Tolerating Memory Latency Through Lightweight Multithreading

Andrew Gale

Submitted for the Degree of Doctor of Philosophy from the University of Surrey

Department of Computing
School of Electronics, Computing, and Mathematics
University of Surrey
Guildford, Surrey, GU2 7XH, U.K.

October 2001
Abstract

As processor clock frequencies continue to improve at a rate that exceeds the rate of improvement in the performance of semiconductor memories, so the effect of memory latency on processor efficiency increases. Unless steps are taken to mitigate the effect of memory latency, the increased processor frequency is of little benefit.

This work demonstrates how multithreading can reduce the effect of memory latency on processor performance and how just a few threads are required to achieve close to optimal performance.

A lightweight multithreaded architecture is discussed and simulated to show how threads derived from an application's instruction-level parallelism may be used to tolerate memory latency.
Acknowledgements

I would like to thank my supervisor Dr. Roger Peel for his invaluable support and guidance during the course of this study.

Thanks also go to my colleagues in the Distributed Systems Research Group and the Department of Computing, particularly Dr. Alex Shafarenko and Dr. Slava Muchnick, who have provided me with a stimulating research environment.

I would also like to thank my friends and family for the encouragement they have given me throughout.

Finally, special thanks go to my parents without whose support and encouragement none of this would have been possible.
Contents

1 Introduction 1

2 The problem of memory latency 3
  2.1 Methods of coping with latency 4
    2.1.1 Caching 4
    2.1.2 Pre-fetching 5
    2.1.3 Dynamic scheduling 6
    2.1.4 Multithreading 6

3 Multithreading 8
  3.1 Design of multithreaded architectures 8
    3.1.1 Latency tolerance vs. increased parallelism 8
    3.1.2 Context switch overhead 9
    3.1.3 Hardware support for memory latency tolerance 9
    3.1.4 The type of parallelism supported 10
  3.2 Example multithreaded architectures 10
    3.2.1 Simultaneous Multithreading, University of Washington 11
    3.2.2 APRIL/Sparcle, Massachusetts Institute of Technology 14
    3.2.3 Interleaving - Gupta's work at Stanford 15
    3.2.4 Denelcor HEP 17
    3.2.5 Tera 18
    3.2.6 Microthreading — Surrey 18
    3.2.7 Other multithreaded architectures 20
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>The amount of instruction-level parallelism in common applications</td>
<td>23</td>
</tr>
<tr>
<td>4.2</td>
<td>An attempt to produce threaded code from instruction-level parallelism</td>
<td>25</td>
</tr>
<tr>
<td>4.3</td>
<td>Techniques to increase the amount of instruction-level parallelism</td>
<td>29</td>
</tr>
<tr>
<td>4.4</td>
<td>Summary</td>
<td>33</td>
</tr>
<tr>
<td>5.1</td>
<td>Achievable processor efficiency</td>
<td>34</td>
</tr>
<tr>
<td>5.2</td>
<td>Three memory-throughput utilisation scenarios</td>
<td>36</td>
</tr>
<tr>
<td>5.3</td>
<td>The simulated uniprocessor and memory system</td>
<td>37</td>
</tr>
<tr>
<td>5.4</td>
<td>Implementation of the simulator</td>
<td>39</td>
</tr>
<tr>
<td>5.5</td>
<td>Experimental parameters</td>
<td>42</td>
</tr>
<tr>
<td>5.6</td>
<td>Results</td>
<td>43</td>
</tr>
<tr>
<td>5.7</td>
<td>Effect of (L_q) on performance</td>
<td>47</td>
</tr>
<tr>
<td>5.8</td>
<td>Summary</td>
<td>48</td>
</tr>
<tr>
<td>6.1</td>
<td>Latency &gt; (1/T_q)</td>
<td>52</td>
</tr>
<tr>
<td>6.2</td>
<td>Latency &lt; (1/T_q)</td>
<td>55</td>
</tr>
<tr>
<td>6.3</td>
<td>The cost of switching threads</td>
<td>57</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Switch-on-load</td>
<td>57</td>
</tr>
<tr>
<td>6.3.2</td>
<td>Switch-every-cycle</td>
<td>61</td>
</tr>
<tr>
<td>6.4</td>
<td>Performance of strategies to reduce the cost of thread switches</td>
<td>61</td>
</tr>
<tr>
<td>7.1</td>
<td>Summary of the design</td>
<td>66</td>
</tr>
<tr>
<td>7.1.1</td>
<td>Synchronisation</td>
<td>66</td>
</tr>
<tr>
<td>7.1.2</td>
<td>Long latency memory accesses</td>
<td>67</td>
</tr>
<tr>
<td>7.1.3</td>
<td>Thread creation and descheduling</td>
<td>67</td>
</tr>
<tr>
<td>7.2</td>
<td>Implementation considerations</td>
<td>68</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Invalid source operands</td>
<td>68</td>
</tr>
<tr>
<td>7.2.2</td>
<td>Returning data from the memory to the register set</td>
<td>70</td>
</tr>
<tr>
<td>7.2.3</td>
<td>Branches</td>
<td>71</td>
</tr>
</tbody>
</table>
# Contents

7.2.4 Thread management .............................................. 72
7.2.5 Condition codes .................................................. 73
7.2.6 Traps and interrupts .............................................. 74

7.3 Summary ................................................................. 75

8 A lightweight multithreaded processor: simulation 76
8.1 Implementing the simulator - software issues ................. 78
8.2 Thread management .................................................. 79
8.3 Data dependency on LOAD result .............................. 79
8.3.1 Thread creation .................................................... 80
8.3.2 Synchronising threads .......................................... 81
8.3.3 Two threads awaiting the same result ..................... 83
8.3.4 Two threads waiting on two shared results ............... 83

8.4 Performance ............................................................. 85

9 Multiprocessor efficiency 87
9.1 Network type ......................................................... 87
9.2 Router ................................................................. 88
9.3 Clocking ............................................................... 90
9.4 Nodes ................................................................. 90
9.5 Uniform random traffic ............................................ 91
9.6 Simulator implementation ......................................... 91
9.7 Experimental parameters ......................................... 91
9.8 Results ............................................................... 92

10 Conclusions .............................................................. 94

Bibliography ............................................................... 96
Chapter 1

Introduction

Multithreading is a programming method that promises to mitigate the effect of the large memory latencies found in modern shared-memory multiprocessors. By dividing a multiprocessor application into a number of parallel tasks (‘threads’), and switching execution to a different thread whenever the current thread encounters a long-latency memory event, multithreading should be able to hide many of the latencies that were not detected or were unpredictable at the time of compilation.

It seems natural that the number of threads in a multithreaded application must be related linearly to the memory latency of the machine on which the application is to be run, if the threading is to be sufficient to hide the latency. This is an assumption that has been challenged by research conducted by the Distributed Systems Research Group at Surrey. The research described in [Barsky96] and [Bolychevsky96] has shown that only a few threads are required to achieve peak performance and, beyond that, further threading does not result in further gains in performance. This is a result of queueing effects in the memory system, and the typical characteristics of the networks used to connect memories and processors in multiprocessor machines. The maximum attainable performance can be achieved with just three or four threads for a single-processor system, and two or three threads per channel per node for a multiprocessor system.

This finding challenges the approach of some other multithreaded research where applications are required to contain massive levels of concurrency. Instead, this low level
of threading should be attainable from the instruction-level parallelism of a single-threaded application, meaning that existing applications should be able to yield adequate performance, requiring just a re-compilation to produce threaded code (rather than a re-writing of the application in a multithreaded fashion).

As threads will be extracted from a non-threaded application, the threads will share the same register set and stack. This allows a simple dynamic thread scheduling mechanism to be employed, as described in [Bolychevsky96] and later in this thesis. The mechanism uses the processor's register set to hold state information relating to threads that are awaiting responses from memory, and the result is a multithreaded architecture that has low thread-switching overheads and requires a minimum of modifications to an existing processor. The small thread management overheads suggests that the short threads likely to be derived from an application's instruction-level parallelism are practical for hiding memory latency. Hence the scheme has been called 'microthreading' to reflect the fine granularity of the supported threads.
Chapter 2

The problem of memory latency

The latency of memory accesses is an increasingly significant problem in modern computer architectures as processor clock rates continue to increase. Although the number of data items that can be transferred between a processor and its memory in a given time unit (the 'memory bandwidth') can be made sufficiently high for most applications, the delay in returning a result to the processor following a memory request (the 'memory latency') still requires a processor to spend a large proportion of its time idle, and much research is directed towards reducing or eliminating this need to wait for memory.

In a uniprocessor machine, memory latencies typically range from nil for data stored in the primary cache to around 50 processor cycles for data stored in main memory (assuming that it is not necessary to access virtual memory pages held on disc). A shared-memory multiprocessor machine has several interconnected processors, each of which holds a portion of the global memory available to all processors in the machine. If a processor wishes to access data which happens to be stored in the portion of the global memory that it holds itself, then the memory access is said to be 'local'. If the data is held in another processor's memory, then the access is non-local and a request to the holding processor has to be sent across the interconnecting network or bus. Memory latencies are similar to those of a uniprocessor when data is local but can reach several hundreds of cycles when the data is non-local. Also, non-local memory latencies depend on the state of the network at the time of the memory request and are
2.1. Methods of coping with latency

Therefore less predictable than local memory latencies, meaning that some conventional latency hiding techniques are impractical.

This chapter discusses a number of methods of dealing with these long memory latencies and discusses some of the key features of systems that use multithreading to tolerate latency. The remainder of the chapter reviews the research of other groups investigating multithreaded systems.

2.1 Methods of coping with latency

The simplest way to build a processing system that did not suffer due to memory latency would be to increase the performance of the memory system until it matched the processor's requirements. Even if this was not impossible to achieve, it would be very expensive. Therefore, there are a number of methods to reduce the effect of memory latency without physically reducing the latency itself. Three general categories of methods of reducing the effects of latency exist - those methods that attempt to physically reduce the memory latency for the most frequently used data (caching), those that try to fetch required data in advance (pre-fetching) and those that 'hide' the latency by overlapping the memory access with independent instructions (dynamic scheduling and multithreading). These methods are described briefly in the following sections.

2.1.1 Caching

For programs where the same data is used several times in quick succession (temporal locality), or where sequential memory accesses are to memory locations whose addresses that are numerically close (spatial locality), cache memories can reduce memory latency effectively. Cache memories are small, high speed memories which hold copies of recently accessed data so that, if the data is cached (a 'cache hit'), the processor does not need to access the slow main memory. In the case of a cache hit, the memory latency is the latency of the high-speed cache memory. For a cache miss, the memory latency is that of the main memory.
Programs exhibiting temporal locality constantly re-use the same data items, and hence the cache tends to fill with copies of the most frequently used data, resulting in the majority of memory accesses producing cache hits. Also, because the cache fetches clusters of sequential memory addresses from main memory rather than individual words, data is frequently pre-cached for programs exhibiting spatial locality.

Scientific programs frequently do not exhibit much temporal or spatial locality, or have data sets that are too large for a cache to be able to cater for any temporal or spatial locality that the program does have. Because of this, the benefit of a cache to scientific programs is often minimal.

When caches are used in multiprocessors, it is common for a copy of a data item to appear in the cache of more than one processor. It is essential that, whenever such an item of data is accessed, these copies accurately reflect the most recent value written to that memory address by any processor. This is not easy to achieve, and most solutions use a lot of network bandwidth to ensure consistency between the caches [Stenstrom90, O'Krafka90]. Multiprocessors without caches, or multiprocessors with caches but no coherence logic, are much simpler to build, but require to be operated under a more restricted software regime where consistency issues cannot occur [Cook96].

2.1.2 Pre-fetching

For programs that do not exhibit temporal or spatial locality, but still access data in a predictable fashion, pre-fetching can be an effective way to reduce memory latency. Rather than fetching data only at the instant that it is needed in a computation (and stalling the processor whilst the data is returned), the data can be 'pre-fetched' a number of processor cycles in advance, giving the memory the chance to return the data in time for the computation.

Where the architecture allows it and the programs can be re-written, this pre-fetching of data can be triggered by inserting an explicit pre-fetch instruction in the program code, telling the memory unit the address of the data (or the cache line) to pre-fetch [Mowry98]. Because the programmer knows the memory access pattern of the program, software pre-fetching can be highly effective, but the time allowed for the memory to
2.1. Methods of coping with latency

respond has to be accurately predicted (or, at least, over-estimated) at the time the program is written.

An alternative hardware-based method is appropriate when the program code cannot be re-written. In this scheme, the hardware predicts the addresses generated by the most frequently executed \texttt{LOAD} instructions (using a mechanism similar to the branch history tables used for branch prediction) and issues pre-fetch requests [Chen95]. This hardware approach has the potential disadvantage of wrongly predicting the memory addresses and issuing useless pre-fetches, using-up memory bandwidth and, in the worst cases, causing 'cache pollution' by pushing more-useful data out of the cache.

2.1.3 Dynamic scheduling

When latencies are small, the dynamic scheduling mechanisms found in superscalar processors can occasionally hide the latency by overlapping the memory access with instructions that do not rely on the data being returned by the memory. The effectiveness of the latency hiding depends on the size of the processor's look-ahead window, and the number of independent instructions subsequent to the memory instruction. It is often possible to improve this performance with additional compiler optimisations such as trace scheduling and other techniques described later.

2.1.4 Multithreading

Both caching and pre-fetching attempt to reduce the amount of time for which a processor needs to idle on a memory read, preferably eliminating it altogether. Multithreading [Thekkath94, Weber89, Agarwal93] is an attempt to tolerate (rather than reduce) memory latency by switching to a different task until the memory responds. In order to do this, a program needs to be split into a number of parallel tasks (called 'threads') which are independent of each other. Dynamic scheduling relies on hardware to detect, at run-time, independent instructions, but a multithreaded machine requires this independence to be stated explicitly in the code.

