents and write to its shares as usual. The parent thread specifies, via bits in the create instruction, the offset in his local registers he wants the child-shares to be mapped. This mapping then continues to exist until all threads in the family have terminated.

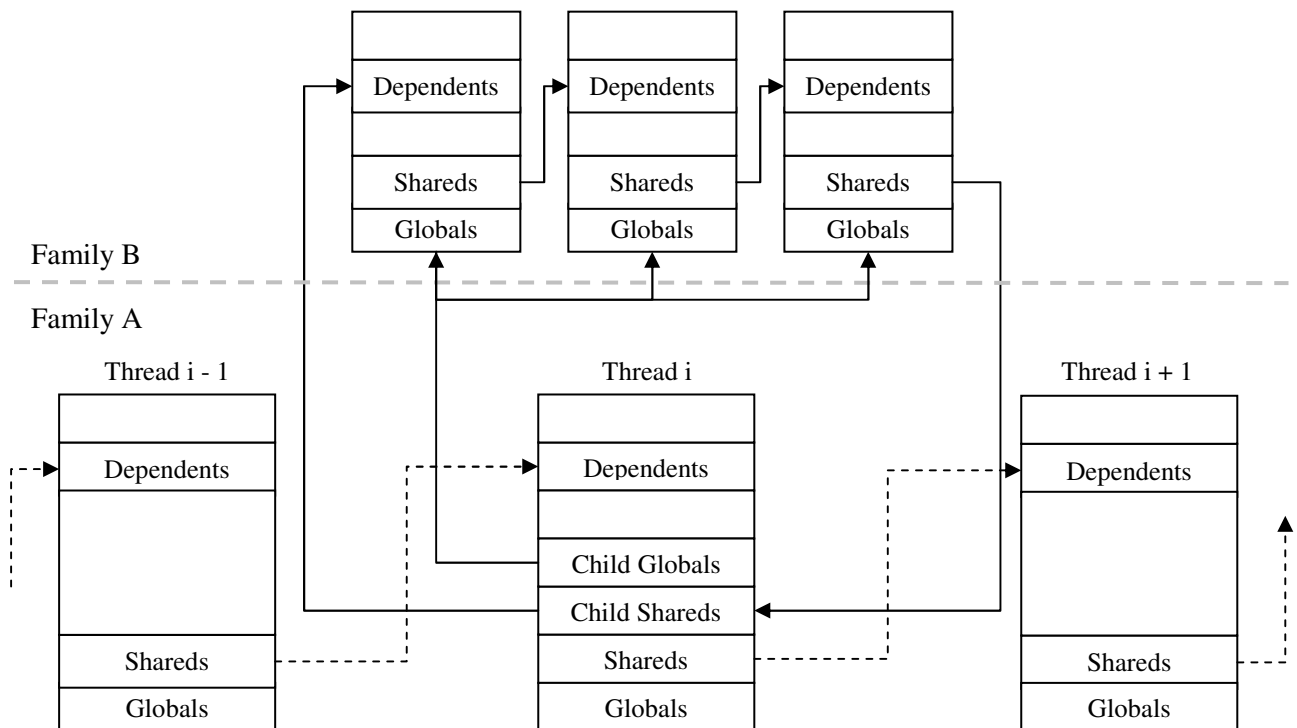


Figure 2 - Relation between shares and dependents between threads and families

The third, and last, class of thread communication registers is *globals*. This is a mapping between registers in the parent thread and read-only registers in every thread in the family. Globals contain thread-independent values, such as base memory addresses.

Figure 2 shows how three classes of registers work together when creating a family. Thread *i*, part of family A, has its own shareds and dependents, which map to the previous and next threads in family A. At some point the thread executes a CREATE instruction and creates family B, containing three threads. Registers from the parent thread are mapped to each thread to form the globals and the initial shareds and final dependents. Figure 3 shows this mapping in the parent thread by showing the parent thread's register space before and after the create.

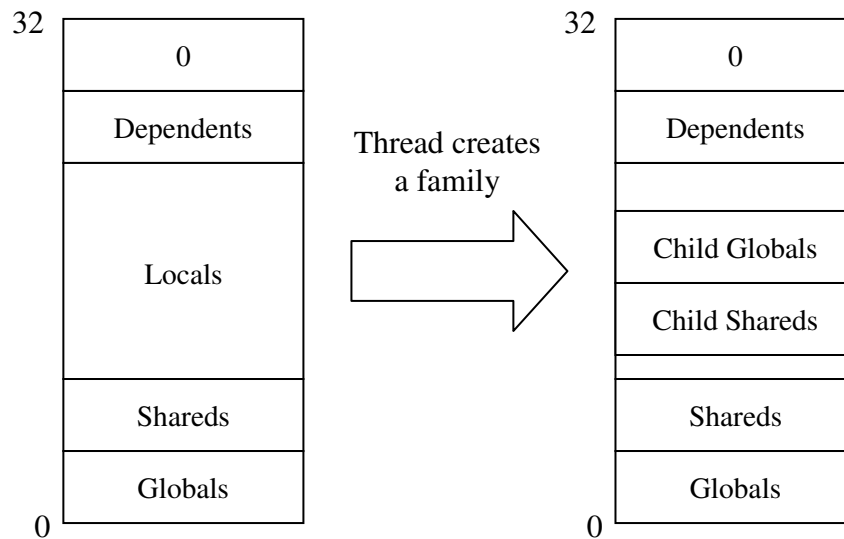


Figure 3 - Example of register reclassification on family creation

An example to illustrate the usage of these three kinds of registers is the following piece of C code:

```
int sum = 0;
for (int i = 0; i < N; i++) {
    sum = sum + A[i];
}
```

This piece of code sums all elements of array A. When compiling this to a microthreaded instruction set, the code would create a single family with N threads, one global value and one shared-dependent mapping per thread. The parent-thread would initialize the chain with value 0. Each thread would then read its index, add it to the value of the global register to construct an address, load the memory from that address, read its dependent register, add the loaded value to that value and write the result to its shared register. Eventually, after all threads have executed, the parent thread would receive the sum. Each thread is free to execute concurrently but synchronization on the shared registers provides an order over the additions that comprise the summation.

A processor manages its threads in its *Thread Table*, a table where each thread is a row, and each piece of state is a column. From this we define the *Thread Identifier* of a thread as the thread's row index in the table.

Threads terminate their own execution by executing an instruction with the END tag. Once all threads of a family have terminated this way, the family is considered terminated and its termination is passed on the parent thread of the family. The way by which the parent thread can monitor this event is covered in the next sections.

3.3 Thread Synchronization

Any system that deals with multiple threads needs to define a synchronization mechanism to eliminate non-determinism. In the microthreading model, this is implemented in a single way, by adding state bits to each register in the register file. These bits indicate whether the register is full, empty, or is being waited upon. Instructions that read a register that is empty cause their thread to suspend and wait on the register until it is full. This is done by marking the register as being waited on and writing the Thread Identifier into the register. Later, when the register is being written to, the Thread Identifier is first read from the register and used to activate the thread.

This mechanism provides a non-blocking implementation of synchronization, allowing the pipeline to continue executing instructions from other threads. This is opposed to conventional in-order pipelines, which stall completely when one or more operands are not available.

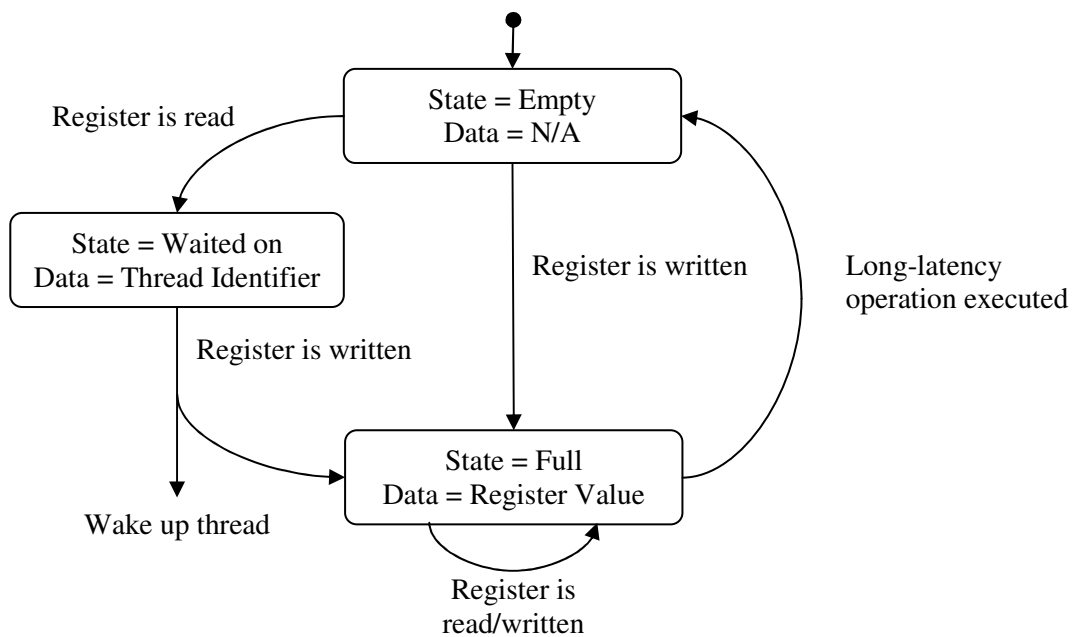


Figure 4 - Register state transitions

However, in such a case where an operand is not available for the current instruction, any instructions from the same thread following it in the pipeline must not be allowed to complete; the pipeline must be flushed at the cost of several cycles, depending on the pipeline implementation. To avoid this, instructions which could cause a pipeline flush are annotated with a so called *SWITCH* tag, indicating that the pipeline should switch to, and start fetching from another thread. This will ensure that no pipeline stages will have to be flushed due to a missed data dependency, keeping the pipeline as full as possible. The process of tagging

instructions in the instructions stream is implementation-specific and in the next chapter, a specific approach will be discussed.

As discussed, the general cause of a pipeline flush is a missed data dependency in the form of an empty register. Assuming a correct program, this comes from using the result of a long latency operation before that operation has completed. A typical example of this is a memory read. In the microthreading model, long-latency operations are decoupled and do not cause the pipeline to stall. Once executed, long-latency instructions dispatch their operation and mark the resulting register as empty at writeback. Once the decoupled operation has completed, it is asynchronously written back into the register file, triggering the wakeup mechanism described above. Therefore, those operations need to be tagged with the target register address to indicate to where they should write the resulting value upon completion.

In the microthreading model, reading from synchronizing registers is strictly speaking also a long-latency operation. A thread reading from one of its dependent registers can be seen as similar to reading the result of a memory load. Both registers are empty until the data has been asynchronously written, after an indeterminate amount of time. Family creation and synchronization also follows this model. When the family is created, the target register is cleared. Once the family has terminated, the target register is written. This allows the parent thread to synchronize on the termination of a created family by simply reading the target register. Also, aside from instructions using the result of long-latency operations, jumps and branches also cause a pipeline flush and must therefore also be annotated with the SWITCH tag.

As an example, the thread representing a single iteration in the previous code segment is shown here, in the Alpha ISA:

```
s4addl $R2, $R2, $R0
ldl    $R2, ($R2)
addl   $R1, $R3, $R2
swch
```

Each thread must read its element of the array A, add it to its dependent register, and write the result to its shared register. In this case, R0 holds the address of the array A as a global register, R1 is the shared register, R2 holds the thread index i and R3 is the dependent register. First the code uses the `s4addl` instruction to scale the index with the size of each array element (32 bits) and add it to the base address of the array. The resulting address is used to dispatch a memory load with the `ldl` instruction. This will set the target register R2 to empty. The next instruction attempts to add the read value to the dependent register and write the result to the shared register. If the pipeline, after reading the source operands, determines either operand is empty, it will write back the thread identifier to the empty register. If both are empty, it will write to one of them. Once this register is written at a later time, the thread will be activated. Once the thread enters the pipeline again and both operands are available, it calculates the results and writes it to the shared register. Should the other register still be empty, the thread suspends again, on the other register.

Because either operand in the `addl` instruction is the result of a long-latency operation, it has to be annotated¹ by a switch to prevent the performance loss from flushing the pipeline. The

¹ Here, the switch is displayed as an instruction following the annotated instruction. This is only for illustration purposes and does not necessarily have to be the case in the binary instruction stream after compilation.

switch will cause instruction from another thread to enter the pipeline after the *addl* instruction has been fetched.

3.4 Family Synchronization

Family synchronization is a mechanism to allow a thread to synchronize on any of the families it created. It uses the same basic mechanism from thread synchronization, by suspending the thread on a register until it is written. However, in this case the long-latency operation can be considered to be the family execution. The operation is ‘dispatched’ by issuing the create instruction. The asynchronous writeback is done once the family is terminated.

A family is considered terminated once all threads have executed to completion and all memory writes are guaranteed to be visible to the parent thread, and any subsequent families it might create. This implicit memory-barrier is in effect a method of bulk-synchronization on the conventional memory.

This mechanism allows a thread to spawn one or more families which run concurrent to the thread itself. Once the thread requires the result from the family, either in one of its child-shares, or in the memory that the family wrote to, it issues a *Sync* instruction, which is nothing more than a normal instruction (such as *add*) that reads from a register that holds a return code stored by the hardware on family termination. In between the create instruction and the sync instruction, the thread can perform other operations, including creating additional families.