Each thread needs its own workspace and usually this means that each thread has its own set of registers. These registers, along with the program counter (a pointer to the
next instruction in a thread) and any other related state information, form the thread's context and, whenever a thread is replaced with another thread, the context for the current thread needs to be stored and the context for the next thread needs to be restored. This context switching takes time and storage space. In order for any benefit to be gained from multithreading, the context switch time must be lower than the latency being tolerated. Some multithreading schemes might choose to change threads only when the latency is expected to be larger than the time taken to switch threads, assuming the latency is predictable.
Chapter 3

Multithreading

3.1 Design of multithreaded architectures

There are a number of design decisions that separate the various multithreading research projects that the author has reviewed; prime differences being the context switch time, the amount of hardware support required to achieve that switch time, the type of parallelism the architecture is intended to support, and whether or not the goal is actually latency tolerance.

3.1.1 Latency tolerance vs. increased parallelism

The main motive for using a multithreaded system over a single-threaded system is usually that the former takes advantage of the latency tolerance it offers or, in some circumstances, exploits the fact that certain tasks are easier to program in a parallel fashion. Another benefit, however, is that the explicit parallelism in the code can be used to simplify the execution of more than one instruction at any one time: a multiple pipeline machine could have each pipeline being fed with a separate thread. This would be significantly easier to design than more conventional superscalar machines which attempt to unravel a program's code into parallel instructions as it runs. The performance gain is not immediately obvious, however: we would expect a three-pipeline machine to execute a threaded benchmark program in a third of the single-pipeline execution.
time, if there was no effect from memory latency. But, if memory latency is significant, then we may find that our spare threads have been ‘wasted’ on producing instructions for the other pipeline to execute, rather than being held in reserve for covering latency.

Multithreaded multiple-pipeline machines would have benefits over VLIW machines where the parallelism is fixed and one pipeline can be stalled by another’s actions (since the parallel instructions are locked together in bundles). Although parallel threads may start simultaneously, they do not have to progress at equal rates.

3.1.2 Context switch overhead

Switching from one thread to another on a cache-miss can improve processor performance if the time taken to perform the switch is less than the time taken to service the cache miss. Heavyweight threads with large amounts of context information could take several hundreds of cycles to be switched, especially if performed in software. Some multithreading schemes replicate portions of hardware (e.g. the register set) so that there is less activity required at each switch.

3.1.3 Hardware support for memory latency tolerance

All processors that exploit multithreading require some sort of hardware assistance if they are to tolerate the smallest latencies offered by memory systems. In most cases this means that all the processor’s resources that hold context information (i.e. the registers, program counter, etc.) are replicated and simply switched in or out as appropriate. This can be expensive if a large number of threads are to be supported, and if there are more logical threads than physical threads then some form of software intervention is usually required. The more radical the architectural changes required to implement threading, the more obvious the benefits of threading will have to be, in order to convince processor manufacturers that the cost of change is worthwhile. One design that uses a combination of hardware and software to implement a low-cost threading solution is the Sparcle [Agarwal93].
3.1.4 The type of parallelism supported

The success of a multithreaded machine depends very much on the availability of suitable parallel code with which to feed it. In an unsophisticated scenario, this could come from multiple applications, where each thread is a separate user program, i.e. the machine would be used as a multi-programmed machine. The cache performance could be rather poor for such a scheme, as threads will have large, non-overlapping, data sets. For such a multi-programmed approach, threads could be millions of instructions long.

A more likely scenario would be for the threads to be drawn from within the same application, and the easiest way of achieving sufficient parallelism is for the programmer to write the application in an explicitly parallel manner, possibly using a parallel language, such as Occam, where the process parallelism is very easy to extract. Another method is to use a parallelising compiler which recognises certain forms of parallelism within the program and exploits them. Such compilers are usually called ‘vectorising’ or ‘loop-unrolling’ compilers, and would be expected to perform well on scientific codes, FORTRAN programs potentially having a high rate of success under such schemes. Thread lengths could be tens or hundreds of instructions. High Performance Fortran programs provide more detailed parallel information in the source code which may be exploited even more effectively.

The last method to be considered here is the use of instruction-level parallelism (ILP) where threads are extracted from the implicit parallelism within the machine code. This need not be used in isolation - it is conceivable that threads produced by any of the other methods will contain ILP which could be exploited as separate threads. Threads derived from ILP are potentially short (probably less than ten instructions), and the cost of implementing such short threads must be appropriately low. Various techniques are available that can enhance the levels of ILP that may be found in most code.

3.2 Example multithreaded architectures

Several major multithreaded architectures are surveyed in the following sections.
3.2.1 Simultaneous Multithreading, University of Washington

Research conducted at the University of Washington has investigated simultaneous multithreading (SMT), where a superscalar processor attempts to issue as many instructions as possible in each cycle, grabbing potential instructions from any thread. Unlike single-issue multithreading, all threads are active simultaneously. Rather than using multithreading to tolerate the long memory latencies found in a multiprocessor, their prime focus is on obtaining speedup on an SMT uniprocessor, running applications traditionally considered to be serial. [Tullsen99]

[Tullsen95] investigates SMT as a method of maximising processor utilisation for a superscalar processor. The authors simulate a processor based on the Alpha 21164 but with larger on-chip lock-up free caches, ten functional units, and a maximum issue rate of eight instructions per cycle. Dynamic execution is limited: each thread has an eight-instruction scheduling window which receives in-order dependence-free instructions, and can issue them out of order. This significantly simplifies the design, and removes the need for register renaming. Advanced static scheduling (using the Multiflow trace scheduling compiler) is used to increase the chance of being able to execute several instructions in parallel.

Taking the performance of the processor running a single thread as a base measurement, they find that for the SPEC92 benchmark suite, the average processor utilization is 19% i.e. an average execution of 1.5 out of the maximum of 8 instructions per cycle.

To support multithreading, up to eight hardware contexts are permitted. They study a number of strategies when executing instructions:

Fine grain multithreading: in each cycle, instructions from just one thread may be issued. This method results in 1.5 to 3.2 instructions being issued per cycle, a maximum speedup of 2.1 over single-threaded operation. This gain is primarily due to the elimination of pipeline bubbles rather than through making use of more functional units per cycle, and the execution time improves little when more than 4 threads are used.
3.2. Example multithreaded architectures

Full simultaneous issue: all threads compete to issue instructions during the same cycle, and each can issue up to the maximum number of instructions (eight). This strategy results in an instruction rate as high as 6.3 instructions per cycle, and a speedup ranging from 3.2 to 4.2 over the single threaded case. The number of instructions issued per cycle improves as the number of threads is increased, with no dramatic levelling-off in the range simulated (1 to 8 threads).

Restricted simultaneous issue: similar to the ‘full simultaneous issue’ above, but each thread is restricted to issuing either one, two, or four instructions per cycle. This method shows better results than the fine-grained multithreading, except for the case where only one instruction may be issued in each cycle, and the benchmark has less than five threads. The case where threads are restricted to issuing no more than 4 instructions per cycle achieves most of the performance of the full simultaneous scheme.

Limited connection: each hardware context has access to only one of each type of functional unit (e.g. if there were two adders, only threads 1, 2, 3, and 4 could access the first adder, and only threads 5, 6, 7, and 8 could access the second adder). This shows performance somewhere between the performance of fine-grain multithreading and restricted simultaneous issue with a limit of two issues per thread per cycle.

The workload for these simulations is not a single application from which a number of threads have been drawn. Instead, each thread is an entirely independent application from the SPEC92 suite. This lack of dependencies between threads means there are no synchronisation delays.

[Tullsen96] reduces the number of changes that are needed to add SMT support to a standard superscalar processor, and simulates the memory system more fully. The base superscalar architecture has register renaming, and out-of-order issue, in-order completion of instructions. Rather than replicating every architectural feature for each thread (as was done in [Tullsen95]) the authors now simply replicate the program counter and the return stack which is used to predict subroutine return addresses.
Branch target buffer entries are tagged with a thread ID number, and the register file is enlarged to support logical registers for each thread and extra registers for renaming. The instruction queue is shared between all threads.

Because of the enlarged register set, register access takes two pipeline cycles, increasing branch misprediction penalty by one cycle, and requiring an extra level of bypass logic. The processor has 9 functional units (3 floating point, 2 integer, 2 integer/load/store).

Again, each thread runs an independent application from the SPEC92 suite, compiled with the Multiflow trace scheduling compiler, but trace scheduling is turned off (unlike in [Tullsen95]) because software speculation is less useful for this architecture, which features hardware speculation. When running a single thread, performance is just 2% below a standard superscalar processor. When making use of the SMT feature, results show that with eight threads they can achieve an issue rate of 5.4 instructions per cycle, a 2.5 speedup over the single-threaded superscalar processor. They find that with 200 physical registers, performance decreases with more than 4 threads.

Essentially, [Tullsen96] is [Tullsen95] with a more realistic and easier-to-implement architecture, and more accurate modelling of that architecture.

A problem for superscalar processors when managing the mapping of logical registers onto physical registers is knowing when a physical register is no longer needed (i.e. its contents have been read by all instructions that need to use it) and can be de-allocated. This unnecessary tying-up of registers can reduce the scope for issuing instructions. In the past, the only way a processor has known that it can reallocate the register is when a new value is written to the register, but in [Lo97b] they propose using an explicit 'free register' instruction (they actually offer five variants of such an instruction) to free-up physical registers so they can be reallocated. Their results show that, for small register files, execution speed can be improved by an average of 77%.

[Tullsen99] proposes a low-overhead synchronisation method which allows a thread to acquire a memory-based lock in one cycle (or so it seems to that thread). An architectural feature called a lock-box implements ACQUIRE(LOCK) and RELEASE(LOCK) primitives in hardware, descheduling the thread when the lock is not available, and rescheduling it when another thread releases the lock. This removes the need for the
thread to constantly attempt and fail to acquire the lock with a test-and-set primitive, wasting processor cycles. By hand-modifying source code for a number of benchmarks, the authors found that they were able usefully to parallelise a number of benchmark applications that their parallelising compiler previously considered not worth parallelising.

Simultaneous multithreading has recently been introduced into a commercial device [Intel01]. Intel's 'HyperThreading' technology duplicates all the registers necessary to indicate an application's state on a processor. Instructions are fetched from two applications simultaneously and issued to the functional units in an attempt to occupy every functional unit every cycle. The two virtual processors appear as two separate processors in a multiprocessor system to the programmer and operating system. [Intel01] reports a performance gain of up to 30% for multithreaded code when using HyperThreading technology.

3.2.2 APRIL/Sparcle, Massachusetts Institute of Technology

Sparcle is the first implementation of APRIL, a processor architecture developed at Massachusetts Institute of Technology to power the MIT Alewife cache coherent distributed shared memory machine. APRIL supports what they call 'coarse-grain' multithreading, where contexts are only swapped when a long latency instruction is encountered (This type of multithreading is also commonly referred to as 'blocked' multithreading). APRIL is described in [Agarwal90], and the details of the Sparcle implementation are given in [Agarwal93].

Each thread has its own set of registers, and a number of register sets are duplicated in hardware so that rapid context switching can be achieved between those threads in the working set. Context switching is performed by software, but the duplicate register sets remove the need to save registers, thereby reducing the context switch time. Contexts are swapped only when long latency events such as cache misses or synchronisation faults occur, and not cycle-by-cycle or on data-dependence stalls. The floating-point register set is not duplicated, but is partitioned so that each thread has a reduced number of registers privately available to it.
3.2. Example multithreaded architectures

Sparcle is a slightly modified SPARC processor with support added for fast traps on cache misses and synchronisation faults. The SPARC core used has eight register windows, and these are used in pairs to provide four register sets - one for fast traps, and three for user threads. Threads are denied the usual SPARC feature of using register windows to pass parameters to procedures, but the authors claim [Agarwal92] that with suitable register allocation techniques they can equal the performance benefit of windowing. When a cache miss or synchronisation fault occurs, the external hardware indicates this to the processor by signalling on one of two specially implemented fast-trap pins. The trap handler saves the program counter (PC) and processor status word (PSW) of the existing thread in one of its registers, moves the register window to point to the new thread's register set, restores that thread's PC and PSW, and then restarts the thread. This software sequence takes 14 instructions, which is sufficiently low when compared to the 50-cycle or so latencies which they hope to tolerate.

When simulating Sparcle in a 64 node Alewife machine [Kurihara91] they find that increasing the number of threads from 1 per processor node to 2 per processor node results in an application speedup of 20% and an increased demand for network bandwidth of approximately 25%.

In [Agarwal92], the author produces several graphs of processor utilisation vs. number-of-processes to study the effect of varying context switch overhead, cache size, cache miss rate, and network radix. Interestingly, the processor utilization is almost identical when the context switch overhead is 0 cycles and when it is 4 cycles. Their results show that, with large caches, they can achieve around 90% processor utilisation with just 2 to 4 threads.

3.2.3 Interleaving - Gupta's work at Stanford

Weber & Gupta [Weber89] simulate a multithreaded processor in a directory based cache coherent multiprocessor which supports between two and four hardware contexts, each with their own set of registers. The context is changed on a cache miss or a write hit to read-shared data. They investigate the performance as the number of contexts, the context switch time, and the network latency vary. Results show that processor
utilisation improves significantly for all applications when two contexts are used in place of a single context, and that there is further improvement when four contexts are used, though the improvement over two contexts is more application-dependent. The improvement is significant when the context switch time is just one cycle or four cycles, but there is little improvement (and in some cases a degradation) in processor utilisation when the context switch time is sixteen cycles. They find that processor utilisation degrades less rapidly with increasing network latency when the application is multithreaded, and that degradation is noticeably worse for 2 contexts than it is for four contexts.

Cache performance degrades with additional contexts, and degradation with reducing cache size is worse for the multithreaded processor than for the single threaded processor (particularly with a very small cache). Progressing from one context to four contexts results in the cache miss rate increasing from 1.6% to 2.2% for a 64Kbyte cache, and from 2.6% to 7.5% for a 16Kbyte cache.

[Landon94] is concerned with improving single threaded code execution on a multithreaded machine, arguing that too much multithreaded research is multiprocessor based and manufacturers are only likely to make modifications to their processors if uniprocessor performance were to be improved. The authors claim that block multithreaded machines, where the processor executes a single context until a long-latency event occurs, offer poor performance because they have to stall on data dependencies and suffer from a relatively high context switch time. This is because cache misses are not detected until late in the pipeline, and all previous stages have to be flushed, wasting several cycles. The alternative of cycle-by-cycle context switching is better, but traditionally such machines have been implemented without hardware interlocks and without caches. This means that single thread performance is abysmal since an eight-stage pipeline can only execute one instruction every eight cycles, and every memory reference is a long-latency reference.

The authors propose an interleaved multithreaded processor where contexts are switched on every cycle, hardware interlocking is provided, and caches are supported. Consequently, when a cache miss occurs, only those instructions belonging to the thread that
3.2. Example multithreaded architectures

caused the miss need be squashed, and not all previous pipeline stages. Threads are
drawn from a pool of threads that are not waiting on memory accesses, and in the
case that there is only one thread to run, hardware interlocking resolves dependencies,
stalling the pipeline when necessary. When there are available threads, cycle-by-cycle
interleaving reduces the amount of stalling compared with a block multithreaded pro­
cessor.

For two of their benchmarks run with four contexts on a uniprocessor, they find through­
put increases (over the single threaded case) by 23% and 9% for a block multithreaded processor, but this increases to 65% and 46% respectively for their interleaved scheme.

Multiprocessor simulations (based on the Stanford DASH architecture) show interleaved
multithreading to be slightly better than block multithreading, the interleaved scheme
providing an average speedup of 74% in one case, versus 46% for the block multithreaded scheme.

3.2.4 Denelcor HEP

The Denelcor HEP [Smith81, Dennis94] was a commercial shared memory multiproces­
sor which used multithreading to tolerate memory latencies. The system consisted of
of up to 16 processing nodes and up to 128 memory nodes. As memory was not local
to any processor and there were no caches, all memory accesses were remote and had
considerable latency - a typical time between the request and return of an item of data
being 2.4 microseconds [Hockney88].

Each processing node could process up to 128 threads in groups of up to 16 tasks, 8
of which could be user tasks. Each task was assigned a block of registers (from a pool
of 2048) and a block of data memory. Threads were switched between on a cycle-by­
cycle basis, runnable threads being held in the process queue before being issued to the
pipeline. When a thread wished to access memory, it was removed from the process
queue until the memory request was satisfied, and then returned to the process queue.

The pipeline was eight stages deep, and no thread could have more than one instruction
in the pipeline at a time. This saved the need to check for data dependencies and
stall the pipeline when they occurred. It also meant that, when there were fewer than eight schedulable threads, the pipeline had a number of unused slots. Single-thread performance was therefore one eighth of the peak instruction issue rate, which is abysmal.

3.2.5 Tera

The Tera [Hwang93, Bokhari98, Dennis94] is a development of the HEP and Horizon (see section 2.2.6) architectures and, like them, it interleaves threads on a cycle-by-cycle basis. A full-scale Tera is intended to have 256 processor nodes and 512 memory nodes. As with the HEP, there are no caches. Rather than sharing a [segmented] register file, each thread (up to a maximum of 128) has its own hardware context which consists of a stream status word, 32 general purpose registers, and 8 target registers. The stream status word holds the program counter, mode descriptor bits, and condition codes.

Tera instructions are VLIW in nature, containing three operations, one of which can be a memory operation, one an arithmetic operation, and one which can be a control operation or another arithmetic operation. Each instruction has a 3-bit look-ahead field which indicates how far away the nearest dependent instruction (in the same thread) is. In other words, it indicates how many instructions can be executed before a pause is required to allow the result [that the later instruction is dependent on] to be written back to the register file. Where there is sufficient instruction level parallelism, this scheme eliminates the need for an instruction to completely finish executing before the next instruction from the same thread can enter the pipeline, thereby improving single thread performance (c.f. Denelcor HEP). The Tera pipeline, at 21 stages, is longer than HEP's, and therefore needs more available threads to keep it busy. Memory latencies are much longer than 21 cycles, though, so having just 21 threads is not sufficient to keep the processor busy.

3.2.6 Microthreading — Surrey

By considering the queueing in the memory system of both a conventional memory and a networked memory, [Barsky96, Bolychevsky96] show that there is an upper limit to
the gain in performance that can be obtained from the use of additional threads. More importantly, that upper limit is not determined by the latency of the memory (for most single-processor memories), or the size of the network.

They show that the maximum attainable performance for a uniprocessor system is obtained with just three to four threads. This is a surprisingly low number, and this result is not dependent on the actual value of the latency. This result holds for the three levels of load on the memory system they investigate — when a program exerts a load requiring 0.5, 1.0, and 2.0 times the maximum available memory bandwidth. When the desired throughput is twice the maximum permitted throughput, the maximum attainable performance is 50%. This is as expected: the application is simply too demanding for the memory, and the only solution is to improve the memory bandwidth or reduce the memory bandwidth required by enhancing the program's cache performance.

For a networked memory system as found in a typical multiprocessor, they show that the number of threads required to achieve the maximum attainable performance is just two threads per outgoing channel at each node. A two-dimensional torus would need just eight threads per node to achieve maximum performance, and a three-dimensional torus would need twelve threads per node.

Because of the low number of threads required to approach maximum performance, they present a low-overhead 'microthreaded' processor designed to exploit threads extracted from instruction level parallelism. Threads extracted from an application's instruction-level parallelism are derived from the same context, i.e. they share the same stack frame, and can use the same register set. Taking advantage of this, [Bolychevsky96]'s architecture has just a thread's program counter (PC) as the state that needs to be saved when descheduling a thread. Furthermore, the architecture takes advantage of the fact that the PC is sufficiently small to enable a large number of ready-to-run threads to be held in a hardware 'continuation queue' in the processor. Thread-switching can then be implemented entirely in hardware and the switch time can be reduced to zero cycles. The small size of the PC permits a novel synchronisation method where the thread information of a sleeping microthread can be stored in a processor register.

To enable threads to synchronise, registers can be marked as invalid. An attempt to
3.2. Example multithreaded architectures

read an invalid register results in the PC of the offending instruction being written to the register and execution of that thread ceasing. When valid data is written to that register, the PC it holds gets 'pushed-out' into the continuation queue to allow rescheduling later. Load instructions are converted to 'invalidate destination register' instructions if a cache miss occurs. When data is returned by memory the value is written to the register and the PC of any thread awaiting the data is returned to the continuation queue.

They also propose marking all instructions with two tags to allow the compiler to statically schedule code based on its knowledge of which loads may result in cache misses. The same two tags can be used to create and kill threads.

[Bolychevsky96]'s uniprocessor result seems increasingly exciting as queued memory systems become more common due to load-store units being decoupled from the main processor pipeline in modern processors. The Sun MAJC-5200 [Sun99] processor, for example, allows up to five outstanding requests to main memory.

3.2.7 Other multithreaded architectures

The Horizon [Iannucci95] is an architecture from the designers of the HEP and Tera, thought it was never built. Like the HEP and Tera, it features a deep pipeline and uses massive parallelism to cover memory latency, as well as the pipeline latency. Rather than relying on the hardware to determine data dependencies within a thread (and hence whether or not another instruction could be issued from that thread before the previous one had completed), the Horizon has a field within its instruction word to hold a small integer. This integer indicates to the hardware the number of subsequent instructions that are independent of this instruction - in other words, how many more instructions the hardware may issue before having to wait for a result.

The M-Machine [Fillo95] is a 3d-mesh connected multicomputer with multithreaded processing nodes. Each node has 12 functional units which can execute 12 operations from a single thread or operations from up to 6 threads at once. Each node has sufficient resources to support up to six V-threads, each of which consists of up to 4 H-threads, a H-thread being a three-operation-wide instruction stream. H-threads are interleaved
cycle-by-cycle on a cluster (a quarter of the node’s functional units). Clusters do not have to be executing the same V-thread as other clusters.

The Multilisp Architecture for Symbolic Applications (MASA) [Iannucci95] is a multithreaded architecture which, like the Denelcor HEP, interleaves the execution of multiple threads on a cycle-by-cycle basis. Active threads are assigned a ‘task frame’ which includes a register set, the thread’s program counter, and pointers to the task frames of the thread’s parent and child threads. Threads can access the registers of their parent and child, and this is used to implement a parameter passing mechanism for procedure calls, similar to that of register windowing in the SPARC architecture. The MASA has a limited number of hardware task frames, so software intervention is required to swap contexts between hardware frames and memory. Each memory location has a number of tag bits to support the garbage collection required for programming languages such as Lisp, and memory accesses can cause traps according to the state of these tag bits. These are handled by a fast-trap mechanism.
Chapter 4

Attaining the required parallelism

The microthreaded processor described in [Bolychevsky96] is designed to exploit threads drawn from an application’s instruction-level parallelism. Previous research, detailed in [Barsky96] and [Bolychevsky96], has discovered that just two threads per network node are all that is required to reach maximum attainable efficiency in a multithreaded multiprocessor machine. Three to four threads are sufficient for a single-processor machine.

Section 4.1 briefly describes a number of studies that attempt to determine the degree of instruction-level parallelism within common applications, and what these findings mean for thread-extraction from those applications. Section 4.2 describes work conducted by the Distributed Systems Research Group into the level of threading possible in a selection of uniprocessor applications. Some methods of how the levels of instruction-level parallelism can be enhanced are then described.
4.1 The amount of instruction-level parallelism in common applications

Instruction-level parallelism (ILP) is heavily exploited in virtually all new processors, either directly by the processor through superscalar techniques, or by the compiler for very-long instruction-word (VLIW) processors. Consequently there have been a number of studies that attempt to determine the maximum amount of parallelism within an application - in other words, the number of instructions that can be issued in parallel if hardware constraints are ignored.

Those ILP limit studies [Tjaden70, Jouppi89] which failed to consider movement of code between basic blocks found disappointing results with average ILPs of just 3 or 4 instructions capable of being issued in parallel per cycle. The study described in [Wall91] removed most hardware constraints and studied parallelism along an instruction trace produced by running their benchmark programs, rather than from static analysis. In contrast to the aforementioned studies, movement of code across basic block boundaries was permitted, and ILPs as high as 60 instructions per cycle were reported.

The results of these studies can be interpreted easily in the context of superscalar and VLIW machines: once the major hardware assumptions such as infinite functional units, infinite registers, and penalty-free branching have been addressed, the only thing preventing these processors exploiting the maximum ILP is the size of the superscalar processors look-ahead window, or the width of the VLIW machine's instruction word. How the results of these ILP studies apply to the availability of threads for a microthreaded processor is less clear. It is tempting to think that an average ILP of 3 means that an average of three threads should be available, but this is not necessarily the case. Consider this piece of sequential code:

\[
\begin{align*}
  r1 &:= 100 \\
  r2 &:= 200 \\
  r3 &:= r1 - r2 \\
  r4 &:= r1 + r2
\end{align*}
\]

cont...
This code has an ILP of two because a superscalar or VLIW machine could issue the instructions in the following pairs:

\[ \begin{align*}
\text{r1} & := 100; & \text{r2} & := 200 \\
\text{r3} & := \text{r1} - \text{r2}; & \text{r4} & := \text{r1} + \text{r2} \\
\text{r5} & := \text{r3} \times \text{r4}; & \text{r6} & := \text{r3} / \text{r4}
\end{align*} \]

Despite the average ILP of two instructions per cycle, it is not practical to split this code into parallel threads. The only threading possibility would be for two parallel threads to be created, one executing the \text{r1}:=100 instruction, and the other executing \text{r2}:=200 instruction. Once these two instructions had both completed, the threads would need to join and then spawn two more parallel threads for the next two instructions. The overheads of thread creation and joining are certainly going to be too high to tolerate threads as short as this, especially if there are no memory accesses to cover.

However, when higher average ILPs are detected, such as the sixty instructions per cycle of [Wall90], threading seems more viable. ILP studies usually generate their results by modelling a machine with infinite hardware resources and, though this renders the results unobtainable for practical machines, the results are still promising: in a multithreaded processor we are interested in using the parallelism to cover memory latency, and do not want to run threads physically in parallel, so the lack of large numbers of functional units does not matter. At the instant where such a study finds an ILP of thirty, this thirty-wide bundle of instructions could be divided into three parallel threads, each containing ten instructions (providing there are sufficient registers).

Most ILP limit studies use a number of advanced code manipulation techniques to enhance the ILP available, some of which are described later in this chapter.
4.2 An attempt to produce threaded code from instruction-level parallelism

[ChisSter98] examines a number of typical application programs to determine the availability of threads within them. By simulating a single-issue statically-scheduled SPARC processor and running highly optimized code through it, run-time dependency information was gathered, enabling a multithreaded code generator to split the application into a number of threads. The advantage of gathering information at runtime rather than attempting to derive it statically from source files is that memory accesses are no longer ambiguous, possibly enabling the production of more parallel threads. However, the values of the addresses may vary according to the application’s input data, and so the threading derived after one particular run is valid for that input data only. This produces an optimistic threading of the code.

The multithreaded code generator in [ChisSter98] only considers code within a single basic block, which is somewhat limiting as most ILP limit studies that restrain themselves to not moving code between basic blocks find only a small amount of parallelism. Also, the simulator does not enter the code of library functions, which somewhat distorts the results. This makes it impossible to reason about the possibility of threading across procedure calls/returns, since all threads are required to join prior to a procedure call/return - this happens anyway, due to the fact that threads are extracted only from within a single basic block.