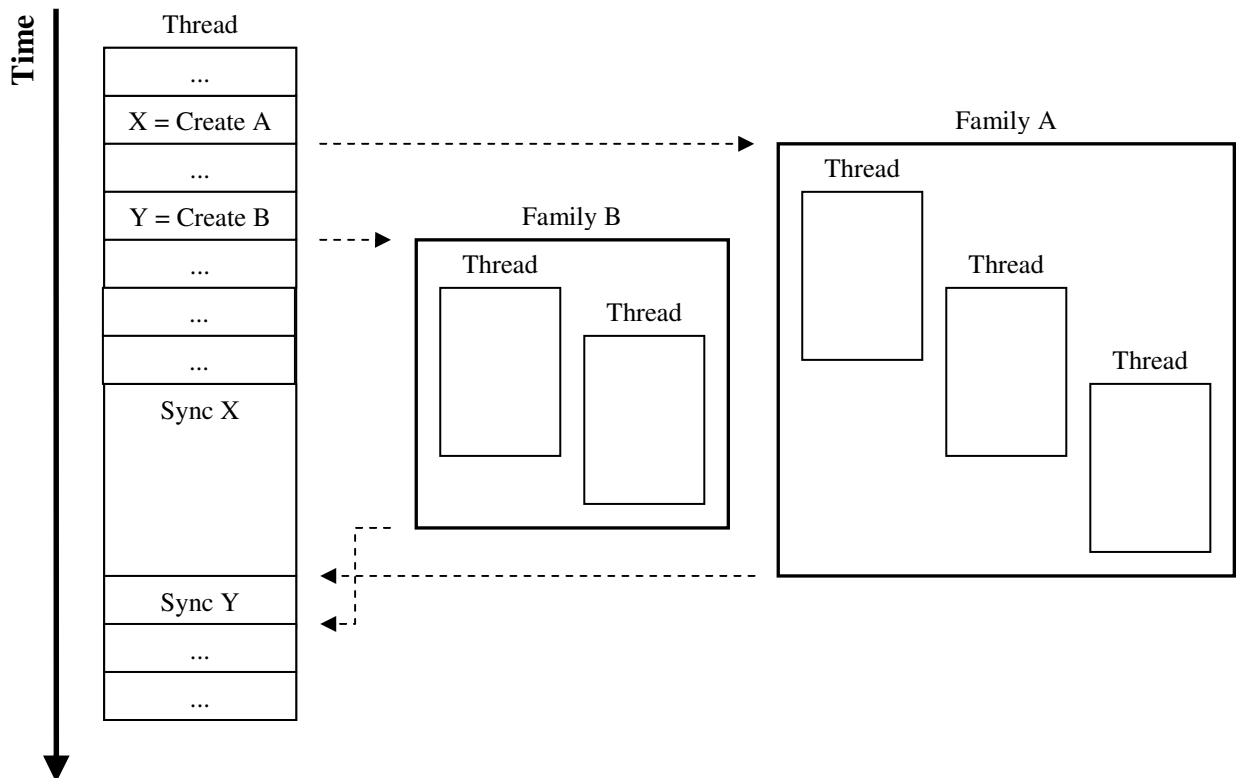


Figure 5 - Family synchronization example

An example situation is shown in Figure 5, where the parent thread creates two families, A and B, performs several other operations and then synchronizes on A and B, respectively. Note that the synchronization on A is executed before family A has terminated, causing the thread to suspend on register X until A has actually terminated and writes to X. Also note that B terminates before A does, causing the synchronization on B *not* to suspend because Y has already been written to before the sync instruction is executed.

3.5 Family Termination and Pre-emption

Normally, all threads in a family run to completion after which the family is considered terminated. However, there are also two mechanisms to terminate a family and one mechanism to pre-empt one. All three methods cause premature family termination, yet the parent thread of the family can, and will, still synchronize on the family as usual. A special return value, called the *exit code* of the family, is written to a register in the parent's locals upon family termination that indicates in which of the four² ways the family was terminated. This register is specified when creating the family.

The BREAK instruction is the first mechanism to terminate a family. When executed by a thread of a family, this instruction immediately and unconditionally terminates all threads in the family, thus terminating the family itself. However, besides simply terminating the family, the thread executing the instruction can specify a value as an argument to the BREAK instruction that will be written back to a register in the parent thread. This register is different from the exit code register, and is called the *exit value* register. This effectively allows one to create a family of threads to search a parameter space and return a certain value of interest.

Note that, when multiple threads can execute a BREAK instruction, no guarantees are made as to which exit value will be written back to the parent thread. Consequently, the use of BREAK in a family of unconstrained, concurrent threads searching a parameter space effectively yields an “any result is fine” return value. If, however, one wants to find, say, the *first* match in a parameter space, based on some ordering, the threads must be constrained in some way. This can be done by passing a value from thread to thread in a shared register. An example of this situation is shown in the following code fragment, where the first occurrence of the value “12” is searched for in an array.

```
s4addl $R3, $R2, $R0
ldl    $R3, ($R3)
cmpeq  $R3, 12, $R3
swch
beq    $R3, not_found
swch
addl   $R31, 0, $R4
swch
break  $R2

not_found:
addl   $R1, $R4, $R31
end
```

² The two termination mechanisms, the single pre-empt mechanism, and normal completion.

In this case, the global \$R0 holds the address of the array, \$R1 is the shared register, \$R2 is the local with the thread index, \$R3 is a temporary local and \$R4 is the dependent register. First, the thread index is scaled by 4 and used to index the array to load the element. Then, the loaded value is compared to 12. \$R3 will be set to 1 if \$R3 equals 12 or to 0 otherwise. If \$R3 equals 0, the thread jumps to *not_found*, where it reads the dependent and writes the shared as last instruction. If, however, \$R3 is 1, meaning the read value equals 12, the thread reads the dependent register (and ignores the actual value read) and breaks with the thread index in \$R2 as exit value.

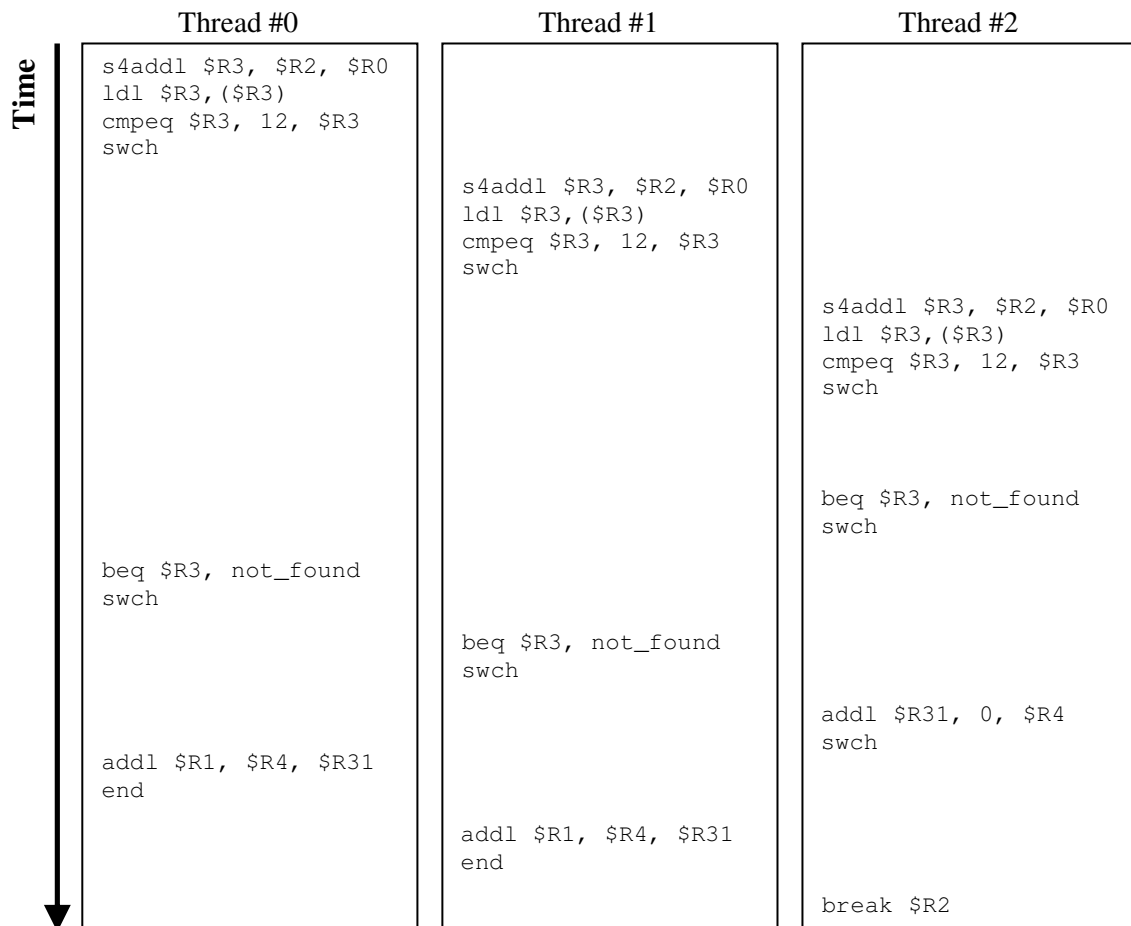


Figure 6 – Example of scheduling with constrained threads

Figure 6 shows this code being run and scheduled for a family of three threads on a single processor. The family searches the array with values 5, 7 and 12, respectively. Based on the code, the exit value of the family should be 2, the index in the array that contains the value 12. As can be seen, every *swch* causes the processor to switch execution to a different thread. At first, all three threads dispatch their memory requests and suspend on the results of those memory loads. For the sake of the example, the third memory request returns first, causing the third thread to wake up first. As can be seen, the third thread finds the value and wants to break, but cannot do so until it has read \$R4. The register is not written, however, until the second thread writes to \$R1. And since this requires the first thread writing to \$R1 as well, this forms a sequential ordering of the execution of the threads, ensuring that the third thread breaks only once the first and second threads have confirmed that they have not found the value by writing their shared registers.

The second mechanism to terminate is the KILL instruction. This instruction takes the Family Identifier of the family to kill as argument and is intended to be used to forcibly kill other families. A kill operates in much the same way as a break, in that it immediately and unconditionally halts execution of all threads of the family, causing the family to terminate. The exit-code register in the parent thread will receive a value different from when the break instruction is used to terminate a family so the parent thread is able to recognize whether the family terminated itself, and was therefore probably intended, or if it was terminated by a thread in another family. Killing a family does not cause a value to be written back to the exit-value register in the parent thread as is the case when breaking a family.

As with breaking a family, when multiple unconstrained, independent threads intend to kill the same family, no guarantees are made as to which kill is actually the cause of the family termination. The same applies for multiple concurrent kills and breaks. In such a case, only the parent thread will know whether a break or a kill is the cause of family termination, by examining the exit-code register in its register space.

In a distributed environment it is feasible that as a family of threads is running at a certain set of processors, another set of processors become available and the family could achieve higher performance by running on these new processors. In other words, one would want to *move* a running family of threads to different processors. The first step in this process is to suspend execution of, or *pre-empt* the family. For this purpose, the SQUEEZE instruction has been included in the micro-threading model. It takes the Family Identifier of the family that is to be pre-empted as an operand and upon completion will have pre-empted the family. Squeezing a family works by halting the creation of new threads in the family and allowing the existing threads to complete gracefully. The squeeze then captures the state of the family, consisting of the index³ at which thread creation was suspended and the value of the shareds of the last thread. This state is stored in the registers of the creating thread by the SQUEEZE instruction and can be used to restart, or resume, execution of the family on a different set of processors by skipping creation of the threads that have already been executed, and feeding the value of the shareds as saved in memory into the dependents of the first thread. Note that it is typically required to copy the squeezed family's state to conventional memory in order to make it available to the processor on which the execution will be resumed.

As a final note, both the kill and squeeze operations should be constrained in some way so as to prevent threads arbitrarily killing or squeezing unrelated families. The mechanisms which will efficiently and scalably provide this security are currently the topics of ongoing research.

³ This means that for distributed thread creation, the processors must agree upon a single, preferably the smallest possible, sequential thread index in the family for which no threads with an index greater than it yet exist on any processor.

4. The reference model

*“A designer knows he has achieved perfection not when there is nothing left to add,
but when there is nothing left to take away.”*

- Antoine de Saint-Exupéry

The previous chapter described the general micro-threading model. This chapter will focus on the specific design of this model as a group of in-order pipelined processors based on the Alpha instruction set. The processors are connected with each other via a ring network for neighbour-to-neighbour communication and a bus for broadcast messages. These networks are abstract and can be implemented in any way, as long as they behave as though they were a ring and bus network, respectively.

This model has been developed over the course of the development of the simulator, because the fact of implementing the model and testing the implementation revealed certain architectural design issues that needed to be fixed. Also, please note that during the writing of this thesis, research into the microthreaded model has continued and several architectural details have since been changed from how they have been described in this thesis.

4.1 The Family Table

The Family Table is an important data structure in the model, since it stores information about every family in existence. Its size is an important consideration in the viability of the model. Therefore, its fields have been listed and explained below. The fields have been grouped together based on their general purpose.