Once run-time data has been gathered for an application, the multithreaded code generator builds a dependency graph for each basic block, eliminating all false dependencies where two instructions share a register. Hence they are unable to execute in parallel, even though there is no meaningful flow of data between the them. The multithreaded code generator then attempts to divide the instructions into a number of parallel threads $T_1$, $T_2$, $T_3$, with preceding and following sequential sections $S_i$ and $S_f$ respectively, as illustrated in 4.1. The sequential sections of code $S_i$ and $S_f$ should be short and, preferably, non-existent.

A register re-allocation phase takes place, where each parallel thread is given its own
4.2. An attempt to produce threaded code from instruction-level parallelism

distinct set of registers to use. Because the analysis is restricted to a single basic block, it proves problematic to identify exactly which registers are available for use - some registers may be unused within a basic block, but may store useful data for use by surrounding code. The register allocation scheme used is somewhat pessimistic, only permitting the use of those registers which were modified by the original basic block. This does not always provide sufficient registers, and in such cases the division of code into threads is reviewed. This may mean that many potential threads are not realised due to a lack of register availability. [ChisSter98] admits that it would be possible to identify more available registers, but found it too computationally expensive to do so.

The applications studied are the integer applications Flex, Yacc, and Compress, which are compiled with a high degree of optimization. This makes threading more difficult, but reduces the chance of producing 'fake' threads from superfluous instructions. Table 4.1 shows the results for basic blocks containing more than eight instructions and
4.2. An attempt to produce threaded code from instruction-level parallelism

<table>
<thead>
<tr>
<th></th>
<th>Yacc</th>
<th>Flex</th>
<th>Compress</th>
</tr>
</thead>
<tbody>
<tr>
<td>static average block size</td>
<td>10.9</td>
<td>12.3</td>
<td>15.2</td>
</tr>
<tr>
<td>runtime operations inside those basic blocks</td>
<td>8.5%</td>
<td>30.7%</td>
<td>33.4%</td>
</tr>
<tr>
<td>runtime memory loads inside those basic blocks</td>
<td>14.1%</td>
<td>39.9%</td>
<td>49.5%</td>
</tr>
<tr>
<td>runtime average memory loads per basic block</td>
<td>3.7</td>
<td>2.9</td>
<td>2.7</td>
</tr>
<tr>
<td>those blocks successfully split into threads</td>
<td>40%</td>
<td>57%</td>
<td>81%</td>
</tr>
<tr>
<td>threaded blocks with Si of length 0</td>
<td>78%</td>
<td>75%</td>
<td>77%</td>
</tr>
<tr>
<td>threaded blocks with Sf of length 0</td>
<td>56%</td>
<td>59%</td>
<td>55%</td>
</tr>
</tbody>
</table>

Table 4.1: Profile of those basic blocks with more than 8 instructions

<table>
<thead>
<tr>
<th></th>
<th>Yacc</th>
<th>Flex</th>
<th>Compress</th>
</tr>
</thead>
<tbody>
<tr>
<td>static average block size</td>
<td>17.7</td>
<td>17.8</td>
<td>21.2</td>
</tr>
<tr>
<td>runtime operations inside those basic blocks</td>
<td>3.5%</td>
<td>11.2%</td>
<td>25.5%</td>
</tr>
<tr>
<td>runtime memory loads inside those basic blocks</td>
<td>6.2%</td>
<td>14.0%</td>
<td>36.7%</td>
</tr>
<tr>
<td>runtime average memory loads per basic block</td>
<td>5.76</td>
<td>3.8</td>
<td>3.0</td>
</tr>
<tr>
<td>those blocks successfully split into threads</td>
<td>54%</td>
<td>63%</td>
<td>100%</td>
</tr>
<tr>
<td>threaded blocks with Si of length 0</td>
<td>57%</td>
<td>65%</td>
<td>80%</td>
</tr>
<tr>
<td>threaded blocks with Sf of length 0</td>
<td>63%</td>
<td>65%</td>
<td>60%</td>
</tr>
</tbody>
</table>

Table 4.2: Profile of those basic blocks with more than 12 instructions
table 4.2 shows the results for basic blocks containing more than twelve instructions.

The results for some applications are a little disappointing at first glance. Looking at table 4.1 we see that, with Yacc, only 40% of the basic blocks can be split into threads, meaning that approximately 6% of the memory accesses in the application are covered by threaded code. The situation is somewhat better for Compress where 81% of the blocks can be split, and this equates to covering some 40% of the memory accesses for that program.

A simple statement such as '40% of memory accesses covered by threaded code' does not provide any clue as to the expected performance. It would be quite acceptable not to cover the memory accesses which are most likely to be in caches close to the processor providing that those accesses that are hidden are of long latency.

[ChisSter98] sees one of the main problems of extracting code as being the shortage of reasonable length basic blocks, giving the result that threads are short and there are usually no more than two of them. There are two responses to this: the first is that, if the overheads of thread creation and synchronisation are low, the length of any threads created may not be terribly significant. The second response is that short basic blocks only matter when the scheduling strategy is restricted to extracting threads from within a single basic block.

The number of blocks for which the start and end sequential sections are of zero length is promising. This suggests that it may be possible to blend adjoining basic blocks into a single block and improve the scope for decomposition into threads. Compilation techniques such as superblocking and trace scheduling (described in section 4.3) enhance the amount of ILP by exploiting parallelism across basic block boundaries.

[ChisSter98] works on the assumption that each thread will take one cycle to be formed and another cycle to synchronise and terminate, hence a block with \( n \) threads needs to recover \( 1 + 2 \times (n - 1) \) cycles in order to justify its creation. Considering the low percentage of memory accesses that are covered by threading and the small probability of a cache miss, the chance of recovering 2 cycles per thread seems low. If, however, the thread create and synchronise-and-destroy overhead is virtually zero, then this is less of a worry.
4.3 Techniques to increase the amount of instruction-level parallelism

[ChisSter98] highlights some areas in which progress can be made to generate longer threads. As well as trace scheduling, one suggestion is to thread across boundaries, possibly using a join-branch primitive, and including sub-programs into threads so that multiple parallel threads can call functions without corrupting the stack.

Summarising, it should be possible to overcome a number of limitations within this study:

- the benchmarks are not easily parallelisable and scientific code could show more promising results.
- threading is limited to within one basic block.
- there is some hesitation to produce short threads for fear that the overheads involved will be too high.
- the data used to aid thread extraction was gathered by running the applications using specific input data. Memory disambiguation for more general data sets may produce significantly fewer threading opportunities.

4.3 Techniques to increase the amount of instruction-level parallelism

Simply constructing a data dependency graph from a stream of machine instructions and identifying independent instructions for execution as separate threads will not produce a large number of reasonable length threads. Re-use of registers in sequential code creates anti-dependencies and output-dependencies which inhibit parallel execution of instructions. Such false dependencies can be removed through the well-understood technique of register renaming, which can occur dynamically at run-time (as in a superscalar processor), or statically at compile-time (such as for a VLIW machine). Although register renaming does have a significant impact on the ability to execute instructions in parallel, the improvement is unlikely to be sufficient to provide the required number of threads at all times.
To increase the number of available threads, a variety of techniques can be used to modify the application code. The most obvious and easily identifiable source of parallelism is code that features regular loops. Scientific code will frequently use loops which apply the same operations to all the elements of a large data set. Consider the following code example:

```plaintext
FOR j := 0 TO 99 DO
BEGIN
   { pseudo machine code for a[j] := a[j] + b[j] }
   r1 := a[j]
   r2 := b[j]
   r1 := r1 + r2
   a[j] := r1
END

Providing subsequent iterations are independent of each other, which is the case for this example, the loop can be unrolled:

FOR j := 0 TO 49 DO
BEGIN
   r1 := a[j*2]
   r2 := b[j*2]
   r1 := r1 + r2
   a[j*2] := r1

   { pseudo machine code for a[j*2+1] := a[j*2+1] + b[j*2+1] }
   r1 := a[j*2+1]
   r2 := b[j*2+1]
   r1 := r1 + r2
   a[j*2+1] := r1
END
```
This loop unrolling provides us with the opportunity to run the two portions of code within the loop body in parallel. By renaming the registers in the second portion of the new loop body, we eliminate any dependency between the two sections of code, and they may now be executed in parallel as separate threads. More threads can be created by further unrolling the loop.

The example used above is sufficiently simple to be able to reason about the dependency between iterations - in other words, it is clear that a \([j]\) and a \([j+1]\) in subsequent iterations are to different memory locations, and hence there are no dependencies due to memory usage.

It is not always so obvious that memory accesses are to different locations, and this is where 'memory disambiguation' methods are essential. Such methods determine that two memory accesses are dependent, independent, or impossible to reason about. If memory accesses are independent then those instructions may form separate parallel threads. If the accesses are dependent then the associated instructions must be synchronised appropriately.

When it is impossible to reason about the dependence between two memory accesses, they have to be treated as though they are dependent. Ideally memory disambiguation will occur at compile-time and threads be created accordingly. Where this is not possible, there can be some benefit in doing address comparison at run-time [Nicolau89] and altering the flow of the program accordingly - possibly following a stream of sequential code when memory accesses are found to be dependent, and a parallel version of the code when memory accesses are independent. As well as possible dependencies due to memory usage, another complication to producing sufficient threading is control dependence. In a simple scenario, threading is restricted to within a basic block, and all threads are required to join before a conditional branch decision is encountered. This would not be such a problem if conditional branches occurred infrequently, but typical RISC code features conditional branches every few instructions. Therefore, most basic blocks are short, and the scope for producing reasonable-length threads from within them is minimal.

The solution is to enlarge as many basic blocks as possible by moving code from the
4.3. Techniques to increase the amount of instruction-level parallelism

basic blocks following a conditional branch to the block above the branch. Code in the basic blocks following a branch is normally only executed when the branch is taken in that block's direction, hence this movement of code above the branch results in speculative execution of that code. Ordinarily, code will be moved from the basic block that is at the most common destination of the branch, but care must be taken so as not to alter the correctness of the program - sufficient state needs to be left intact so that the program can still pursue the code at the less-frequently-executed destination of the branch.

Trace scheduling [Fisher81, Ellis85, Lowney93] is a technique frequently used in compilers for VLIW and superscalar machines to identify large numbers of independent instructions for execution in parallel. The program is studied as a whole to determine the most likely path of execution through it, called a trace. This is done through a combination of static branch prediction and studying the run-time behaviour of the program with test data representative of a real workload. Once the most probable path through the code is determined, loop-unrolling may be employed on the loops within that path. As the compiler knows the most probable sequence of execution of basic blocks, it then proceeds to move code across basic block boundaries, as described in the paragraph above. The trace scheduling compiler is willing, if necessary, to greatly increase the number of instructions on the least-probable paths in order to win additional parallelism on the most-probable paths.

Trace scheduling can be difficult to implement due to the need to ensure correct program behaviour when execution strays onto the less-probable paths. This is addressed in [Hwu93] where the idea of a 'superblock' is introduced. A superblock is a trace of instructions which does not have any 'side-entrances', i.e. other code cannot branch into the superblock - the only entrance is at the very top. This is achieved by restricting the type of code-motions that can be employed, and results in simpler 'book-keeping' (the process of ensuring correct program behaviour).

The final ILP-enhancing technique to be mentioned here is software pipelining, where loops are re-formed so that the loop body no longer contains all the instructions from one loop, but a selection of instructions from a number of subsequent iterations. This
means that the work of one iteration of the original loop now takes several iterations of the new loop to complete, but that several iterations of the original loop are in progress simultaneously - similar to the hardware concept of pipelining. The result is that the instructions within the new loop are more likely to be independent, and easier to divide into parallel threads.

4.4 Summary

A number of instruction-level parallelism enhancement techniques have been used successfully to generate code for VLIW and superscalar processors. Studies of these techniques suggest that they will be able to yield sufficient ILP from typical applications to produce the required number of threads for a microthreaded processor, though this will not be confirmed until there exists a compiler that has been written to extract those threads.
Chapter 5

Uniprocessor efficiency

5.1 Achievable processor efficiency

[Bolychevsky96] contradicts the assumption that the number of threads required by a multithreaded processor to achieve optimal efficiency is dependent primarily on the latency of the physical memory. The authors show that the effect of queueing outstanding memory requests from multiple threads is the dominant factor in determining the maximum attainable processor efficiency, and that typically just three or four threads are required to achieve near-optimal performance.

They make a number of assumptions in their modelling of a low-overhead multithreaded uniprocessor with queued memory system, namely:

- each thread may have just one outstanding memory request — i.e. a thread is put to sleep as soon as it issues a memory request, and is only free to be re-scheduled once the memory request has completed.

- the rate of arrival of memory requests to the memory system queue fits a Poissonian distribution.

They consider a uniprocessor connected to a memory system which can queue a number of outstanding memory requests. These queued requests are served in a first-come, first-
served fashion by the physical memory which has fixed latency and throughput. They show that for a multithreaded uniprocessor with \( n \) threads:

\[
GL(G) = n \left( 1 - \frac{G}{R} \right)^{\frac{1}{n}}
\] (5.1)

Where \( G \) is the average memory throughput actually granted to a program and \( R \) is the "requested" memory throughput, i.e. the value that \( G \) would take when the program was run on a processor with a zero-latency, infinite-throughput memory system. The processor efficiency (the fraction of the time for which it is executing code rather than stalling on an outstanding memory request) is \( G/R \).