Thread Allocation:	
PC:	Initial program counter for new threads.
PhysicalBlockSize:	The block size as determined by available registers.
Start:	Start index for the family.
Step:	Step size for the family.
Index:	Index of the next to-be-allocated thread.
LastThread:	Index of the last thread in the family.
DoneAllocating:	A flag that's set when the last thread has been allocated.
NumAllocated:	Number of threads currently allocated.
LastAllocated:	TID of the last thread allocated to this family.
List administration:	
Next:	FID of the next family in a linked list families.
MembersHead / -Tail:	Two fields that define a linked list of all threads that belong to this family.
ActiveHead / -Tail:	Two fields that define a linked list of all threads in this family that are in the Active state.

Dependencies:	
CleanupDependencies:	A count of the number of outstanding dependencies that are required to complete after a family has been killed before the family can be cleaned up.
KillDependencies:	A count of the number of outstanding dependencies that are required, among other things, to complete before the family is considered terminated.
Killed:	If set, the family has been killed and pending operations should not write back.
Communication:	
VirtualBlockSize:	The block size as read from the TCB.
RegisterInformation:	Counts and address of this family's registers.
FirstThreadInBlock:	TID of the first thread in a block on the processor.
LastThreadInBlock:	TID of the last thread in a block on the processor.
Parent link:	
Parent TID / PID:	ID of the parent thread and ID of the processor where the parent thread resides.
ExitCodeReg:	Register address of the Exit Code register
ExitValueReg:	Register address of the Exit Value register

The "Thread Allocation" group contains the variables needed to allocate new threads and to know when thread allocation is finished. The PC is taken directly from the TCB and defines that starting point for new threads in this family. The PhysicalBlockSize is determined by taking the block size from the TCB (which is stored as the VirtualBlockSize) and reducing it if there are not enough free registers to allocate that many registers. Thus, the following relations hold: $1 \leq \text{PhysicalBlockSize} \leq \text{VirtualBlockSize}$. No more threads than PhysicalBlockSize will be allocated but their communication pattern will be determined by VirtualBlockSize. I.e., only if a thread is the last thread in a *virtual* block will a write to a shared register result in inter-processor communication. The start index and index step are also taken directly from the TCB. The current index is initialized at 0 when the family is created and incremented whenever a thread is allocated. The index is used to determine when the family is done creating (after the allocation when $\text{Index} = \text{LastThread}$), and it is used to determine the value for the first local register in a new thread: $\text{Start} + \text{Index} * \text{Step}$. LastThread is calculated from the End Index, Start Index and Index Step from the TCB as follows: $(\text{End} - \text{Start}) / \text{Step}$. LastThread is equivalent to the number of threads minus 1 and since the End Index is inclusive that leads to following equivalence (all divisions except the first one are integer divisions):

$$\begin{aligned}
\text{LastThread} &= \#Threads - 1 \\
&= \left\lceil \frac{\text{End} - \text{Start} + 1}{\text{Step}} \right\rceil - 1 \\
&= \frac{\text{End} - \text{Start} + 1 + \text{Step} - 1}{\text{Step}} - 1 \\
&= \frac{\text{End} - \text{Start} + 1 + \text{Step} - 1 - \text{Step}}{\text{Step}} \\
&= \frac{\text{End} - \text{Start}}{\text{Step}}
\end{aligned}$$

So where calculating $\#Threads - 1$ the naive way might lead to errors due to integer overflow, $(End - Start) / Step$ is guaranteed not to overflow. However, a separate flag is now necessary to indicate when the allocation has completed, because the processor cannot determine this from the value of the Index and LastThread in extreme cases. This flag is DoneAllocating and is set when a thread is allocated and Index was equal to LastThread. Note that in a hardware implementation, this may be replaced with a NumThreads field that is 65 bits to accommodate the overflow. The NumAllocated field holds the number of currently allocated threads in the Thread Table. It is used to check if PhysicalBlockSize has not yet been reached and to determine the base offset of a new thread in the register file. The LastAllocated field contains the TID of the last allocated thread. This information is used to properly map the dependents of the next allocated thread to the shareds of the last allocated thread.

The Next field in “List Administration” group is used to link the family in a linked list of families. The Members and Active linked lists, each consisting of a head and tail pointer, make up linked lists of all threads in the family and all threads in the family in the Active state, respectively.

The three “Dependency” fields are used to count pending asynchronous operations and prevent their completion should the family be killed. The KillDependencies is a field that, when it has reached zero, means that the family is considered terminated and any threads synchronization on the family may resume. The CleanupDependencies count the number of outstanding operations that need to complete after the family has terminated before the family can be deallocated and subsequently reused. The Killed field is set when the family has been killed or broken with a KILL or BREAK instruction, respectively. This field is checked whenever an asynchronous operation would complete and write back its results to a thread in the family. If the field is set, the result is discarded instead.

The fields in the “Communications” group deal with the inter-processor exchange of shareds. The VirtualBlockSize is the block size as taken from the TCB and allows every processor to calculate which threads it should run and which threads (the first and last in a virtual block) should communicate with the neighboring processor. The RegisterInformation field is made up of several components describing the base address of the registers for this family and how many globals, shareds and locals there are. This allows the family to calculate the proper offsets into the registerfile for register reads and writes. The FirstThreadInBlock and LastThreadInBlock fields contain the TIDs for the first thread and last thread in a block, respectively. These are used when receiving shared values or shared requests from neighboring processors. This process is described in detail in section 4.7.

Finally, the three fields in the “Parent link” group contain the information that allows the family to communicate the results of its threads back to the parent. The Parent TID / PID field identifies the parent thread and processor on which this thread is running. The ExitCodeReg and ExitValueReg fields are only valid on the processor which contains the parent thread and contains the addresses into register file where the exit code and exit value should be written, respectively. These addresses are in the register space of the parent thread.

4.2 The Thread Table

Just like the family table, the thread table is another important data structure in the model since it contains the state of every thread in existence on the processor. The thread table consists of the following fields for each thread. The fields have been grouped together based on their purpose.

General:	
PC:	If a thread is not running, this field will hold the current program counter of the thread.
FID:	This is the Family Identifier of the family that this thread belongs to
RegisterBase:	This field holds the base address in the register file for this thread.
DependentsBase:	This field holds the base address in the register file for the dependents of the thread.
List administration:	
NextState:	This field holds the TID of the next thread in the linked list of threads that are in the same state in this thread's family.
NextMember:	This field holds the TID of the next thread in this thread's family. It also used to link to the next empty thread, if this thread is empty.
Dependencies:	
IsKilled:	A flag that notes whether this thread has been killed or not.
IsNextKilled:	A flag that notes whether the next thread in the family has been killed or not.
IsPrevReleased:	A flag that notes whether the previous thread in the family has been cleaned up or not.
Communication:	
IsFirstThreadInFamily:	A flag that notes whether this thread is the first thread in the family.
IsLastThreadInFamily:	A flag that notes whether this thread is the last thread in the family.
IsLastThreadInBlock:	A flag that notes whether this thread is the last thread in the block.
PrevThreadInBlock	A field that holds the TID of the previous thread in the block.
NextThreadInBlock	A field that holds the TID of the next thread in the block.

The two fields in List Administration provide links to the next threads in the, at most two, linked lists a thread can be linked in. These lists will be described in section 3.3.

The three dependency flags are all required to be set to one before this thread can be released. The IsNextKilled flag makes sure that a thread isn't released until its successor in the family has been killed. This is because that successor could still be using this thread's shares as its dependents. The IsPrevReleased flag ensures that threads are cleaned up in the same order they are created. This simple ordering avoids complicated logic that would have to clear registers while taking into account that a thread's predecessor might not yet be cleaned up. The IsKilled flag is simply to mark when the thread has been killed such that, when any of the other flags are set, the thread can be released.

In the case of a family of independent threads, there are no shares and dependents that need to be taken into consideration when releasing threads, so the IsNextKilled and IsPrevReleased

flags are set the moment the thread is created. This effectively ensures that a thread is released the moment it is killed while the logic that checks these flags can remain virtually unaltered.

The five communication fields enable the pipeline to determine where a thread's predecessor and/or successor threads exist; either on the current, previous or next processor. This information is used when writing shareds, to know whether the value should be written to the local register file or be sent to the next processor as well and when killing and releasing threads, to update that thread's previous and/or next thread in the family.

4.3 Thread States

A thread can be in one of six logical states: empty, waiting, ready, running, suspended and unused.

Empty threads are threads not yet allocated to a family. This is the initial state for threads. Once a family has been allocated in the family table, threads may become allocated to a family. Once a family is cleaned up or killed, all threads belonging to that family are reset to the empty state.

Waiting threads are threads that are waiting for their instruction data. This state is entered once a thread is normally ready to be run, but the PC address misses the I-Cache, causing the cache line containing the thread's instructions to be asynchronously fetched from memory. Once the cache-line returns, the thread is advanced to the ready state.

Once a thread enters the *ready* state, they are available to be run by the pipeline. This means that the thread is able to execute the instruction at the PC and has no known dependencies and its instruction stream has been cached.

Running threads are threads that were previously in the ready state, and have been selected to be run by the pipeline. They will continue in this state until a thread switch occurs by either the thread executing an instruction causing one, or when the program counter crosses a cache-line boundary and the availability of the next instruction can not be guaranteed. Depending on what causes the thread switch, the thread advances to the suspended state, the waiting or ready state, or is killed.

Once a thread enters the *suspended* state, it has attempted to read an empty register and has to wait until said register is written to. The register it attempted to read will contain its Thread Identifier, allowing the processor to wake up the thread once the register is written. After the thread is woken up, the thread enters the waiting state.

If a thread is killed, the processor first determines whether the family that the thread is part of still has threads left to allocate. If this is the case, the thread is immediately reused by 'overwriting' it with the next thread in the family. This has the benefit of not needing to update several fields in the thread table.

If, however, the family that the thread is part of has already allocated all of its threads, the thread enters the *unused* state. This state is much like the empty state in that the thread is no longer used by the processor. However, unlike the empty state, the thread cannot be allocated to other families. The unused thread is, in effect, a 'lost' thread, until the family that it is part

of terminates. Once that happens, all threads in the family are simultaneously changed to the empty state.

These states and their transitions are clarified in Figure 7.

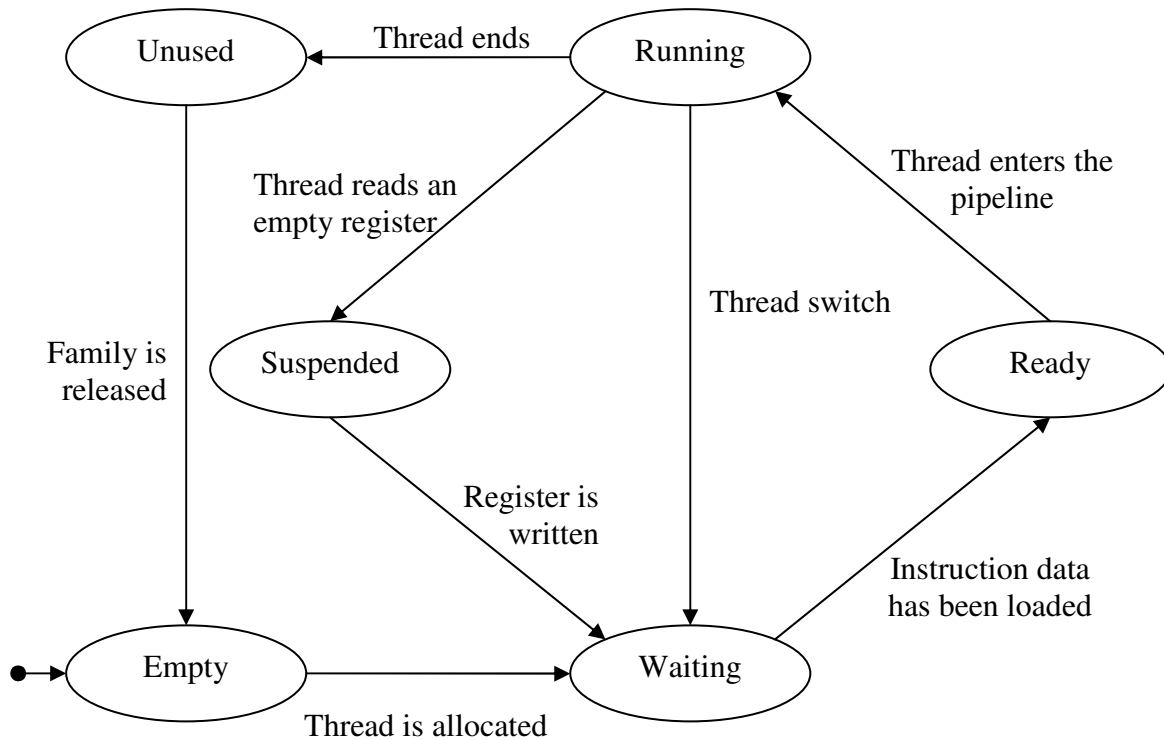


Figure 7 - Thread states and their transitions

4.4 Thread Lists

Because microthreaded model aims to eliminate high-latency actions blocking the processor, threads have to be put ‘aside’ should they encounter a long-latency delay. To allow threads to be efficiently suspended and restored, several linked lists of threads are used. Linked lists have the benefit of having a single-cycle access and update time and scale very well with the number of threads. A linked list of threads is referred to with a simple head/tail pointer. A head and tail combination is used instead of just a head pointer because in the latter case, both adding and removing threads always requires access to the head pointer. With a separate head and tail pointer, threads are added to the back and removed from the front. As long as the length of the list is greater than one, these operations are completely disjoint. In addition, the threads are now added and removed in a FIFO order, aiding fairness in processing threads.