\( L(G) \) in eqn. 5.1 is the latency of the memory system which, due to the existence of the queue for access to the physical memory, is a function of \( G \). The overall latency of the memory system is the sum of the time that a memory request spends waiting in the queue before it is permitted to access the physical memory, and the latency of the physical memory once access has been granted:

\[
L(G) = L_0 + L_q
\]

Here, \( L_0 \) is the [fixed] latency of the physical memory, and \( L_q \) is the average amount of time a memory request spends in the queue. If the physical memory has a throughput of \( T_0 \) then this is the rate at which the queue is serviced. The rate at which memory requests join the queue follows a distribution with a mean of \( G \), and queueing theory tells us that for this \( M/D/1 \)

\[
L_q = \frac{G}{2T_0(T_0 - G)}
\]

For most real memories, \( L_0 = 1/T_0 \). Substituting \( (L_0 + L_q) \) for \( L(G) \) in (5.1) gives us:

\[
G \left( \frac{1}{T_0} + \frac{G}{2T_0(T_0 - G)} \right) = n \left( 1 - \frac{G}{R} \right)^{\frac{1}{n}}
\] (5.2)
5.2 Three memory-throughput utilisation scenarios

The authors of [Bolychevsky96] consider three levels of load exerted by a program on the memory system:

1. \( R = T_0 \) — the memory throughput requested by a program is equal to the maximum available memory throughput, i.e. the memory system is fully loaded.

2. \( R = T_0/2 \) — the requested memory throughput is only half that of the physical maximum, i.e. the memory system is underutilised.

3. \( R = 2T_0 \) — the requested memory throughput is twice that of the physical maximum, i.e. the memory system is doubly overloaded.

Solving eqn. 5.2 numerically shows that, for all three scenarios, no significant further gain in performance is achieved by having more than 3 to 4 threads. The expected processor efficiencies are shown in table 5.1 and plotted in figure 5.1. Note that when the memory system is doubly overloaded, it is impossible to achieve an efficiency of greater than 0.5.

<table>
<thead>
<tr>
<th>Number of threads</th>
<th>Requested memory throughput, ( R )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( T_0/2 )</td>
</tr>
<tr>
<td>1</td>
<td>0.620</td>
</tr>
<tr>
<td>2</td>
<td>0.900</td>
</tr>
<tr>
<td>3</td>
<td>0.986</td>
</tr>
<tr>
<td>4</td>
<td>0.999</td>
</tr>
<tr>
<td>5</td>
<td>0.999</td>
</tr>
<tr>
<td>6</td>
<td>1.000</td>
</tr>
<tr>
<td>7</td>
<td>1.000</td>
</tr>
<tr>
<td>8</td>
<td>1.000</td>
</tr>
</tbody>
</table>

Table 5.1: Expected processor efficiencies [Bolychevsky96]
5.3 The simulated uniprocessor and memory system

A block diagram of the system simulated to experimentally verify [Bolychevsky96]'s results is shown in figure 5.2.

The system falls into two parts: the processor and the memory system. The processor contains a 'core', although this really represents just the memory access stage of a CPU. Alongside the core is a queue that holds a pool of identifiers of threads that are available to be scheduled for execution by the core. Whenever it is idle, the core obtains a thread from the thread queue and executes it until a LOAD instruction is encountered. When the LOAD executes, the core sends a request to the memory system and stops executing that thread. On the next cycle the core will attempt to obtain a new thread from the thread queue.

When the memory system receives a new request, it places it into the memory request
5.3. The simulated uniprocessor and memory system

queue. Whenever the physical memory is idle, it takes a request from the queue and services it for a period $L_0$. Once the physical memory has finished servicing the request it sends details of the thread that issued the request to the processor’s thread queue where the thread waits to be re-scheduled. (It should be noted that no attempt is made to return any valid data from the memory system).

A thread consists of only two types of instructions, namely \texttt{NOP} (no-operation) instructions and \texttt{LOAD} instructions. The decision as to whether an instruction is a \texttt{NOP} or a \texttt{LOAD} is made just before execution with a probability of $1/(L_0/2)$, $1/L_0$, and $1/(2L_0)$ for the memory loads of $R = T_0/2$, $R = T_0$, and $R = 2T_0$ respectively.

A sample execution trace for two threads is shown in table 5.2. At cycle 0 the core is idle and there are two threads T1 and T2 in the thread queue awaiting scheduling. In cycle 1 the core takes a ready thread (T1) from the thread queue and starts to execute it until it encounters a \texttt{LOAD} instruction in cycle 2. It then passes a memory request to the memory system, the request being temporarily placed in the memory queue (in the same cycle). On the next cycle (cycle 3), the idle memory takes a request from the memory queue and starts to service it. Note that although the memory request has been placed in the memory queue, its queueing time has actually been zero cycles.
because the memory was free to accept the request immediately — the purpose of it temporarily entering the queue was simply to ensure that the memory did not start to service the request until the next cycle after the one in which the \texttt{LOAD} was encountered.

Also in cycle 3, the core fetches another thread \text{T2} and starts to execute it. In cycle 4, thread \text{T2} features a \texttt{LOAD} instruction and a memory request is made to the memory system. This time, the request is not removed from the memory queue on the very next cycle because the memory is still busy servicing the memory request from thread \text{T1}.

During cycles 3, 4, 5, and 6, the memory is busy servicing the request from thread \text{T1} — the number of cycles taken is \( L_0 \) which, in this case, is 4 cycles. In cycle 7 the memory passes details of its finished request back to the processor and the thread identifier for the thread to which the \texttt{LOAD} belonged (\text{T1}) is placed into the thread queue. In the same cycle the core gets \text{T1} from the thread queue and starts to execute it. Also in cycle 7, the memory begins to service the queued memory request from thread \text{T2}.

Note that, if the core is idle, a returned thread passes through the thread queue in zero cycles. If this wasn’t so and there was a thread queue minimum latency of \( d \) cycles, \( L(G) \) would be \((L_0 + L_q + d)\). For the results presented in [Bolychevsky96], \( d = 0 \).

### 5.4 Implementation of the simulator

The implementation of the memory system is shown in figure 5.3, and is split into three sections — the memory unit, the memory system, and a router. The router exists so that a processor + memory system node could be attached to a network of similar nodes with the data held in each node’s memory being available to the processors of every other node. A \texttt{LOAD} generated by the core would either be to locally held memory or memory held in a remote node.

A demultiplexer sends the memory request to the local memory queue or to the router depending on whether it is a request to local or remote memory. Another demultiplexer determines whether completed memory requests should be returned to the local processor or a remote processor (via the router). A third demultiplexer splits packets
5.4. Implementation of the simulator

<table>
<thead>
<tr>
<th>cycle</th>
<th>thread queue</th>
<th>core</th>
<th>memory queue</th>
<th>memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T1;T2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>T2</td>
<td>T1: NOP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>T2</td>
<td>T1:LOAD</td>
<td>T1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>T2: NOP</td>
<td>T1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>T2:LOAD</td>
<td>T2</td>
<td>T1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>T2</td>
<td>T1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>T2</td>
<td>T1</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>T1</td>
<td>T1: NOP</td>
<td>T2</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>T1: NOP</td>
<td>T2</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.2: Two threads passing through the simulated system. \( L_0 = 4 \text{ cycles} \)

from the router into memory requests from remote processors (which it sends to the memory queue) and returned memory requests for this node's processor which have been satisfied by a remote node's memory. Such returned memory requests are sent to the processor for the corresponding thread identifier to be placed in the thread queue.

Since two memory requests (one from the local CPU and one from a remote CPU received via the network) may need queueing in one cycle, a multiplexer is used to prioritise the local request before placing them both in the memory queue sequentially in the same cycle. Other multiplexers are used at the input to the router and for returning completed memory requests to the processor.

All multiplexers and demultiplexers have a latency of zero cycles.

In the case of a uniprocessor, all reads and writes are to local memory, and no router (or network of similar nodes) is necessary. By incorporating the facilities for a networked memory system at this early stage, it has been possible to avoid two separate implementations of a memory system for a uniprocessor and multiprocessor. This avoids the possibility of small discrepancies in behaviour complicating a comparison of uniprocessor results with multiprocessor results. For the simulations presented in this chapter, a 'dummy' router was attached to the memory system.
The simulator is implemented in the Occam parallel programming language, with each unit in figure 5.3 corresponding to one Occam process. In order to simplify the tracing of requests through the system, and to model more closely a physical implementation, the system in synchronised. A synchroniser process receives 'ready' signals from each block in the system. Once it has a ready signal from every block, it issues a 'proceed' signal to all the blocks. Upon receiving the proceed signal, each block performs as much computation as it would be able to in a clock cycle in a real implementation before sending a ready signal to the synchroniser and awaiting the next proceed signal.
The synchroniser is implemented in a tree fashion with a topmost synchroniser processor and a hierarchy of 'fanout' synchroniser processes leading to each unit. This arrangement makes it simple to expand the design by adding extra blocks, and to replicate whole nodes when simulating a multiprocessor system. It also allows for portions of the design to be replaced, for example a range of processors could be attached to the memory system (providing they all feature a common memory interface).

As well as acting as the clock, the synchroniser receives diagnostic text from each block which it prints at the end of every cycle.

In order to simplify the implementation, the router is required to accept two packets on each cycle (one new read request and one completed request), avoiding the need for a queue before the router. One or both of these packets may be empty, and the average number of packets sent to the router each cycle will be significantly less than unity in a reasonably loaded system. Similarly, the continuation queue and memory queue are both required to accept two inputs per cycle.

5.5 Experimental parameters

In order to experimentally recreate the three scenarios considered in [Bolychevsky96], it was necessary to vary either the latency (and hence throughput) of the fixed memory or to vary the requested throughput of the program being executed on the processor. It was decided to keep the latency $L_0$ constant at 16 cycles and to have the probability of an instruction being a LOAD as 1 in 32, 1 in 16, or 1 in 8. These LOAD probabilities relate the case where $R = T_0/2$, $R = T_0$, and $R = 2T_0$ respectively. All other instructions were NOP instructions, taking one cycle to complete.

For each of the three load scenarios, the number of threads in the system was varied from one through to eight threads. The total number of instructions executed in a simulation was chosen to keep the number of LOADs around 100,000, the instructions being divided equally between the threads.

The simulation parameters are summarised in figure 5.3.
5.6 Results

The measured processor efficiencies are tabulated in 5.4 and plotted against the expected efficiencies in 5.4. The efficiency was taken as the total number of cycles required to run the simulation divided by the total number of instructions in a simulation run.

The efficiency observed with 7 and 8 threads when $R = 2T_0$ is slightly greater (0.501) than the expected physical maximum of 0.500. The total number of LOADS that were randomly generated for these two runs was slightly below the intended 100,000 and this would account for the higher-than-expected efficiency.

In all three scenarios, the measured efficiency for a given number of threads is higher than that predicted in [Bolychevsky96], the level of variation from the expected shown graphically in figure 5.5.

It can be seen that the variation from the expected is at its greatest with a small number of threads. This would appear to be due to one of the assumptions used in [Bolychevsky96], namely that the interval between the arrival of memory requests at the queue is distributed according to a negative exponential distribution.

The model insists that each thread may only have one outstanding memory instruction. This means that when all threads are sleeping [i.e. each has an outstanding memory request] there are no threads left to schedule, hence the memory queue can not receive

<table>
<thead>
<tr>
<th>$R$</th>
<th>probability that instruction is a LOAD</th>
<th>total number of instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_0/2$</td>
<td>1 in 32</td>
<td>3,200,000</td>
</tr>
<tr>
<td>$T_0$</td>
<td>1 in 16</td>
<td>1,600,000</td>
</tr>
<tr>
<td>$2T_0$</td>
<td>1 in 8</td>
<td>800,000</td>
</tr>
</tbody>
</table>

Table 5.3: Simulation parameters ($L_0 = 16$ cycles)

As the instruction type (LOAD or no-operation) was decided randomly, the exact number of LOADS in a given run varied, but this was never found to be by more than one percent.
5.6. Results

Table 5.4: Measured processor efficiencies ($T_0 = 16$ cycles)

<table>
<thead>
<tr>
<th>Number of threads</th>
<th>Requested memory throughput, $R$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$T_0/2$</td>
</tr>
<tr>
<td>1</td>
<td>0.667</td>
</tr>
<tr>
<td>2</td>
<td>0.908</td>
</tr>
<tr>
<td>3</td>
<td>0.975</td>
</tr>
<tr>
<td>4</td>
<td>0.993</td>
</tr>
<tr>
<td>5</td>
<td>0.998</td>
</tr>
<tr>
<td>6</td>
<td>0.999</td>
</tr>
<tr>
<td>7</td>
<td>0.999</td>
</tr>
<tr>
<td>8</td>
<td>1.000</td>
</tr>
</tbody>
</table>

Figure 5.4: Measured processor efficiencies
Figure 5.5: Fluctuation from measured efficiency from expected efficiency
5.6. Results

a new memory request. When all the threads are sleeping, the interval before the next memory request is queued must be greater than that given by the negative exponential distribution. The end result is that the average queue length is smaller than predicted, and that the average amount of time for which a memory request must be queued, $L_q$, is shorter than was anticipated, reducing the overall latency and returning the thread to the thread queue more quickly than was predicted.

The three graphs in figures 5.6, 5.7, and 5.8 show the actual distribution of the interval between subsequent LOAD instructions for memory loads of $R = T_0/2$, $R = T_0$, and $R = 2T_0$ respectively. It can be seen that for a small number of threads the actual distribution is skewed to the right of the expected, i.e. the average number of clock cycles between subsequent LOADs is higher than expected.

![Graph showing distribution of interval between loads](image)

*Figure 5.6: Distribution of interval between loads, $R = T_0/2$, $L_0 = 16$*

Figure 5.9 plots the average time a request was found to spend in the memory queue against the expected queueing time for the three memory load scenarios. It can be seen
5.7 Effect of $L_0$ on performance

According to the expected results, the processor efficiency should be independent of the actual value of $L_0$ (where $L_0 = 1/T_0$). In order to verify this the simulation was run with four values of $L_0$, namely 2, 8, 32, and 128 cycles. The results for a memory load of $R = T_0$ and a total of 250,000 instructions per run are plotted in 5.10 (the result for $L_0 = 32$ is omitted for clarity).

As can be seen, the performance improves with decreasing $L_0$. Suppose that the thread queue holds all the threads and that the memory and memory queue are idle. Consider what happens if the next instruction to be executed for each thread is a LOAD — the
memory queue will rapidly fill with queued memory requests. If there are $n$ threads then at cycle $n$ there will be $n - 1$ requests in the memory queue and the first request made (now being serviced by the memory) will be $n$ cycles towards being completed. It will be a further $L_n - n$ cycles until the memory request is completed and the core has a free thread to execute. This is the worst-case scenario.

5.8 Summary

We have found that the processor efficiency is actually better than expected due to the average time spent queueing for memory being less than anticipated. This is due to the fact that no new requests can enter the queue when all threads are sleeping. Also, although the effect of queueing is still the dominant factor in determining the processor efficiency, the latency of the memory does still have an effect.
Figure 5.9: Average queueing time, $L_0 = 16$
Figure 5.10: Effect of latency on efficiency, $R = T_0$
Chapter 6

Efficiency of a multithreaded uniprocessor

The processor simulated in the previous section represented an ideal system with no performance cost associated with implementing multithreading. Two characteristics in particular were different from a likely physical implementation of a single-pipeline multithreaded processor, namely:

- \( \text{Latency} = 1/T_h \) — as soon as the memory returned the data for the memory request, it started servicing a new request. This is a reasonable model for a cache-less memory system, but may not be appropriate for a cached memory system where main memory accesses result from the need to perform a cache line refill. Also, once a memory request was completed, its thread became available for rescheduling immediately. In a real processor there is likely to be a delay of at least one cycle and possibly more before the request can pass through the thread queue to be rescheduled.

- no thread switching cost — once a thread had issued a memory request the core was able to start execution of a new thread immediately (providing there was a free thread waiting in the thread queue). A real implementation might suffer a penalty when switching threads.
This chapter considers the effect on processor efficiency when the above two assumptions prove not to be valid.

6.1 Latency > 1/\(T_0\)

In the experiments presented earlier, a thread that was awoken by data being returned from main memory was able to be scheduled in the very next cycle if the core was idle. In other words, the minimum amount of time a thread spent in the thread queue was zero cycles.

A real implementation is likely to take one or more cycles to process the data returned by memory, possibly to match the returned data with the details of a sleeping thread, or because the implementation of the thread queue has been simplified by insisting that all threads must pass through it (rather than bypassing an empty queue).

In order to study the effect of this minimum thread-queue latency, the simulator was expanded to feature a delay of a fixed number of cycles before queueing a thread corresponding to a returned memory request. This was implemented by attaching a shift-register of \(T\) stages to the input end of the thread queue, as illustrated in figure 6.1.

![Figure 6.1: Simulator for system with lower limit on thread queue latency](image)

The processor was connected to the same memory system as used previously and the simulations performed with a main memory latency \(L_0\) of 32 cycles and a memory throughput \(T_0\) of 1/\(L_0\). The probability that an instruction was a LOAD was set to 1 in 32, i.e. the program was attempting to use the full memory bandwidth available. The simulator was run for a total of 100,000 instructions which were divided into 1 to 8 independent threads.
The value of $T$, the number of shift register stages in the simulation and the minimum number of cycles a thread takes to be awoken and rescheduled once the memory request on which it is sleeping completes, was taken to be 1, 2, 4, 8, or 16 cycles. These are 3%, 6%, 13%, 25%, and 50% of $LQ$, respectively.

Figure 6.2 plots the processor efficiency measured for some of these simulations, and figure 6.3 shows the results for all the simulations, but normalised [for each number of threads] to the efficiency when $T = 0$.

![Graph showing processor efficiency with different thread queue latencies](image)

**Figure 6.2: Processor efficiency with a lower limit on thread queue latency**

As expected, the efficiency decreases as $T$ increases, but the impact on efficiency is diminished as the number of threads increases — even with $T$ being as high as 50% of $LQ$ the processor efficiency is still 90% of the maximum achievable with three or more threads.

The performance loss is only considerable when there are just one or two threads in the system. This is not surprising — with just one thread the latency of a LOAD will
be \((L_0 + L_q + T)\), with \(L_q\) being 0. Since we have set the probability of an instruction being a LOAD to \(1/L_0\), the execution time for a total of \(I\) instructions will be:

\[
\text{execution time} = I \left( 1 + \frac{1}{L_0} (L_0 + L_q + T) \right) = I \left( 2 + \frac{T}{L_0} \right)
\]

The processor efficiencies for \(T = 8\) and \(T = 16\) are shown only to illustrate the trend in the reduction of the processor efficiency with the increase of \(T\) — a real system is likely to have a minimum wake-up-and-reschedule cost of just a few cycles.
6.2 Latency < 1/T_0

The results in the previous chapter were all produced from a simulated memory system with the characteristic \( L_0 = 1/T_0 \), where \( L_0 \) is the latency of a memory access and \( T_0 \) is the maximum throughput. Whilst this is true for accessing a single memory location, it overlooks the fact that most accesses to physical memory are in the form of cache line fill requests. A cache line of 8 words (with a word-wide data bus to memory) will require 8 sequential memory accesses. The data at the \texttt{LOAD} address (the ‘critical’ data) is usually returned first, allowing the sleeping thread to be placed in the thread queue for rescheduling as soon as possible. However, the memory is not then free to accept a new request because it needs to complete the 7 reads needed to fill the remainder of the cache line, each read taking another memory cycle. Furthermore, a number of writes may be necessary to write-back any ‘dirty’ words from the evicted cache line.

Common memories are able to provide data from an address that is sequential to the previous request much more rapidly than it can for a non-sequential address. It would be quite possible to have an implementation where, assuming that the memory returns the critical word first and ignoring write-back of dirty data, \( L_0 = L_{\text{setup}} + 1 \). Here, \( L_{\text{setup}} \) is the set-up time for a read from physical memory. The throughput might then be given by \( T_0 = 1/(L_{\text{setup}} + C + D) \) where \( C \) is the number of memory accesses required to perform a line fill and \( D \) is the number of memory accesses needed to write back dirty data from the evicted cache line. If the cache line size was 8 words and the memory bus was 2 words wide, then \( C \) would be 4.

If the memory system ran on a clock frequency \( M \) times smaller than the CPU’s clock (as is typical) then \( L_0 = M(L_{\text{setup}} + 1) \) and \( T_0 = 1/M(L_{\text{setup}} + C + D) \).

Summarising, it is perfectly reasonable to consider a system where \( L_0 < 1/T_0 \). The actual relationship between \( T_0 \) and maximum physical throughput of the physical memory will depend on the design of the cache line refill mechanism, the line size, and possibly the number of dirty words in the evicted line.

The graph in Figure 6.4 shows the effect of having a latency smaller than the throughput for a throughput of 32 cycles and a program which exerts a full load on the memory
6.2. Latency < $1/T_0$

system — i.e. the probability of an instruction being a LOAD is $T_0$. Figure 6.5 shows the same result but normalised to the efficiency found when $L_0 = 1/T_0$.

![Figure 6.4: Processor efficiency for $L_0 < 1/T_0$](image)

The processor efficiency is improved when $L_0 < T_0$ and this is particularly noticeable with just 1 or 2 threads. Considering the result for a latency of 8 cycles and just one thread, it is clear that the thread can be rescheduled 8 cycles after it has been descheduled due to a LOAD. However, 24 cycles must pass before the thread can issue another LOAD without the LOAD needing to be held in the memory queue. The probability that the thread will need to be queued is equal to the probability that there are less than 24 instructions between LOADs (given that the average is 32). The time which that LOAD will need to be queued is 24 cycles less the number of instructions since the last LOAD.

These results show that, with the values chosen, reducing the latency to a quarter of $1/T_0$ can improve efficiency by 50% for a single thread. In short, it is worth ensuring that the critical word is made available to the processor as soon as possible.
6.3. The cost of switching threads

The processor in the previous section only switched threads when it encountered a LOAD that resulted in a cache miss ('switch-on-miss'). If this was implemented on a pipelined processor then it would be necessary to flush any instructions following the LOAD before scheduling the next thread. This is illustrated in table 6.1 for a pipeline with three stages in front of the [data] memory access stage. In cycle 6 thread t1's LOAD is found to cause a cache miss and the t1:nop3, t1:nop4, and t1:nop5 instructions are required to be flushed. Instruction fetching continues from thread t2 in cycle 7.

6.3.1 Switch-on-load

In an attempt to avoid the cost associated with flushing the pipeline in the switch-on-miss strategy, the instruction fetch stage could speculatively start to fetch instructions
6.3. The cost of switching threads

from a different thread whenever it detects a LOAD instruction. Once the LOAD reaches the memory stage of the pipeline it can be determined whether the LOAD results in a cache hit or a cache miss. If the cache is hit then the thread can be returned to the thread queue for rescheduling, and if the cache is missed then a main memory request will be issued.

Table 6.2 is a pipeline diagram that shows the switch-on-load scheme when a LOAD hits the cache. In cycle 3 the fetch stage notes that the last fetched instruction (belonging to thread t1) was a LOAD and switches to a new thread (t2) from the thread queue. It continues fetching from thread t2 until it encounters another LOAD in cycle 5. There are no more threads in the thread queue so the pipeline is fed with NOP 'bubbles' (not belonging to any thread).

In cycle 6, thread t1's LOAD reaches the data memory access pipeline stage (stage 4) and the LOAD is discovered to be a cache hit, so the thread identifier is sent to the thread queue for rescheduling. Thread t1 is then re-scheduled in cycle 7.

Table 6.3 shows what would happen if the LOAD had resulted in a cache miss — when the LOAD reaches the memory access pipe stage it does not get returned to the continuation.
6.3. The cost of switching threads

The cost of switching threads is significant. It involves the overhead of contexts, which can be substantial. The table below illustrates the cost of switching threads in a simplified model.

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Thread Queue</th>
<th>Stage 1</th>
<th>Stage 2</th>
<th>Stage 3</th>
<th>Stage 4 (Data Memory)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>t1, t2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>t1, t2</td>
<td>t1:nop1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>t2</td>
<td>t1:nop2</td>
<td>t1:nop1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>t2</td>
<td>t1:LOAD</td>
<td>t1:nop2</td>
<td>t1:nop1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>t2</td>
<td>t2:LOAD</td>
<td>t1:LOAD</td>
<td>t1:nop2</td>
<td>t1:nop1</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>t2:LOAD</td>
<td>t1:LOAD</td>
<td>t1:nop2</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>t1</td>
<td>t1:nop3</td>
<td></td>
<td></td>
<td>t2:LOAD</td>
</tr>
<tr>
<td>8</td>
<td>t1</td>
<td>t1:nop4</td>
<td>t1:nop3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.2: Switch-on-load: cache hit (t1)

It will be placed in the queue for rescheduling only when the memory access has completed.

Rather than the fetch stage switching threads whenever a LOAD occurs, it may be appropriate to have two forms of LOAD, one of which provides a hint to the fetch stage that this LOAD may result in a cache miss. This would reduce the number of unnecessary context switches, but would require flushing in the case of an unpredicted cache miss. Also, it assumes a reasonable amount of knowledge about the memory access patterns.

In these two examples, the pipeline runs dry from cycle 5 because there are no more threads in the continuation queue. A more sophisticated system might return the thread identifier to the continuation queue once a LOAD has been encountered so that it may be rescheduled should the pipe run dry. If the LOAD turns out to be a cache miss, it would be necessary to flush both the pipe and to remove the thread identifier from the thread queue should it not yet be rescheduled. This is illustrated in table 6.4, but without the need for flushing thread t1 from the thread queue.
### 6.3. The cost of switching threads

<table>
<thead>
<tr>
<th>cycle</th>
<th>thread queue</th>
<th>stage 1</th>
<th>stage 2</th>
<th>stage 3</th>
<th>stage 4 (data memory)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>t1; t2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>t1; t2</td>
<td>t1:nop1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>t2</td>
<td>t1:nop2</td>
<td>t1:nop1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>t2</td>
<td>t1:LOAD</td>
<td>t1:nop2</td>
<td>t1:nop1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>t2</td>
<td>t2:LOAD</td>
<td>t1:LOAD</td>
<td>t1:nop2</td>
<td>t1:nop1</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>t2:LOAD</td>
<td>t1:LOAD</td>
<td>t1:nop2</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>t2:LOAD</td>
<td>t1:LOAD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>—flush—</td>
<td>—flush—</td>
<td>—flush—</td>
<td>t2:LOAD</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>—flush—</td>
</tr>
</tbody>
</table>

Table 6.3: Switch-on-load: cache miss (t1)

<table>
<thead>
<tr>
<th>cycle</th>
<th>thread queue</th>
<th>stage 1</th>
<th>stage 2</th>
<th>stage 3</th>
<th>stage 4 (data memory)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>t1; t2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>t1; t2</td>
<td>t1:nop1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>t2</td>
<td>t1:nop2</td>
<td>t1:nop1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>t2</td>
<td>t1:LOAD</td>
<td>t1:nop2</td>
<td>t1:nop1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>t2; t1</td>
<td>t2:LOAD</td>
<td>t1:LOAD</td>
<td>t1:nop2</td>
<td>t1:nop1</td>
</tr>
<tr>
<td>5</td>
<td>t1</td>
<td>t1:nop3</td>
<td>t2:LOAD</td>
<td>t1:LOAD</td>
<td>t1:nop2</td>
</tr>
<tr>
<td>6</td>
<td>t1</td>
<td>t1:nop4</td>
<td>t1:nop3</td>
<td>t2:LOAD</td>
<td>t1:LOAD</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>—flush—</td>
<td>—flush—</td>
<td>—flush—</td>
<td>—flush—</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>—flush—</td>
<td>—flush—</td>
<td>—flush—</td>
<td>—flush—</td>
</tr>
</tbody>
</table>

Table 6.4: Switch-on-load (optimised, with flush): cache miss (t1)
6.3.2 Switch-every-cycle

A third thread-switching strategy is to switch between threads every cycle regardless of whether the fetched instruction is a LOAD or not. This is the strategy used in the Denelcor HEP [Smith81] and Tera [Bokhari98]. There are two possible variants — the first allows only one instruction from each thread to be in the pipeline at a time, and the second allows more than one instruction from each thread should there be fewer threads available for scheduling than there are pipeline stages. Pipeline diagrams for these two schemes are shown in tables 6.5 and 6.6 respectively.