All threads in the thread table are, at any time, linked in one or two per-family linked lists. These lists are a *family state list* and the *family member list*, and each list has a “next” field in each slot in the Thread Table, called the “NextState” and “NextMember” fields, respectively.

Threads are linked in a family state list when they are in the ready or waiting state. Threads in the empty state are linked in a processor-global empty list with the “NextMember” field and threads in the running, unused and suspended states are not linked in a state list at all. The

reason these lists are per-family is so that when a family is killed or broken, all threads on that family can be immediately released without breaking some processor-global linked list.

The family state list is used to link threads of a family that are in the same state together.

Threads are put on the family member list once they are allocated to the family for the first time and continue to be on the list for as long as the family exists. The family state list exists to provide an efficient way to clean up *all* threads belonging to a family at once when the family is killed or broken. This is done by simply appending the family’s member list to the single, processor-wide empty thread list in a single cycle. An example of the usage of linked lists is given in Figure 8.

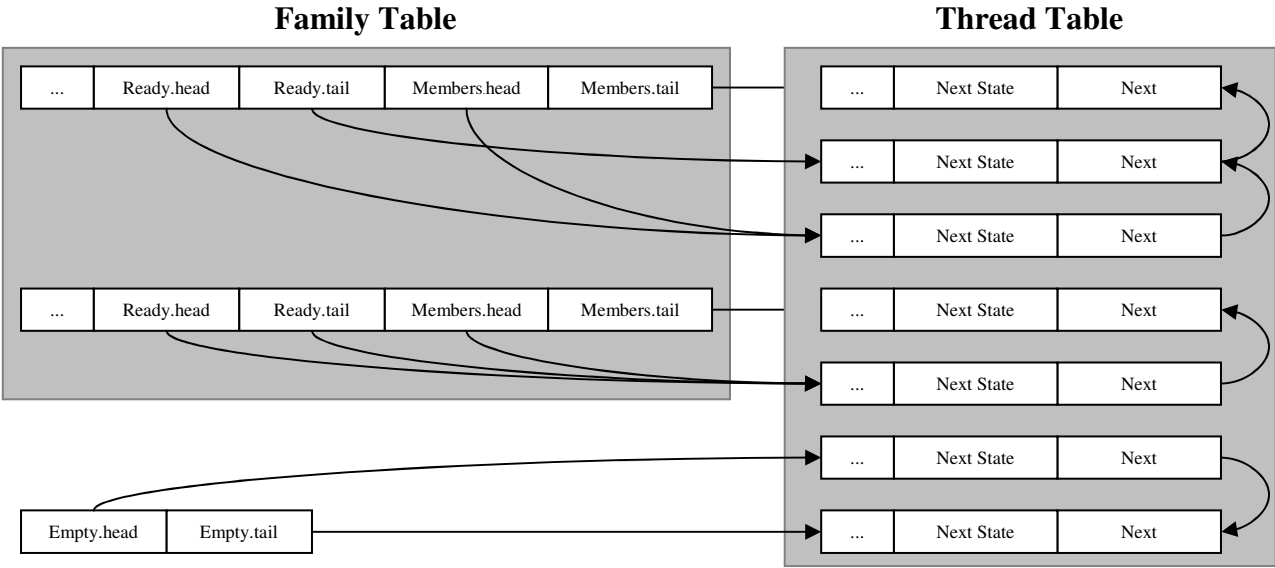


Figure 8 - Example of thread lists

4.5 Family and Thread Allocation

From any processor’s point of view, there are two sources of creates: the pipeline and the ring network linking a group of microthreaded processors. A family create originates from the pipeline when the CREATE instruction passes through the pipeline on the processor. A family create arrives from the network when another processor has created a group-wide family and has broadcast the create information to the other processors in the group. Note that the latter type of create can contain significantly more data, such as all the globals of the family.

These creates are queued in two queues; creates from the pipeline are queued in the Local Create Queue (LCQ), and creates from the network are queued in the Remote Create Queue (RCQ). As soon as either queue contains a create, one is picked and selected as “Current Create”. To ensure existing families can complete as soon as possible, creates from the RCQ are favored over creates from the LCQ.

Once a “Current Create” has been selected, a memory request is sent for its TCB, whose layout is shown in Figure 10. The TCB contains the limits, step size, initial address and register counts for the threads in the family as well as some flags and the block size. The only

flag currently supported is the LOCAL flag, which means that the family should only be created on the processors executing the create, regardless of how many threads it has.

Once the TCB has been fetched into a temporary buffer, the Family Table and Register Allocation Unit (RAU) are checked to see if a family can be allocated. The RAU will allow a family to be created if it has at least enough space for a single thread of that family (as indicated by the register counts in the TCB).

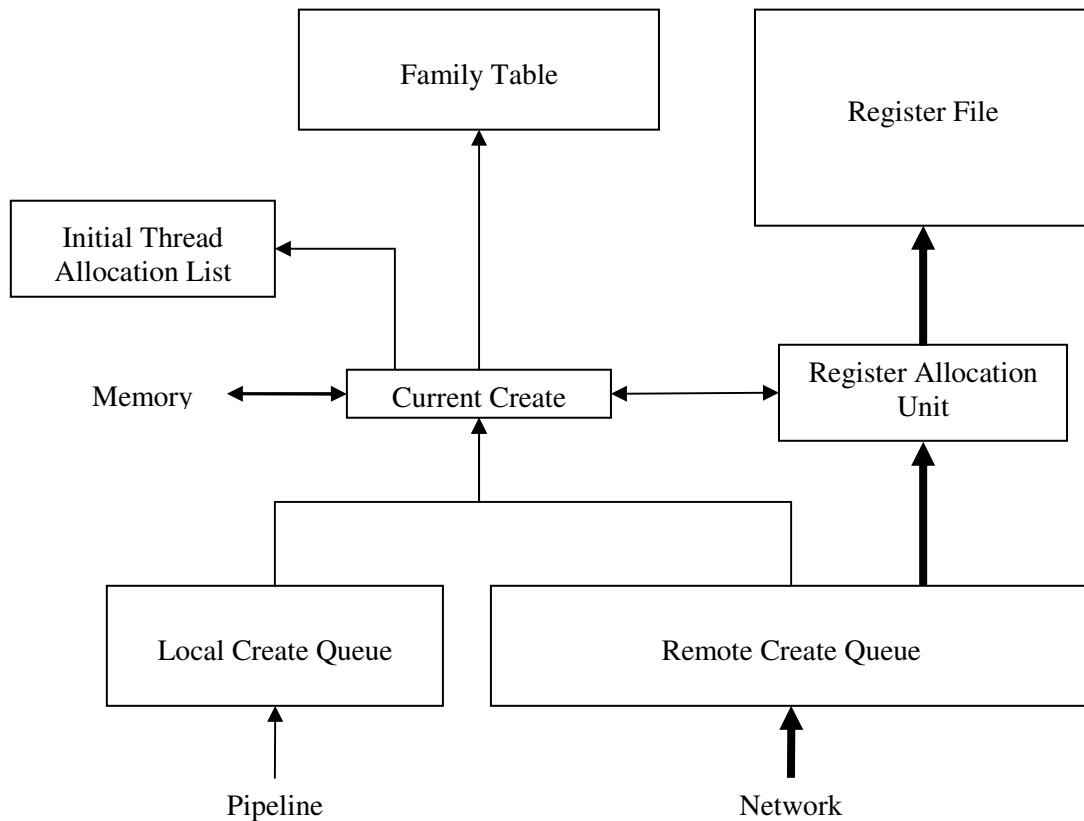


Figure 9 - Family creation overview

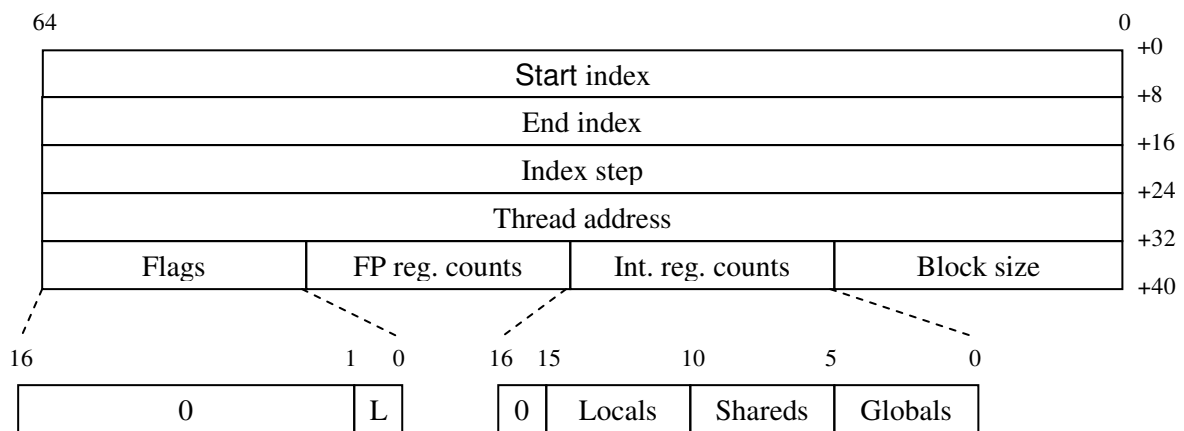


Figure 10 - Thread Control Block layout

With registers allocated and an unused family slot in the Family Table selected, the family table slot is initialized with the base address of the allocated registers and the information from the TCB and Current Create. If the create originated from the RCQ, the global registers contained in the create are also written to the newly allocated region in the register file. Once all of this is completed, the family is considered created and put on the back of the “Initial Thread Allocation” linked list. If the LCQ or RCQ contain additional entries, the family allocation process starts anew.

Based on the availability of registers as determined during family creation, and the information in the TCB, a maximum number of threads that are to be allocated to this family has been calculated. The “Initial Thread Allocation” linked list is a linked list of families that have been allocated but do not yet have this many threads allocated.

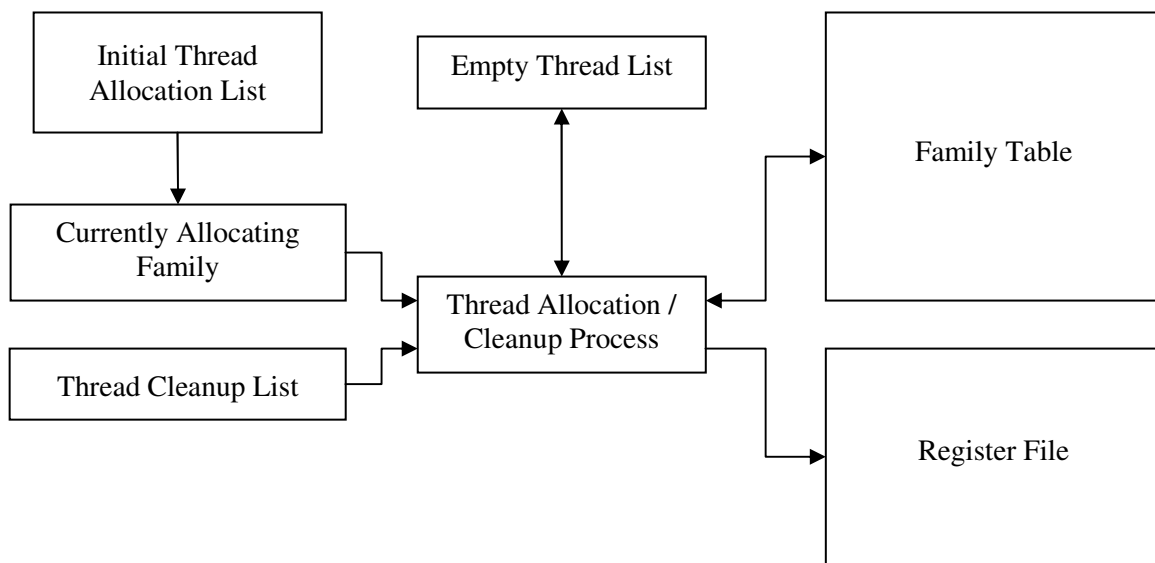


Figure 11 - Thread allocation overview

Once the “Initial Thread Allocation” list contains a family, the first one is chosen as “Currently Allocating” family and removed from the head of the list. The processor will try to keep allocating threads to the “Currently Allocating” family until that family has reached its maximum number of threads or until the family has terminated, whichever occurs sooner. Once this has occurred, the next family in the “Initial Thread Allocation” list is chosen as “Currently Allocating” family.

Threads allocated to the “Currently Allocating” family can come from two sources: the Empty Thread list or the Thread Cleanup list, the former only being used if the latter is empty. The Thread Cleanup list is a linked list of threads that have completed and are available to be cleaned up. However, to avoid the unnecessary moving around of threads, the Thread Allocation / Cleanup Process first checks if the first thread on the cleanup list can be reallocated to the same family. If so, this means that the thread can keep the same registers and family information. If the family is done allocating, the process checks whether there is a Current Allocating family. If so, the thread is allocated to that family. If not, the thread is pushed on the Empty Thread list. If there are no threads on the Thread Cleanup list, but there is a Currently Allocating family, the process will attempt to remove a thread from the Empty Thread list. If this list is empty, the process stalls. The Thread Allocation / Cleanup process’s pseudocode is shown below:

```

Repeat
  If the Thread Cleanup List contains a thread Then
    Read TID and FID from the head of Thread Cleanup List
    If the family is not done allocating threads Then
      Reallocate thread to its family
    Else If Currently Allocating Family register is not empty Then
      Allocate thread to the Currently Allocating Family
    End If
  Else If Currently Allocating Family register is not empty Then
    If Empty Thread List contains a thread Then
      Remove TID from the head of the Empty Thread List
      Allocate the thread to the Currently Allocating Family
    End If
  End If
Until Forever

```

Note that the pseudocode also illustrates that reallocation of cleaned up threads takes priority of allocating threads to new families. This decision was made as part of the general strategy to ensure that advancing already allocated families always takes priority over allocating new families.