On the second variant, it is necessary to return the current thread’s identifier to the thread queue once an instruction has been fetched from it. Also, it is necessary to flush any instructions from the thread of a LOAD that causes a cache miss from the pipeline and to remove the thread from the thread queue. This could be awkward to implement — associatively matching and removing an entry from the middle of a queue is not simple. Instead it may be simpler to mark the entry in the queue as invalid and remove it when it reaches the top of the queue. This might mean that a request to the thread queue results in a nullified thread on occasions, but this is only likely when the request was preceded by another request in the previous clock cycle.

6.4 Performance of strategies to reduce the cost of thread switches

The thread-switching strategies described above were simulated on a memory system with $T_0 = 1/L_0$ and a likelihood of an instruction being a LOAD of $T_0$ — i.e. the program was exerting a full load on the memory system. $L_0$ was taken as 16 cycles. For the ‘switch-on-load’ mechanism, the simulation was also run with a 1-in-4 and a 1-in-8 of an instruction be a LOAD that resulted in a cache hit (and hence did not need to be sent to the memory system). This was to highlight the fact that the instruction-fetch stage can not determine whether a LOAD will result in a cache hit or miss.

The processor was modified to have a variable number of pipeline stages before the stage that issues memory requests — these pipeline stages will require flushing in some
6.4. Performance of strategies to reduce the cost of thread switches

<table>
<thead>
<tr>
<th>cycle</th>
<th>thread queue</th>
<th>stage 1</th>
<th>stage 2</th>
<th>stage 3</th>
<th>stage 4 (data memory)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>t1; t2; t3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>t1; t2; t3</td>
<td>t1:nop1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>t2; t3</td>
<td>t2:LOAD</td>
<td>t1:nop1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>t3</td>
<td>t3:nop1</td>
<td>t2:LOAD</td>
<td>t1:nop1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>t3:nop1</td>
<td>t2:LOAD</td>
<td>t1:nop1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>t1</td>
<td>t1:nop2</td>
<td>t3:nop1</td>
<td>t2:LOAD</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>t3</td>
<td>t3:nop2</td>
<td>t1:nop2</td>
<td>t3:nop1</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>t3</td>
<td>t3:nop2</td>
<td></td>
<td>t1:nop2</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>t3</td>
<td>t3:nop2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.5: Switch-every-cycle: simple

<table>
<thead>
<tr>
<th>cycle</th>
<th>thread queue</th>
<th>stage 1</th>
<th>stage 2</th>
<th>stage 3</th>
<th>stage 4 (data memory)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>t1; t2; t3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>t1; t2; t3</td>
<td>t1:nop1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>t2; t3; t1</td>
<td>t2:LOAD</td>
<td>t1:nop1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>t3; t1; t2</td>
<td>t3:nop1</td>
<td>t2:LOAD</td>
<td>t1:nop1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>t1; t2; t3</td>
<td>t1:nop2</td>
<td>t3:nop1</td>
<td>t2:LOAD</td>
<td>t1:nop1</td>
</tr>
<tr>
<td>5</td>
<td>t2; t3; t1</td>
<td>t2:nop1</td>
<td>t1:nop2</td>
<td>t3:nop1</td>
<td>t2:LOAD</td>
</tr>
<tr>
<td>6</td>
<td>t3; t1</td>
<td>t3:nop2</td>
<td></td>
<td>t1:nop2</td>
<td>t3:nop1</td>
</tr>
<tr>
<td>7</td>
<td>t1; t3</td>
<td>t1:nop3</td>
<td>t3:nop2</td>
<td></td>
<td>t1:nop2</td>
</tr>
<tr>
<td>8</td>
<td>t3; t1</td>
<td>t3:nop3</td>
<td>t1:nop3</td>
<td>t3:nop2</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.6: Switch-every-cycle: with flushing
of the thread switching strategies described above. All previous simulations have been with no pipe stages prior to the memory request stage. The processor with 3 leading pipe stages is illustrated in figure 6.6, the memory request stage being labelled 'd.mem'.

The simulator was run with a total of 100,000 instructions shared between 1 and 8 threads. There were 0, 3, or 6 pipeline stages before the memory request stage. Figure 6.7 shows the measured efficiency with three early pipeline stages and figure 6.8 shows the measured efficiency for 6 early pipeline stages.

In both of these graphs the 'maximum achievable' plot refers to the efficiency with no early pipeline stages - in this scenario all five thread switching strategies perform equally. The result for the switch-on-load strategy where the probability of an instruction being a LOAD that hits the cache was 1 in 8, has been omitted from the graphs for clarity. This result was found to be almost identical to the result when the probability of an instruction being a cache-hit LOAD was 1 in 4.

The performance loss is due to the number of empty slots caused by the simplicity of the thread switching strategy or the number of instructions that had to be flushed.

The simple switch-on-load strategy performs badly with a small number of threads due to the large number of false deschedulings. Unnecessarily switching on LOADs that hit the cache will result in the pipeline running dry when there are no free threads to run.

The switch-on-miss strategy is simple to implement, but there is an upper limit on the performance it can reach. This is due to the need to flush instructions when a cache miss is suffered. As the length of the pipe increases, so the upper limit on the processor efficiency decreases.
6.4. Performance of strategies to reduce the cost of thread switches

![Graph showing processor efficiency versus number of threads for different thread switching strategies.](image)

**Figure 6.7: Thread switching strategies: 3 pipe stages before d-mem stage**

The simple switch-every-cycle performs very badly when there are fewer threads than there are pipeline stages — this is because it only allows one instruction from each thread to be in the pipe at a time. Clearly with 2 threads, a 4 stage pipeline would have 2 slots empty at the best of times and more when the number of free threads is reduced by sleeping on a LOAD.

The variants of the switch-every-cycle and switch-on-load strategies that allow more than one instruction from each thread in the queue perform close to the maximum achievable. The pipeline never runs dry since the processor will continue executing the current thread if the thread queue is empty. The performance is better than the switch-on-miss because when a pipeline flush is necessary there are instructions from a mixture of threads in the pipe and only some will need flushing.

Of the thread switching schemes discussed, the simple switch-on-load and switch-every-cycle are relatively simple to implement but their performance is poor. All the others
6.4. Performance of strategies to reduce the cost of thread switches

Figure 6.8: Thread switching strategies: 6 pipe stages before d-mem stage

require the ability to flush the pipeline by a matching of thread identifiers. Switch-on-miss does not require an additional write path to return a thread to the thread queue, but its performance is poorer than the switch-every-cycle and switch-on-load which allow more than one instruction from each thread to be in the pipe at the same time. Switch-on-load requires the instruction fetch stage to be able to decode an instruction sufficiently to be able to tell it is a LOAD. This leaves the switch-every-cycle as the best choice for many applications, being reasonably simple to implement and with good performance.
Chapter 7

A lightweight multithreaded processor

The previous simulations used a much simplified processor that consisted of nothing more than a memory request issuer and a continuation queue. This chapter considers how a low-overhead ‘microthreaded’ RISC processor could be implemented efficiently. The design borrows from that described in [Bolychevsky96]. First a summary of the design is presented, then the practical challenges in an implementation are considered.

7.1 Summary of the design

If an attempt is being made to extract threads from the available instruction-level parallelism then all threads will share the same register bank. Threads can then be identified by just their program counter, with ready-to-run threads being held in a ‘schedulable threads’ queue. No registers need to be saved on a context switch — just the program counter (PC) of the thread to be scheduled needs restoring.

7.1.1 Synchronisation

A simple way to synchronise the flow of data between threads is to tag all registers to indicate whether or not they contain valid data (a register can be invalidated by an
explicit 'invalidate-register' instruction). If an instruction finds that one of its source registers holds invalid data then it is aborted and its PC is stored in the invalid source register. Simultaneously, any instructions from the same thread as the aborting instruction are flushed from the pipeline.

Whenever an instruction needs to write to a destination register, the destination register is checked to see if it holds the PC of a previously aborted instruction. If so, that PC is placed in the thread queue to await rescheduling and the valid data is written to the register. When the PC is later fetched from the PC, the previously aborted instruction will now find valid data in its source register and be able to proceed. By invalidating a register before creating a pair of threads, data may be communicated through that register — the consuming thread will sleep if it attempts to read the register before the other thread has written valid data to it.

7.1.2 Long latency memory accesses

If a LOAD instruction encounters a cache miss, then the LOAD is converted to an 'invalidate destination register' instruction. Any subsequent instructions that attempt to use the destination register will find it to be invalid and abort, writing its PC to the register. Once the memory system has returned the data, it needs to be written to the destination register, pushing any PC that may be held there onto the continuation queue. The mechanism for doing this will be discussed shortly.

7.1.3 Thread creation and descheduling

[Bolychevsky96] suggests that all instructions should be marked for horizontal and/or vertical transfer. If an instruction is marked for horizontal transfer then the instruction-fetch stage of the pipeline will continue to fetch instructions sequentially (i.e. from the same thread). If the instruction is not marked for horizontal transfer then a new PC will be taken from the continuation queue and instruction fetching will continue from that thread — this can be used if the compiler suspects than an instruction will have a long latency and that scheduling another thread would be worthwhile.
If an instruction is marked for vertical transfer then, after completion, the PC of the next sequential instruction is returned to the continuation queue. If a thread is marked for horizontal and vertical transfer then a new thread is created — i.e. a ‘fork’ is made. Marking an instruction for neither horizontal nor vertical transfer will result in its death.

7.2 Implementation considerations

7.2.1 Invalid source operands

Whenever an instruction's source register is found to be invalid, the instruction should be converted to a 'move PC to invalid source register' instruction. Also, a thread flush signal should be sent to all previous pipeline stages to nullify any later instructions from the same thread as this instruction. Since the source operands are checking in the register-read stage of the pipeline, the flush signal need only be sent to the instruction-fetch stage. The instruction-fetch stage should take this flush order as an indication to get a new PC from the thread queue and start fetching instructions from there.

An invalid source register may or may not hold the PC of another instruction that has aborted due to this register being invalid. Whether this is likely will depend on the code generation strategy used. If the register does hold the PC of another instruction, then clearly this instruction's PC cannot be written to the register. The simplest solution would be to disallow two threads to read the same invalid register, but some safeguard should be put in place to prevent threads being lost by their register-held PCs being overwritten. The simplest safeguard to implement would appear to be to treat this instruction as any other register-writing instruction - i.e. when it finds that its destination register [which is the invalid source register] contains a PC, it sends the PC to the thread queue, and writes its own PC to the invalid source register. However, it should not mark the register as holding valid data. When the previously sleeping instruction gets rescheduled, it may still find the source register to be invalid. In this case, it will evict the PC in the source register and store its own PC in it, and this
cycle continues until the register is finally validated. This busy-waiting is unfortunate, but can be avoided with sensible code generation.

Any instruction that writes to a register needs to return any PC that might be in its destination register to the thread queue. So, all instructions must check their destination registers. There are three possible methods for doing this:

**Treating destination register as a source register**

The destination register would be read in the register-read stage, with forwarding from later stages for results from earlier instructions that write to the same register. The PC can then be sent to the thread queue by the register-read pipeline stage. The same connection from register-read to the thread queue could be used to write new PCs into the thread queue when a fork instruction is encountered. This method requires an extra read path to the register set, and more forwarding paths.

**Reading destination register in the register-write stage**

Here the destination register would be read as it passed from the data-memory pipeline stage into the register-write stage, so that its content is known before being written in the register-write stage. A path from register-write to the thread queue would be needed, and there would be the disadvantage that the sleeping instruction [whose PC is held in the destination register] is not returned to the thread until a few cycles later than it would be in the above method. Also, if the same thread-queue-write channel was used for forking then, similarly, the new thread would not be available until a few cycles later than the above method. Note that this method requires an extra read path to the register set.

**The register set checks destination register's contents**

In this method, the pipeline would do no checking of the destination register's contents. Instead, the register set would read the register contents as it writes a value to it and
7.2. Implementation considerations

sends any PC it may contain to the thread queue through its own direct link. This would require a further read port on the register bank.

This solution would allow data being returned by the memory to be written directly to the register set without having to insert a bubble into the pipeline. Care should be taken to avoid deadlock, however — if data is written directly to the register set by the memory system, any instructions in the pipeline will be unaware of this and may abort, writing their PC to a register that is simultaneously being validated directly by the memory system. This can be cured quite simply in the register set - if a ‘write PC’ is made to a register that is not invalid then, rather than writing the PC to the register, the register should send the PC straight back to the queue of threads awaiting scheduling. This might mean that the instruction was unnecessarily aborted when another scheme may have allowed it to stall in the register-read stage, but the occurrence of such a situation will be rare.

7.2.2 Returning data from the memory to the register set

Two key implementation options are available: writing returned data directly to the register, and inserting a pseudo-instruction into the pipeline to write a constant (the returned data) into the LOAD’s destination register.