Allocating a thread involves choosing its place, its index, in the family. For families with independent threads the exact allocation model doesn't matter as long as the threads are evenly distributed. However, for families with inter-thread communication the allocation model could impact performance if chosen poorly. A block cyclic allocation model was chosen for the reference model. This model allocates blocks of threads on a single processor, and adjacent blocks on adjacent processors. Communication between threads in a block thus remain local to the processor and fast and having the blocks on adjacent processors ensure that inter-processor communication, when it does occur, only occurs between neighboring processors. The block size is specified by the program in the TCB.

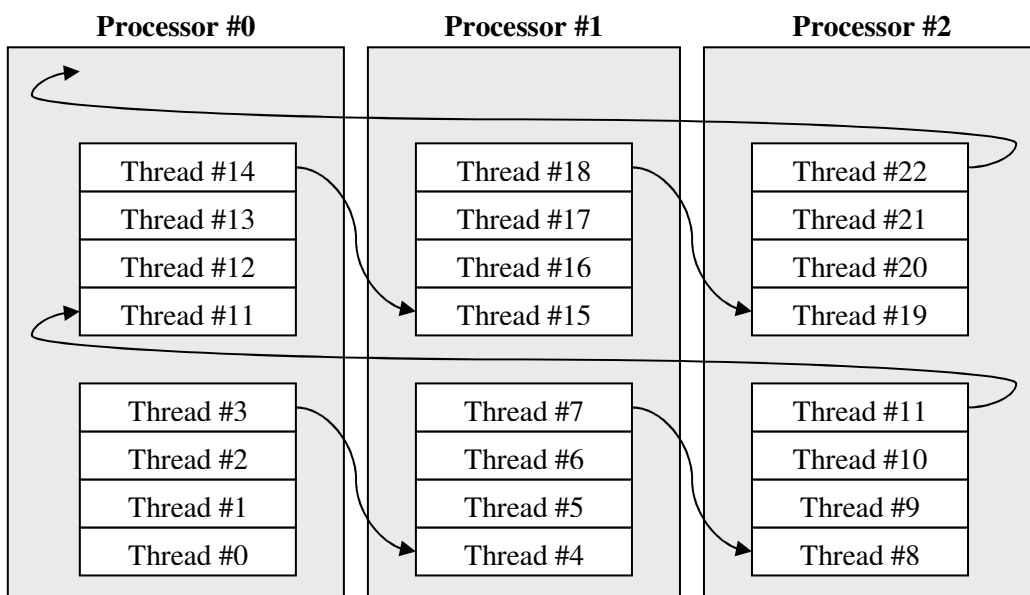


Figure 12 – Block-cyclic thread allocation model

By definition, the first thread in a family is always on the processor that created the family. The other threads in the family are allocated from there on. In Figure 12 this is processor #0, but in practice it can be any processor. In general, if there are N processors and the family with virtual block size B was created on processor P (with $0 \leq P < N$), thread i will be allocated on processor P_i , defined as follows:

$$P_i = \left(P + \left\lfloor \frac{i}{B} \right\rfloor \right) \bmod N$$

Although this predefined mapping would allow a processor to completely allocate all threads mapped to it, the model does not allow more than B threads to be allocated to a family at any given time. This was done to minimize the risk of resource deadlock due to a single family consuming all available entries in the Thread Table. Note that B is a maximum upper bound on the number of allocated thread. The actual limit may be less depending on available register space. This limit is stored as the physical block size in the Family Table.

4.6 Memory and caches

The aim of the microthread processor model is to reduce the impact of high-latency operations, which is done by suspending threads using the results of such operations. The model also allows for multiple long-latency operations, including memory reads, to be active at any time. Scalability to many processors is another important aspect of the microthread model. The memory model must be able to handle both these aspects efficiently. And although actual memory models were not part of the scope of this project, the memory *interface* was.

In the reference model, each processor has a single interface to memory as well a L1 instruction cache and an L1 data cache using that interface. Since multiple memory requests, both for instructions and data can be outstanding, the memory protocol needs to identify each individual memory request. It does this by having the processor attach a tag to every request. To the memory system, these tags can be considered opaque identifiers, having meaning only to the processor where the request originated. Therefore, the memory system must pass the tags through the memory system unaltered.

In the reference implementation, a tag is made up of three parts: a “type” bit, a Family Identifier field and a Cache-line Identifier (CID) field. The type bit indicates whether the request was meant for the Instruction or Data cache. The FID field identifies the family the request is meant for. In the case of a TCB request (which is considered part of the Instruction Cache, although it isn’t cached), this field is set to invalid. The CID field identifies the cache-line that the retrieved data should be written to. If there is no data to write, this field is set to invalid. The FID and CID fields are set to invalid by either setting a flag, or by setting the field to an invalid value.

Since there is a unique combination of values and validities of these fields bound to each type of memory request, a processor can deduce which memory request a response corresponds to. Table 1 lists the different kinds of memory responses and their combination of the fields in the tag. The “Data write confirmation” response exists solely to allow families to properly synchronize. Since family synchronization involves all memory writes made by that family to

be visible to other processors, a processor needs to know when this is the case. In the reference implementation, the memory returns a confirmation for every write once that write is visible to all processors. This response carries no additional data other than the tag to identify the family whose synchronization status can be re-evaluated.

Response type	Type field	FID field	CID field
TCB data	Instruction	Invalid	Invalid
Instruction cache-line read	Instruction	Valid	Valid
Data cache-line read	Data	Valid	Valid
Data write confirmation	Data	Valid	Invalid

Table 1 - Tag fields for memory responses

Each memory request, except those for TCBs, is tagged with the Family Identifier of the family whose thread caused the request. This is done to allow families to be killed. When a family is killed, its threads and other family-specific data structure outside the family table are immediately freed. However, there could still be outstanding asynchronous operation pending for threads of the family. If these operations were not tagged with the FID, the processor would overwrite invalid threads or other entries when they complete. Tagging the operations with the FID and checking if the family has been killed before writing back results prevents this. As a result of this mechanism, the family table entry needs to remain allocated until all outstanding asynchronous operations have completed. To know when this has happened, each family has a counter which is incremented whenever such an operation is issued and decremented when it completes. When this counter reaches zero, the family can be deallocated.

The Instruction cache is a normal instruction cache except that its cache-lines are not addressed with just the memory address, but with the memory address and the FID of the family whose thread is making the request. This is done to allow the processor to easily clear all cache-lines belonging to a family when that family is killed. Each cache-line also has a TID and TIBID field. The TID field acts as the head pointer in a linked list of threads in the Waiting state which are suspended on that cache-line. When the cache-line is fetched from memory, this list of threads can be appended to the list of Ready threads in a single cycle. The TIBID field is an index into the Thread Instruction Buffer that holds the same data as the cache-line.

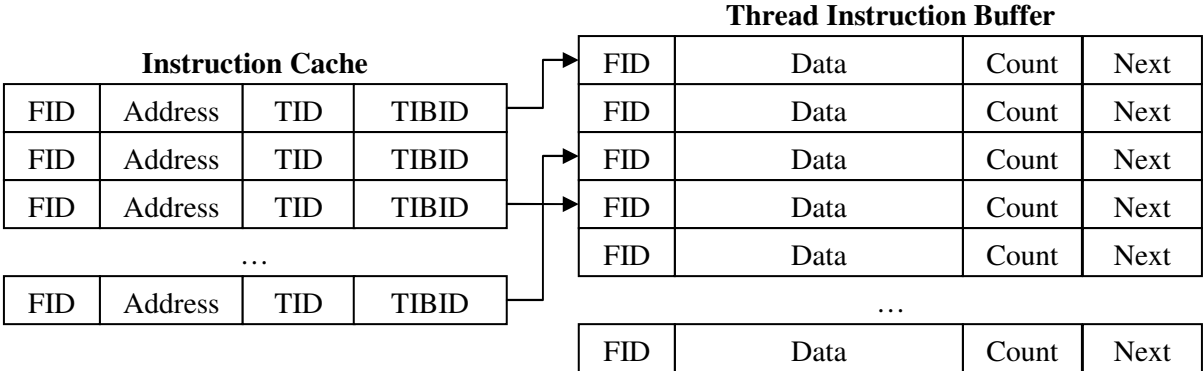


Figure 13 - Instruction Cache and Thread Instruction Buffer interaction

The data cache is a normal data cache except for some multi-threaded adjustments. Namely, the data cache also contains an associative buffer that stores every pending register read request that has missed the cache. After the data has been fetched from memory and stored in a cache-line, the cache-line is added to a linked list of cache-lined that require additional processing, the so-called “Completion List”. The head of this list is used to find and complete all requests in the buffer that can now be completed with the retrieved data. Once no more requests for the head of the list can be found, the cache-line is removed from the list. Note that the cache-line was and stays valid to serve new requests since the data returned form memory. The FID is stored with every request in the Data cache as well to allow the processor to check if the family has been killed since the request was sent.

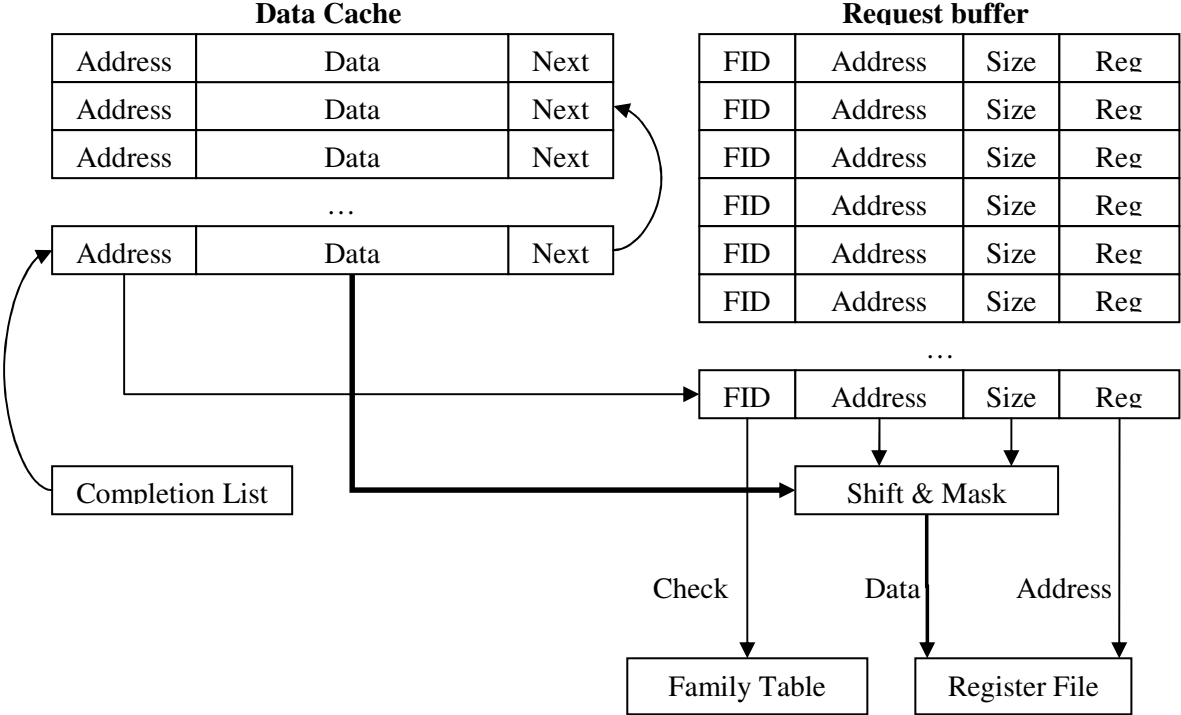


Figure 14 - Data Cache structures and completion operation

4.7 Inter-processor shareds exchange

When a thread that’s first in a block attempts to read a dependent, the read value originates from the shared register of the previous thread, which is located on the neighboring processor. However, due to the thread distribution and the fact that each processor starts allocating and executing threads independently, this thread may not yet exist. The shareds exchange protocol ensures that the thread ends up with the dependents of its predecessor regardless of allocation order or timing.

Since there is no way to know the location of the dependent register in the register file in advance, the request cannot be stored in the dependent register in the same way that a thread suspends on a register. Similarly, if the situation is reversed and the thread that produces the shared value does so before the thread that reads the shared value even exists, this value cannot be stored in a register in advance.

The protocol implemented in the reference model always sends a shared when it is written, always requests a shared when the register is read but empty, and discards received requests or values if they cannot be handled or written. And to prevent a thread being released before its successor can read its dependents, a thread can only be released after its successor has terminated. Note that this, in effect, enforces an ordering of the cleanup of threads in a family with dependencies between threads. This relatively simple protocol ensures that a thread can always transparently read the dependents of its predecessor, without being concerned with whether that thread already exists or not.

To illustrate this protocol, four possible situations regarding thread existence and timing of reads and writes will now be discussed.

In the first situation, which is probably the most common given the thread allocation strategy, the first thread in a block on processor B reads a dependent, which is empty, and thus requires the value of the corresponding shared register from the last thread in the block on the previous processor A. Processor B therefore sends a message to processor A indicating that it wants the value of the shared and suspends on the empty register. Processor B, on receiving the message, determines that the thread that produces the value does not yet exist and discards the request. When the thread has been created and writes its shared register, a message is sent to processor A containing the value of the shared register. Processor A checks that the receiving thread exists, writes the value to the correct register and by doing so, wakes up the thread that suspended on it.

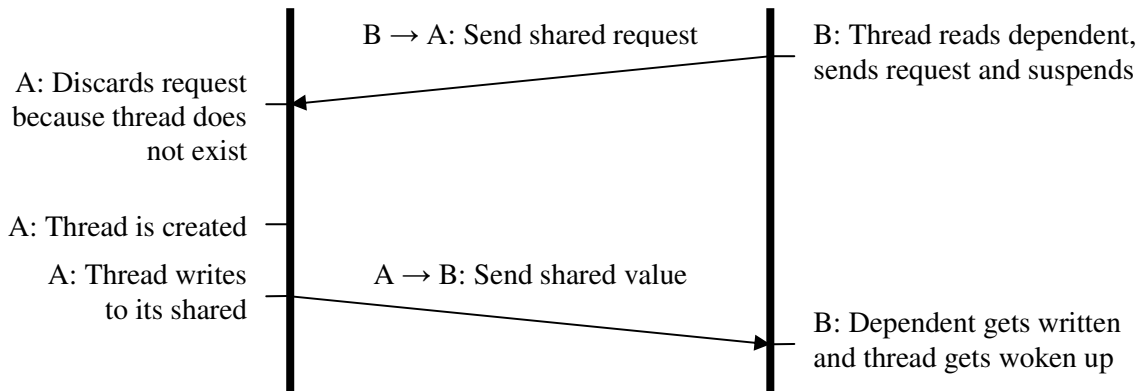
The second situation, while technically distinct from the previous situation, is similar to it in principle. Instead of the thread on A not existing, the thread on A exists, but hasn't written its shared register yet. So, again, processor A discards the requests and when the thread writes its shared, it sends the value to processor B, where it is written to the dependent register and thus wakes up the suspended thread.

The third situation has the thread on processor A writing its dependents before the thread on processor B that will read the shared even exists. By writing the dependent, processor A will send its value to processor B. Processor B will check and see that the receiving thread does not yet exist and discard the request. Later, the thread will be created on processor B and reads its shared. We then basically enter the previous scenario, where processor B sends a request to processor A, processor A sends the value back, and processor B writes the value and wakes up the thread.

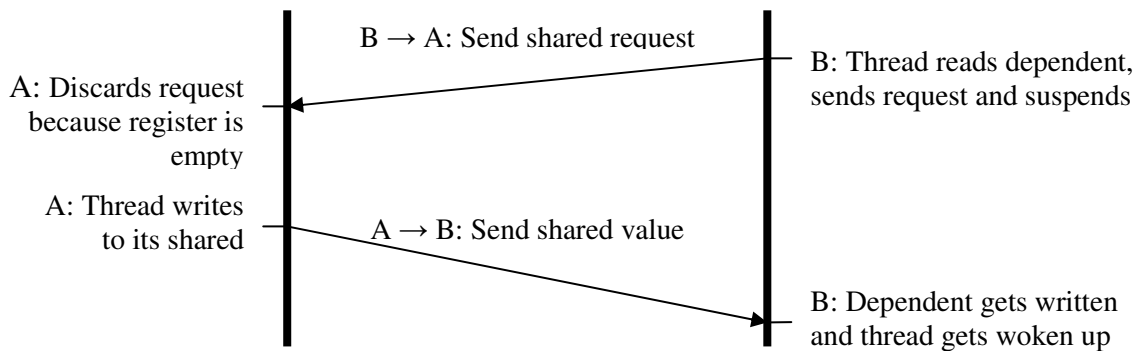
In the fourth situation the thread on processor A writes to its dependent before the thread on processor B reads its shared. However, the thread on processor B does already exist. So this time when the value is received by processor B, it immediately writes it to the shared register of the thread. When the thread now reads its shared, it's already full and the thread can immediately continue.

All of these situations are summarized in Figure 15.

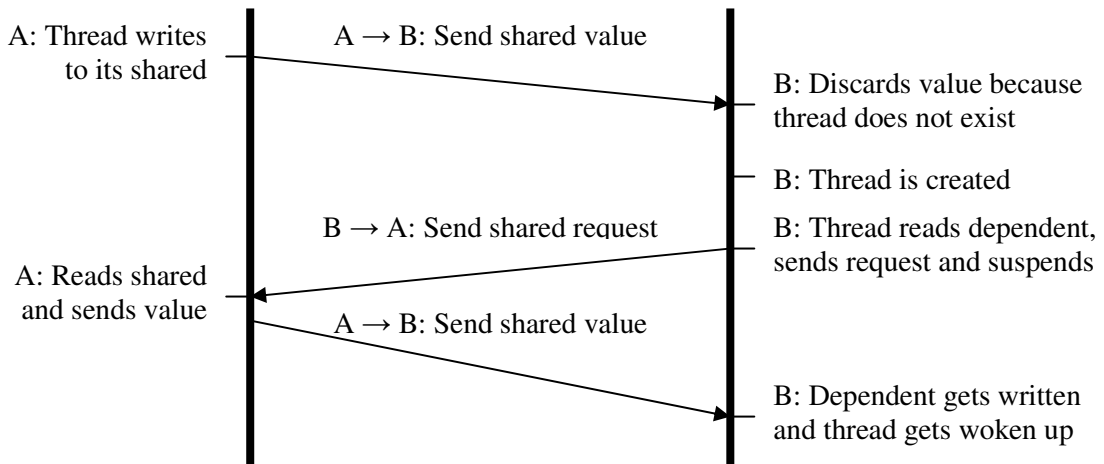
Situation 1: thread on B wants shared from A, where thread does not yet exist



Situation 2: thread on B wants shared from A, where shared hasn't been written yet



Situation 3: thread on A writes shared before thread on B exists



Situation 4: thread on A writes shared before thread on B reads dependent

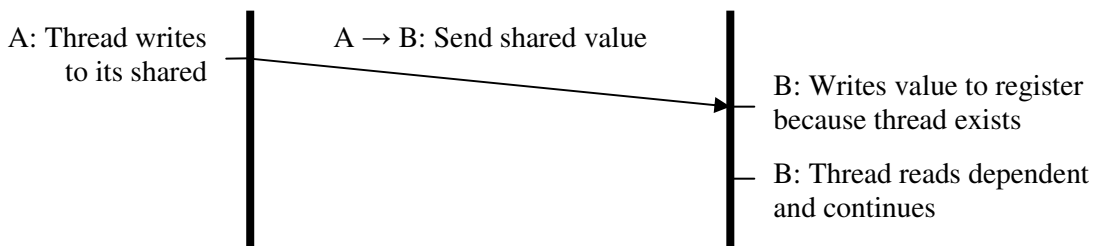


Figure 15 – Four situations in the shareds exchange protocol

4.8 Family Termination, Synchronization and Cleanup

Once a family has been created on a set of processors, each instance of the family will each run independently of the others, although slightly constrained when the threads use shared registers. Since synchronization on the family by its parent thread has to guarantee that the family has terminated on all processors, a scalable mechanism has been implemented to notify the processor containing the parent thread when this event had occurred. The family instance on the processor immediately neighboring the processor containing the parent thread starts with possession of a “*synchronization token*” when it is created. Once a family instance has completed locally *and* is in possession of the synchronization token, it will pass the synchronization token on to the family instance on the next processor. This way, the synchronization token will eventually make its way around the ring and end up in possession of the family instance on the processor containing the parent thread. When this family instance finishes there as well, it is guaranteed that all instances of the family on the processors have finished and thus the parent thread can be woken up by writing the Exit Code register.

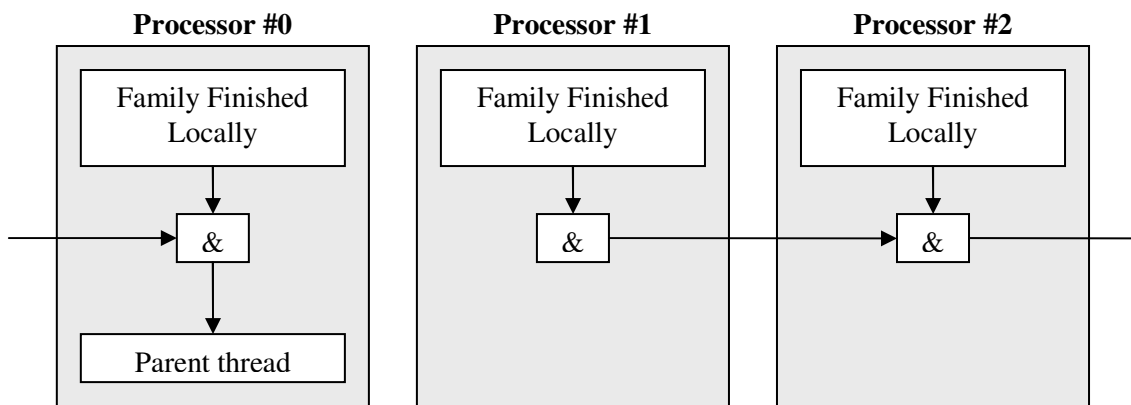


Figure 16 - Family termination mechanism

Once a family has terminated and passed on its synchronization, it can become eligible for cleanup. Since cleanup can also occur after a kill or break, the processor has to ensure that no outstanding references to the family exist before releasing its entry in the family table. Otherwise, another family may be allocated to the entry, and any outstanding reference may influence the wrong family.

To this end, each family has a dependency counter. Whenever an outstanding reference to the family is created, this counter is increased and whenever an outstanding reference to the family is destroyed, this counter is decreased. Only when the family has terminated and this counter has reached zero, can the family entry be released and subsequently reused. Since the requirement of events of family termination is a subset of the requirement of events of family cleanup, this dependency mechanism is extended to include family termination. In this manner, when the dependencies that are required for family termination have reached zero, the family is terminated and once the dependencies that are required for cleanup (which includes the required events for termination) have reached zero, the family is cleaned up. Table 2 lists the complete list of dependencies, with their influence on family termination and cleanup. Note that this method of counting dependencies may not necessarily be implemented identically in hardware, but should at least be implemented to produce the same behavior.

Dependency	Is a requirement for	
	Termination	Cleanup
Thread allocation completed	X	X
Synchronization token received	X	X
Thread count	X	X
Running threads	X	X
Outstanding shares	X	X
Outstanding memory writes	X	X
Outstanding memory reads		X

Table 2 - Family dependencies

First and foremost, all threads must have been allocated before a family can be considered termination. Secondly, a family must have received the synchronization token from the previous processor (this can be ignored if this processor is the first in the ring, as shown in Figure 16). Thirdly, all threads that have been allocated must have been released. Fourthly, since a thread's termination status is determined before the end of the pipeline, all threads must actually completely clear out of the pipeline before a family is terminated. Fifthly, all shares must have been sent to the next processor or written to the parent thread. If the shares are sent to the next processor, it must confirm this by sending an acknowledgement. Upon receiving the acknowledgement, the processor will adjust the dependency counter. This mechanism is used to prevent shares being stalled in whatever network connects two processors and the synchronization token thus overtaking the shares, possibly causing the family on the next processor to terminate before the shares have arrived. A final dependency for family termination is the number of outstanding writes. As described in section 3.4, all writes must be guaranteed to have completed before a family can be synchronized upon. By delaying family termination until all memory writes have been acknowledged by the memory system, this can be guaranteed. Once a family has terminated, a single dependency still needs to be resolved before the family can be cleaned up. All memory reads must have completed. Although in a normal program all threads will wait for the data they requested, when a family is killed, there may still be outstanding memory reads and writes. The family entry must persist until all those requests have returned.

5. The simulator

“Who cares how it works, just as long as it gives the right answer”

- Jeff Scholnik

The model described in the previous chapter has been implemented in a command-line simulator written in C++. The simulator started out as a modification of an existing simulator that implemented an earlier version of the model, but later on it turned out that a rewrite from scratch would be easier than modifying the existing simulator to implement the new model, which was significantly different from the old model.

Unfortunately, due to time constraints, the Kill, Squeeze and Break functionality have not been implemented, but their foundations have. The reference model has been developed with these three operations in mind, and the implementation mirrors this, so implementing them can be done without major architectural restructurings.

5.1 General structure

The simulator consists of three parts: a collection of classes called components and structures that define the implemented processor, a framework that binds all classes together and provides a generic simulation framework and a command-line front-end. Figure 17 shows a schematic overview of the simulator architecture.

The framework consists of the base code for each component and structure and various utility classes that allow the components and structures to interact. The main class in the framework is the *Kernel* class. A kernel is responsible for simulating a set of components in a thread. By creating multiple kernels and distributing components over them, the simulation can become multi-threaded. The components can be connected in a hierarchy, allowing for easy instantiating of multiple components. Each component has a unique name and combined with its parent's names, has a globally unique name.

The simulation operates by the kernels calling their component's cycle functions in lock-step; every component is advanced one cycle before the next cycle commences. This method of execution, opposed to event-based simulation, ensures components can never run ahead of other components and negates the necessity of complex rollback operations. Each component's cycle function can now be written in a straight-forward manner, reflecting its behavior in hardware as much as possible.

The front-end can be any type of interface including command-line, graphical and network. The front-end is responsible for constructing the system and instructing the kernels to advance the components. In the implemented simulator a command-line front-end has been implemented that allows the user to step the simulation or run it to completion and print the component's states for inspection.

The components and structures make up the bulk of the simulated hardware model. Components are objects in hardware that have cycle functions that are run every cycle. They

can be thought of as the objects that actually perform the processing. Structures, on the other hand, are just objects that hold data. The family table, thread table and register file are such structures. The simulator only allows ports (see section 5.3) to be defined on structures. Note that every simulation model-specific component and structure inherits from a base component and structure class that interfaces the component or structure to the framework.

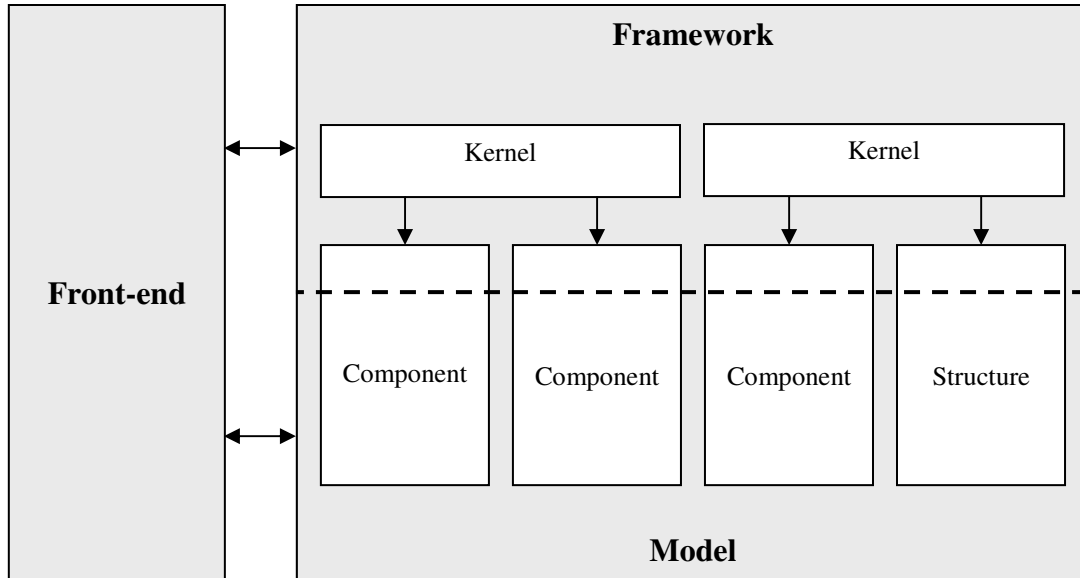


Figure 17 - Simulator architecture

5.2 Cycle phases

An early problem that arose during development of the simulator was the issue of cycles and the scheduling of cycle functions. In many cases, a write to a data structure must not be visible until the next cycle, as this is also the case in hardware. However, if component A writes to the register before component B's cycle function has been called, component B will then read the new value of the register later in the same cycle. If a series of components are scheduled as such, an event can propagate through multiple structures like this in a single cycle, whereas it would normally take several cycles.

To prevent this, every cycle has been split up into a read phase and a write phase. Every component's cycle function was also split into two functions accordingly. First, the cycle read functions were called for all components, and then all cycle write functions. The cycle read functions would read the data structure and store the value in the component class, where it would be used by the write function. This separation effectively simulated the cycle delay in reading and writing data structures.

In order to support ports, which require a separate *acquire* phase and atomicity (see section 5.4), which requires a *verify* phase, the read and write phases are split up in three sub-phases each. The resulting six phases are *read acquire*, *read verify*, *read commit*, *write acquire*, *write verify* and *write commit*.

In the acquire phases, each component's cycle function is called to allow it to acquire any ports it might need. In the verify phases, the functions verify if they actually acquired the port and if the operation can continue. Finally, in the commit phase, the operation is performed.

5.3 Ports and functions

In order to accurately simulate contention on data structures, each data structure has one or more ports, modeled after their hardware equivalents. A port is a class that can either represent a read port or a write port. During the run-time construction of the various components, they are ‘connected’ to various ports which does nothing more than authorizing the component to use them and assign a relative priority value to the connected components for arbitration. If a component wishes to read from or write to a data structure, it must first ‘acquire’ the proper port in the acquire phase. Acquiring a port means that the component’s address is stored in a list in the port. After the acquire phase, a port has a list of all components that wish to use the data structure. The port will then choose the component from the list that has the highest priority. In the next cycle phase, each component will check if it has been chosen, and if so, continue with its operation.

In the case of write ports, this mechanism is slightly more complicated, since multiple writes from different ports to the same location in the data structure cannot occur. Therefore, components must also specify the address they want to write to when acquiring the port. Then, after each write port has chosen its component with the highest priority, the data structure determines which ports can continue with the write operation by comparing the port priorities and write addresses of the selected components.

Functions are similar in operation to read ports, but are used to arbitrate an interface to another component, instead of a data structure. Such an interface could be the I-Cache’s “fetch cache-line” function. This can only be called by one function per cycle, so every component that wishes to fetch a cache-line first has to acquire the function ‘port’.

Ultimately, these port and function classes can be extended to keep track of the contention on them. This can provide valuable data in determining where to add ports if performance was found to be lacking because of contention.

5.4 Atomicity

Many component cycle functions need to perform multiple operations. These operations typically happen in parallel in hardware or function in a way that they are “all or none”. For example, creating a thread involves putting the thread on the family’s membership queue, fetching the thread’s cache-line, updating the family table entry and initializing its virtual register window. If either of those operations cannot succeed, none of them should be done. Essentially, each component’s cycle function must execute atomically.

An early-used solution was to split the read and write cycle phases into three sub-phases for a total of six cycle phases. The read and write phase would each have an *acquire*, *verify* and *commit* phase. In the acquire phase, the cycle function would attempt to acquire any ports it would need. In the verify phase it would verify that it held the port and that the operation actually could succeed. Only then would the operation be performed in the commit phase. These distinct operations were implemented in the cycle functions by having the framework pass the current phase an argument to the cycle function and then constructing multiple if-then-else constructs in the cycle function to act accordingly. However, after implementing a large part of the simulator this way, it became obvious that a lot of code was being repeated. If the current phase was the acquire phase, the function would execute code that acquired a set

of ports. If the current phase was the verify phase, it would then check those same ports if it had indeed acquired them and for every operation that it would do in the commit phase, it checked if it could actually do it. This meant calculating addresses and calling functions and components which were later calculated *again* in the commit phase with the exact same code.

In order to avoid writing the same functionality three times for each cycle sub-phase, the acquire and verify code sections were dropped, and the functions called in the commit phase would internally check the current cycle phase and act accordingly. For instance, where there used to be three function calls, `port.acquire()`, `port.hasAcquired()` and `port.read()`, now there was only `port.read()` which internally called `acquire()` if the current phase is the acquire phase and `hasAcquired()` if the current phase is the verify phase. And to avoid performing each operation three times, several macros were constructed that could be easily used to make a section of code exclusive to the acquire, verify or commit phase.

For instance, the following C++ code:

```
bool addItem(void* item)
{
    if (nItems == maxItems) {
        return false;
    }

    COMMIT(
        buffer[nItems] = item;
        nItems = nItems + 1;
    )
    return true;
}
```

would translate to the following code after replacing the COMMIT macro:

```
bool addItem(void* item)
{
    if (nItems == maxItems) {
        return false;
    }

    if (GetCurrentPhase() == PHASE_COMMIT) {
        buffer[nItems] = item;
        nItems = nItems + 1;
    }
    return true;
}
```

which would prevent adding the item to the buffer during the acquire and verify phases. Note that the check for the buffer capacity is always performed, so that when this function is called in the verify phase, it will return false if the buffer is full. This implies that if this function is called in the commit phase, the buffer cannot be full (because otherwise the verify phase would have indicated that the commit phase cannot be run for the component).

5.5 Verification and validation

Verification of software is the process of ensuring that the software performs the way it is meant to perform. Validation, on top of that, checks that the software can actually perform the function for which it was created.

Because it is practically impossible to test the simulator with every possible permutation of parameters, the simulator was verified by running various test-cases during the stages of its development. These tests were constructed to test the cases where complex behavior was expected from the system. These test-cases were then run with varying simulation and test parameters such as number of processors, memory latency and number of threads. The simulator offers the user the ability to step through the simulation and examine the state of the various components on a per-cycle basis. This was used heavily to verify the correct behavior of initially trivial and later more complex programs where the expected behavior could be checked manually. Using this method of testing often revealed certain architectural design detail problems with the microthread model which were subsequently fixed by altering the model. Further examples of the usage of some of the test cases were to check that:

- the bypass busses in the pipeline were functioning by constructing test cases with subsequent instructions with register dependencies.
- thread suspension and wakeup was functioning by constructing cases where operations on registers used data that wasn't fetched from memory yet.
- operations calculated the correct results by examining the output of the pipeline stages and contents register file and memory after program execution, to compare the results with hand-calculated results.
- thread and family cleanup worked properly by creating cases that created a plenty families and then examining the state of the processor as those threads and families were created, replaced and cleaned up.
- the shared exchange protocol functioned properly by creating dependent families with different structures to verify the general cases of the protocol.
- pipeline flushes worked, especially in combination with bypass busses, by creating test cases that contained jumps, branches and instructions that would utilize the bypass busses in critical situations.

These verification and debugging methods, combined with the straight-forward per-component code that could be written in the six-phase-cycle framework, the simulator could be thoroughly verified.

To prevent future modifications of the simulator breaking old programs, the simulator will use regression testing where tests that uncovered previous bugs will be used to verify newer versions of the simulator, which will ensure that should any bugs be uncovered, they do not appear in later versions.

Validation of the simulator is implicitly contained in the verification methods. Since the verifications works by using the simulator in the manner for which it was designed (i.e., running fragments of micro-threaded code), this also validates the simulator.

6. Results

“There are three kinds of lies: lies, damned lies, and statistics.”

- Benjamin Disraeli

As part of the verification and validation of the simulator, several tests were run and their total cycle counts recorded. This could then be used to determine the speedup of the programs. It should be noted that these tests served as part of the verification that the simulator was working and were not used to verify the model itself. The latter could not be done because there are no previous experimental or formal models against which to compare the results. The simulator has in fact been developed to generate such results. So instead, the results are compared to a general expectation of the behavior of the model.

The tests involved multiplying two matrices together for different parameters of the simulations. The changed parameters were the block size (1, 2, 4, 8 and 16), thread table size (8, 16, 32, 64, 128 and 256 threads), family table size (8, 16, 32 and 64 families) and number of processors (1, 2, 4, 8, 16 or 32).

6.1 Setup

The test used the following high-level code to multiply two matrices together:

```
for (int i = 0; i < N; i++)
{
    for (int j = 0; j < N; j++)
    {
        int sum = 0;
        for (int k = 0; k < N; k++)
        {
            sum += A[i][k] * B[k][j];
        }
        C[i][j] = sum;
    }
}
```

This code was hand-compiled to four different assembly versions. The first version used no micro-threading constructs whatsoever and served as ‘compatibility’ comparison. The second version used transformed the first loop into a family of N micro-threads. The inner two loops were done with regular jumps. The third version transformed the first two loops into two levels of families of micro-threads. The inner loop was done with regular loops. The fourth version transformed all three loops into families of micro-threads.

Note that of the three micro-threaded versions, only the last one has a family with dependent threads. The *sum* variable was passed as a shared variable. Each thread in that family reads its elements from A and B, multiplies them, adds them to their dependent register and writes the result to its shared register. Finally, the parent thread will write the register back to memory into the C matrix.

N was fixed at 30 across all tests. This number was chosen because it was not directly divisible by the block size in many cases, was low enough to notice the expected processor starvation for higher block sizes and was low enough to result in reasonable simulation times.

As a final note on the setup, because the focus of the project was the implementation of the processor, and not of the memory, the memory system that has been implemented in the simulator is simple and not suited for achieving high performance given real-world memory latencies. To allow the simulations to better portray the effects of the processor parameters on the total execution time, the memory latency has been significantly reduced to about ten cycles per cache line.

6.2 Matrix Multiplication #1

For the first micro-threaded program, the results indicated that the size of the family table had no effect on the total time of the program. This is understandable since only a single top-level family is created. Even for the smallest family table size, 7 of the 8 entries are simply never used. Furthermore, the size of the thread table had no or negligible (< 0.1%) effect on the total execution time of the program. Thus, only the block size and number of processors had an effect on the execution time of the program, as the following graph illustrates:

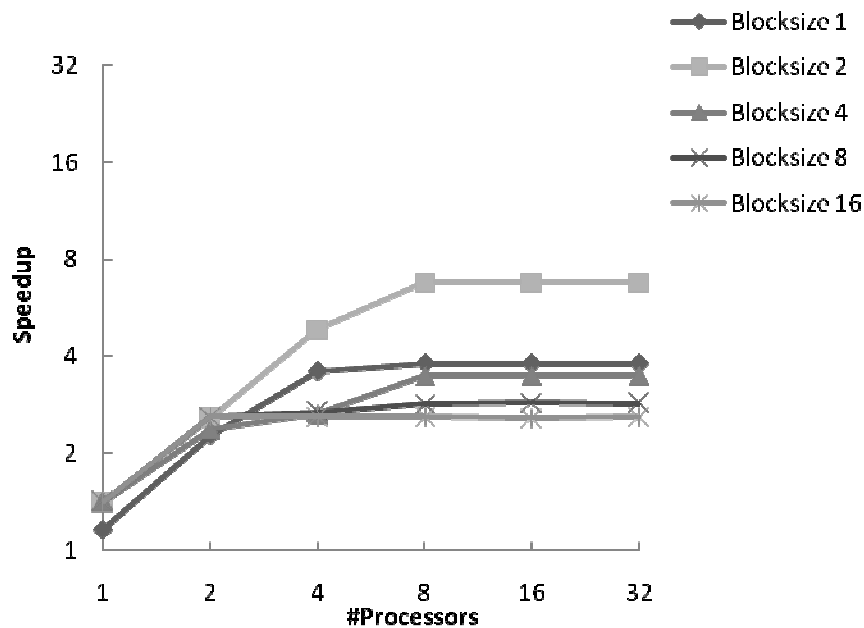


Figure 18 - Performance of matrix multiplication #1

As can be seen, a block size of 2 yields the best performance, even though it plateaus around 8 processors. It is not surprising that the case with a block size of 1 yields less performance since in that case, since there is only one family, there is only one thread allocated to each processor. When this thread suspends on a memory read or branches, the pipeline is inactive and wastes cycles. In the case where there are two or more threads allocated to each processor, the other threads can use these otherwise wasted cycles. It is interesting to note that increasing the number of threads per processor even further to 4 and above actually results in worse performance. This is not unexpected given that there are only 30 threads in the entire family. For a block size of 4, having more than 8 processors simply provides no additional

speedup as the extra processors have no threads allocated to them. For the block sizes of 8 and 16 this occurs even sooner. In the latter case only 2 processors are ever used.

6.3 Matrix Multiplication #2

The second micro-threaded version of the matrix multiplication contains nested families; each of the N threads in the top-level family creates a family of its own. As a result, there are a total of $(1 + N)$ families created. The size of the family table cannot be neglected in this case; however, from the tests it appeared that the size of the thread table and the block size had a far more significant impact on the execution times. The tests showed that the largest difference in execution times lay between a thread table size of 8 and 16 and up, so Figure 19 shows two speedup graphs for the various block sizes. The left graph shows the speedup with reference to the non-microthreaded version for a thread table size of 8 and the right table shows the speedup for all other thread table sizes. Each point represents the average speedup for all family table and thread table sizes applicable to the graph with the error beams indicating the maximum and minimum speedups across those parameters.

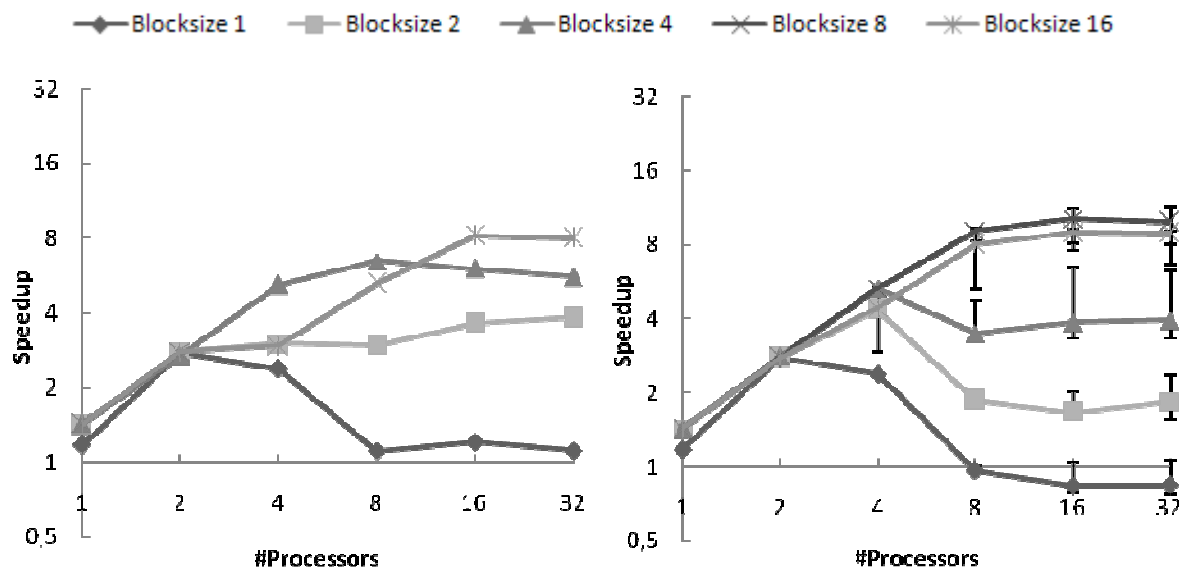


Figure 19 - Performance of matrix multiplication #2.
Thread Table size: 8 (left) and 16 and up (right)

The first thing worth noting is that a block size of 1 results in no significant speedup or even slowdown with respect to the non-microthreaded version for higher number of processors. This can be explained by noting that the threads in the top-level family do nothing more than create the second-level families and wait on them. Thus, there is only ever one second-level family with a single thread active on each processor. This means there are not enough threads to keep the pipeline full at all times. Combined with this is the overhead of creating the families on different processors and the fact that the effects of data-caching have been reduced due to the threads being spread across different processors. Taken together, it is not entirely surprising that the performance suffers in this case. The overhead and data-cache effects can also explain the reduced speedup for the cases with a block size of 2 and 4. Only for a block size of 8 and 16 does the benefit of multiple threads significantly offset these effects.

For the cases with a thread table size of 16 and up the availability of threads allow the cases with a block size of 8 and 16 to reach full performance. This performance is still tempered by the overhead of creating the families on multiple processors and the increased detrimental effects of multiple data caches. Also, the speedup graph does not plateau as clearly as with the first matrix multiplication. This can be attributed to the fact that because of the two-level families, there are now more threads, and those threads are more evenly spread across the processors, meaning all processors are utilized. However, the speedup does slow down due to the earlier mentioned effects of overhead and multiple data caches.

6.4 Matrix Multiplication #3

The third micro-threaded version of the matrix multiplication also contains nested families; each of the N threads in the top-level family creates a family of N threads of its own. Each of those N^2 threads represents a single output cell in the matrix multiplication and creates a family to sum the N multiplications. In this case, resource deadlock turned out to be an issue, most often apparent when the family table size was not bigger than the number of processors. In such cases, each processor would create a family and allocate a family entry on every other processor. With a family table size equal to or less than the number of processors, no additional entries are available to create the third nested family on any of the existing families. Consequently, no thread can continue, and deadlock has occurred. Figure 20 shows the average speedup for the third matrix multiplication program with the error beams indicating the maximum and minimum speedup across all thread table sizes and family table sizes. However, due to the previously mentioned issues, non-deadlock outcomes for block sizes of the 8 and 16 are rare and as such those block sizes have been omitted from the graph.

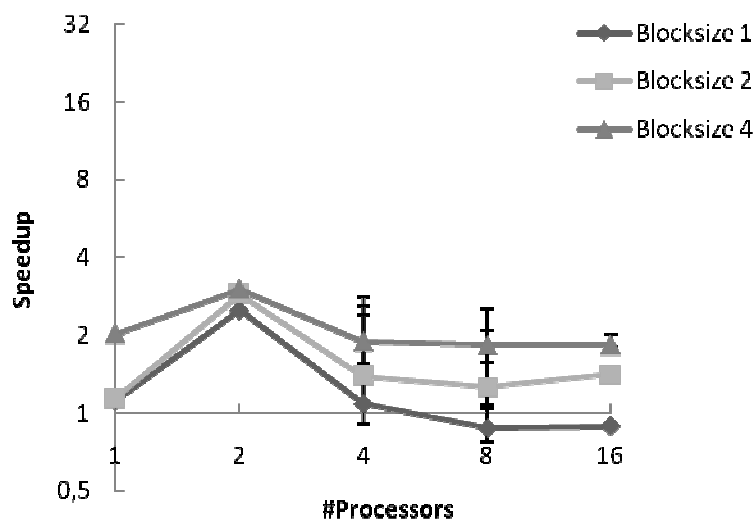


Figure 20 - Performance of matrix multiplication #3

Just like in the second matrix multiplication case, a block size of 1 results in slowdown for higher number of processors. However, in this case *all* block sizes result in significant worse speedup. This is not surprising because the third level family is a dependent family, with the shares written at the end of a thread's lifetime. This, in effect, serializes execution of the threads in those families. With a many threads waiting for their predecessor to write to their shares, their thread table entries are wasted and performance suffers. Add to this the overhead of transmitting shares between processors and the detrimental effects of multiple

data caches and not even the benefit of relatively high block sizes can overcome these disadvantages, as is the case in the second matrix multiplication tests.

7. Conclusion

"The future will be better tomorrow."

- Dan Quayle

With inherently serial processors running up against physical limitations in hardware, processor designers must look into truly parallel solutions if continued performance is to be achieved. Microthreading is a new processor architecture that aims to provide scalable and power-efficient concurrency by creating families of dynamically scheduled microthreads, which can be efficiently interleaved on a pipeline. Microthreads suspend when data they require isn't available, giving their execution data-flow characteristics, and allowing the processor to continue executing threads whose data *is* available, maximizing power efficiency and throughput.

This thesis explained the principles of microthreading and the design and implementation of the model in a software-based simulator as they were developed during this Master's project. The implementation was architecturally designed like a possible hardware implementation and was as such both an early test to the feasibility of the implementation of the model in hardware and a way to get early yet accurate results about the performance of the model. During and after development of the simulator, multiple tests were run in order to verify the simulator by examining, among others, important border cases, and in order to provide early results of the performance of the microthread model by measuring the execution time of an algorithm in various configurations. During development and testing of the simulator issues in the microthread model were uncovered and fixed. In the end, the results from the tests both supported the fact that the simulator implemented the desired model and that the model showed significant speedup across a variable range of processors without requiring a change in the binary code for certain algorithm configurations.

This resulting simulator is now being used by the research group to obtain performance results from programs and algorithms under various model configurations. For instance, the simulator is currently being used to investigate the cost and benefit of different memory protection schemes. Beyond this, it can also be used to generate memory requests to test and compare various memory implementations, to test the output of the compiler that is under development for this architecture or to investigate the effect of changing certain architectural parameters based on their implementation in silicon. The results from using the simulator in these ways can be used to improve the microthread model or its implementation with regards to performance or power efficiency.

References

1. International Technology Roadmap for Semiconductors, 2006 Update, <http://www.itrs.net/>
2. L Hammond, BA Nayfeh, and K Olukotun (1996) A Single-Chip Multiprocessor, *IEEE Computer*, 29(12): 84–89
3. Cotofana, S. and Vassiliadis, S. 1998. On the Design Complexity of the Issue Logic of Superscalar Machines. In *Proceedings of the 24th Conference on EUROMICRO - Volume I* (August 25–27, 1998). EUROMICRO. IEEE Computer Society, Washington, DC, 10277.
4. Wall, D. W. 1991. Limits of instruction-level parallelism. In *Proceedings of the Fourth international Conference on Architectural Support For Programming Languages and Operating Systems* (Santa Clara, California, United States, April 08–11, 1991). ASPLOS-IV. ACM Press, New York, NY, 176–188.
5. Rau, B. R. 1993. Dynamically scheduled VLIW processors. In *Proceedings of the 26th Annual international Symposium on Microarchitecture* (Austin, Texas, United States, (December 01 - 03, 1993). International Symposium on Microarchitecture. IEEE Computer Society Press, Los Alamitos, CA, 80–92.
6. M. S. Schlansker and B. R. Rau. EPIC: Explicitly Parallel Instruction Computing. *IEEE Computer*, 33(2):37–45, February 2000.
7. Schlansker, M. S. and Rau, B. R. (2000) EPIC: An Architecture for Instruction-Level Parallel Processors. *Compiler and Architecture Research*, HPL-1999-111. HP Laboratories Palo Alto.
8. G. S. Sohi, S. E. Breach and T. N. Vijaykumar. Multiscalar Processors. *Annual International Symposium on Computer Architecture*. 22:414, 1995.
9. Gontmakher, A., Schuster, A., and Mendelson, A. 2006. Inthreads: a low granularity parallelization model. *SIGARCH Comput. Archit. News* 34, 1 (Mar. 2006), 77–80.
10. D. Burger et. al. (2004) Scaling to the end of silicon with EDGE architectures, *IEEE Computer*, 37 (7), 44–55, IEEE
11. Swanson, S., Schwerin, A., Mercaldi, M., Petersen, A., Putnam, A., Michelson, K., Oskin, M., and Eggers, S. J. 2007. The WaveScalar architecture. *ACM Trans. Comput. Syst.* 25, 2 (May. 2007), 4.
12. A Bolychevsky, C R Jesshope and V B Muchnick, (1996) Dynamic scheduling in RISC architectures, *IEE Trans. E, Computers and Digital Techniques* ,143, pp309–317.
13. Luo B and Jesshope. C (2002) Performance of a Micro-threaded Pipeline, *Proc. ACSAC 2002 Australia Computer Science Communications*, Vol 24.