The second option seems simple, but costs one cycle. Also, there could be a risk of deadlock. Suppose the memory system queue is full so that no new memory requests can be accepted into the system. The processor could be stalled waiting for its LOAD instruction to be permitted to proceed from the data-memory stage to the memory system. In this case, it may be possible to insert the pseudo-instruction into the pipeline, but there is no guarantee that it will be able to progress. The whole memory system could be stalled until the data is returned to the destination register thereby freeing up a slot in the memory system to accept the new LOAD. Careful design should be able to avoid this — possibly by ensuring that there is always one free space in the memory request queue.
7.2. Implementation considerations

7.2.3 Branches

Delayed branches

All branches in the SPARC RISC processor have one delay slot — i.e. one instruction after the branch is always executed, whether the branch is taken or not. Furthermore, branch instructions have an annul option, where the delay slot instruction is only executed if the branch is taken - this is useful for loops where the instruction at the top of the loop can be moved into the delay slot, and the branch annul bit set.

There is a significant problem with delay-slot instructions in a microthreaded processor. Suppose a branch is taken and one of the source registers for the delay slot instruction is invalid. The PC of the delay slot will be written to the invalid source register and the thread will sleep. When the register is validated the delay-slot instruction's PC will be placed in the thread queue, and the delay slot instruction will be re-tried in a later cycle. However, the pipeline will have lost knowledge of the taken branch instruction, and will instead execute sequentially from the delay slot instruction. Avoiding this will require more state to be stored on a context switch, or the banning of the use of troublesome instructions in delay slots.

Branch prediction

The branch destination for conditional branches can not be determined until the ALU stage where condition code flags are held. This means that at least one or more instructions must be fetched before the branch destination is determined.

One solution is for the instruction fetch stage to make a taken/not taken prediction and to fetch accordingly. The prediction made is sent along the pipeline with the branch instruction for the ALU to compare with the real result. If it notices a misprediction then it sends a mispredict flush to the earlier stages which will nullify their current instructions. The instruction-fetch stage will then start fetching from the correct address. Note that, unlike a thread flush, the instruction-fetch stage does not start fetching from a new thread.
7.2. Implementation considerations

Another solution is for the IF stage to switch threads when it encounters a branch and tag the branch instruction so that the ALU sends the branch destination address to the thread queue to be rescheduled later (executing from the branch destination).

7.2.4 Thread management

Marking every instruction with tags to indicate 'horizontal' or 'vertical' transfer requires that there be two spare bits in every instruction bit-pattern, or that the instruction-word size is increased by two bits. The former is unlikely and the latter is usually impractical, so a simpler solution is to have specific instructions to manage threads.

Forking — creating threads

In order to create a fork in the program flow, the address of the two following threads must be known. The most common solution is to specify the address of just one thread and for the other to continue sequentially from the FORK instruction. The fork instruction encoding can be similar to a branch in format. The address held in the instruction's destination field should be sent to the thread queue, and the instruction-fetch pipeline stage should continue fetching sequentially from the same thread.

Conceivably, forks could be dealt with in the instruction-fetch stage and removed from the instruction stream. This requires that fork destinations are given by immediate operands, though this is the most likely scenario anyway. Fork destinations could be given as an absolute value or as an offset. Although using an offset is flexible, it requires an extra adder if the calculation is not done in the ALU.

Ending a thread

A special instruction is needed to terminate a thread. When spotted by the instruction fetch stage, it should stop fetching instructions from this thread and fetch a new PC from the thread queue. Like the fork instruction, this could be removed from the instruction stream by the fetch stage.
Vertical transfer, and not horizontal transfer

Marking an instruction for vertical transfer and not horizontal transfer provided a hint to the fetch stage that an instruction was likely to have a long latency and that a new thread should be scheduled. The 'switch-on-load' thread switching strategy, described earlier, is a reduced form of this, and could be enabled only for specially tagged loads.

7.2.5 Condition codes

It was stated earlier that the PC was the only state that needs to be saved when descheduling a thread. Some architectures, such as the SPARC, use condition flags to hold the result of compare and ALU operations. With these architectures, if a thread is descheduled between an instruction that sets the condition flags and an instruction that uses the result held in those flags then it is essential that the flags are saved or restored when a thread is descheduled or rescheduled. This can be achieved by saving the condition codes in the destination register along with the program counter when a thread is put to sleep. This may require reducing the size of program counter that can be stored or increasing the width of the register (though the additional width might be available only to the thread-management mechanism).

The condition codes would be passed to the thread queue along with an evicted PC when valid data is written into the register. Then, when a new thread is taken from the thread queue by the fetch stage, a 'new thread' flag is passed down the pipeline, along with the saved condition code flags. When the 'new thread' instruction reaches the ALU, the flags are copied into the ALU's real flags.

Another option is for the ALU to have a bank of condition code flags, indexed by the thread's i.d. number.

Saving flags may seem to be approaching the heavy-duty context switching that our low-overhead threading is supposed to avoid, but realistically there are only a few flags (four on the SPARC) and these could easily be held in a register along with the PC if the PC's length was restricted. On the SPARC, all 32-bit word addresses have their lower two bits set to zero, so these bits could be used to hold two of the four condition code
7.2. Implementation considerations

flags, meaning that the address size need only be reduced by two bits to accommodate the remainder of the flags.

Not all architectures use condition code flags, an example being the MIPS architecture which uses a standard register to hold the result of any comparison operations. MIPS conditional branch instructions refer to a register which holds the result of the comparison. As a result, the MIPS has the ability to hold as many comparison results as there are registers, whereas the SPARC can only hold one comparison result at any time. In this case, the multithreaded architecture can save just the PC when a thread is descheduled, leaving it to the programmer to ensure that different threads use different registers to hold comparison results.

7.2.6 Traps and interrupts

A problem may arise when there are a number of memory requests outstanding when a trap or interrupt occurs. This means that the trap handling routine will find some registers contain PCs, and some contain invalid data. The contents of the register set can not simply be copied to memory because an outstanding memory request may be returned at any time and will not be able to wake up any threads whose program counters have been copied from a register into memory.

One solution would be to have a hardware counter that keeps track of the number of outstanding memory requests and only allows the trap handler to start when there are no outstanding memory requests for that thread. This slows down the trap response a little, but it should be no worse than in any other non-microthreaded processor. This method would not be appropriate for situations where an attempted memory access caused the trap, such as in a virtual memory page fault. The best solution may be to distinguish between traps that are likely to need to have registers saved to memory (e.g. for a context switch) and traps that don't need to have registers saved to memory, but could cope with an alternate set of registers paged-in especially for trap handling routines. Only with the first sort of trap would the trap handler need to wait for outstanding memory requests to be fulfilled.
7.3 Summary

This chapter has considered the implementation of a dynamically-scheduled RISC processor similar to that in [Bolychevsky96]. A number of simplifications have been considered that reduce the number of options available to the programmer but significantly reduce the hardware requirements by eliminating the need for horizontal- and vertical-transfer tags for instructions. The next chapter looks at a simulation of a processor with a design based on the considerations in this chapter.
Chapter 8

A lightweight multithreaded processor: simulation

A simulator has been implemented for a multithreaded processor based on the design considerations in the previous chapter. The simulated processor is a further developed variant of that presented in [Gale00] and is based on a generic RISC machine, though it can execute many of the instructions in the SPARC instruction set. Figure 8.1 shows the simulated processor.

The processor has a single-issue pipeline with no cache — LOADS are marked as cache hits or misses in the instruction stream. Access to memory is through the use of explicit LOAD and STORE instructions that transfer data between registers and memory.

Static branch prediction is used with the assumption that branches are usually taken.

In order to identify threads within microthreaded code, it has been necessary to introduce a FORK instruction into the instruction set. Upon execution, this creates a new thread whose program counter is held in the continuation queue to await scheduling.

To synchronise parallel threads an ‘invalidate register’ instruction is included so that subsequent threads may use that register for synchronisation.

Whenever an instruction finds invalid data in its destination register, any instructions from the same thread that exist in earlier pipe stages are flushed and a new thread scheduled.
In order to simplify the simulation a number of assumptions have been made. Firstly, we assume that there are an infinite number of registers available. Each thread requires a certain number of registers, and clearly the maximum number of threads that is practical is tightly linked to the size of the register set. Allowing the size of the register set to limit the number of threads at this stage would not allow us to observe the direct benefit of microthreading, especially considering that the exact number of registers needed for a thread can vary depending on the register allocation strategy used.

Secondly, we assume that the memory system can handle an infinite number of memory requests at any one time. This is unrealistic but the limit is one that can be addressed.
in the design of the interconnect. As with the unrestricted number of registers, we do not want to restrict the performance gains obtainable by threading yet.

Thirdly, we assume a Harvard architecture where data and instructions are stored in physically separate memories, allowing us to assume that the pipeline is never starved of instructions due to the instruction memory’s behaviour. Alternatively, instruction caching could achieve the same goal.

8.1 Implementing the simulator - software issues

The simulator has been implemented using the Occam programming language, with a similar strategy to simulators presented previously. Each pipeline stage is a separate parallel processes communicating via synchronised channels. A communication takes place on each channel on each simulation cycle — this eliminates the possibility of deadlock.

The thread queue (‘continuation queue’), the register set, and the data memory are each implemented as separate processes, and there is a synchroniser process which corresponds to the processor’s central clock. On each clock tick, the synchroniser receives state information from the pipeline stages and delivers this as the simulator’s output. The synchroniser also has the power to kill every process it controls by sending a ‘die’ clock tick.

The simulator has five stages: instruction-fetch-and-decode, register-read, arithmetic-logic unit, data memory access, and register write-back. There are forwarding paths from all subsequent stages to the register-read stage to prevent read-after-write hazards. Instructions are made available to the instruction-fetch stage as pre-decoded RISC instructions, and a utility has been written to decode SPARC instructions from an assembler output.

The interface to the memory system (not shown) is the same as for the simple processor, so that the two processors can be readily interchanged.

Whenever an instruction’s source register is found to be invalid, the instruction is converted to a ‘store PC in the invalid source operand’ instruction. Also, a thread
flush is passed to the instruction-fetch stage to nullify any later instructions from the same thread and to indicate that a new thread needs to be fetched from the continuation queue.

Data returned by memory is sent straight to the register set, with the register set having the responsibility of sending any PCs held in registers to the thread queue.

8.2 Thread management

We now consider the behaviour of some small examples when run on the simulated processor.

8.3 Data dependency on LOAD result

Consider the following fragment of code:

0: ld r1, [99] ; r1 := mem[99]
1: add r2, r1, r2 ; r2 := r1 + r2
2: nop
3: nop

There is a dependency (through r1) between the ld (LOAD) and add instructions. If we had a memory latency of 8 cycles then a conventional processor would stall the add in the register read stage for 8 cycles. The behaviour of the simulated processor is shown in table 8.1.

In cycle 3 the add instruction reaches the register read (r.r.) stage and the pipeline interlock mechanism holds it here for an extra cycle because the result of the ld instruction is not clear until it reaches the memory stage (d.mem) in cycle 4. The ld is found to cause a cache miss so it converts to an 'invalidate destination register' instruction. This is forwarded to the interlocked add instruction which observes that one of its source operands is invalid and converts to a 'store PC' instruction. All later instructions from the same thread are flushed from the pipe.
8.3. Data dependency on LOAD result

When the data is returned by the memory it is written into register r1 and the PC (of the add) that is stored is pushed into the thread queue to await rescheduling. In the next cycle (14) the thread is rescheduled. This time the add finds valid data in r1 and completes normally.

<table>
<thead>
<tr>
<th>cycle</th>
<th>thread queue</th>
<th>i.f.</th>
<th>r.r.</th>
<th>a.l.u.</th>
<th>d.mem</th>
<th>w.b.</th>
<th>r1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>ld r1,[99]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>add r2,r1,r2</td>
<td>ld r1,[99]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>nop</td>
<td>add r2,r1,r2</td>
<td>ld r1,[99]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>nop</td>
<td>add r2,r1,r2</td>
<td>interlock</td>
<td>ld r1,[99]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>—flush—</td>
<td>—flush—</td>
<td>mov r1, pc</td>
<td>interlock</td>
<td>inval r1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>—flush—</td>
<td>—flush—</td>
<td>mov r1, pc</td>
<td>interlock</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>—flush—</td>
<td>—flush—</td>
<td>mov r1, pc</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td>—flush—</td>
<td>—flush—</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>add r2,r1,r2</td>
<td></td>
<td>data</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td>nop</td>
<td>add r2,r1,r2</td>
<td></td>
<td>data</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 8.1: Split-phase load

8.3.1 Thread creation

A thread is created by giving the address of the two threads, as shown in this fragment of code:

0:    fork 1, 7
1:    nop
2:    nop
3:    nop
4:    end
      cont...
8.3. Data dependency on LOAD result

7:  nop
8:  nop
9:  nop

The 1 of the fork 1,7 is implicit - it is the address following the fork. The behaviour of this code as it passes through the pipe is shown in Table 8.2. The address of the new thread is placed into the thread queue once the fork has passed through the arithmetic-logic unit (a.l.u.) stage. In cycle 5 the first thread’s end instruction is reached, terminating this thread. The instruction-fetch stage (i.f.) observes this and begins fetching the thread from the top of the thread queue.

<table>
<thead>
<tr>
<th>cycle</th>
<th>thread queue</th>
<th>i.f.</th>
<th>r.r.</th>
<th>a.l.u.</th>
<th>d.mem</th>
<th>w.b</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0:fork 1,7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1:nop</td>
<td>fork 1,7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2:nop</td>
<td>nop</td>
<td>fork 1,7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3:nop</td>
<td>nop</td>
<td>nop</td>
<td>fork 1,7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>4:and</td>
<td>nop</td>
<td>nop</td>
<td>nop</td>
<td>fork 1,7</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5:nop</td>
<td>end</td>
<td>nop</td>
<td>nop</td>
<td>nop</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>6:nop</td>
<td>nop</td>
<td>end</td>
<td>nop</td>
<td>nop</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>7:nop</td>
<td>nop</td>
<td>end</td>
<td>nop</td>
<td>nop</td>
<td></td>
</tr>
</tbody>
</table>

Table 8.2: Forking

8.3.2 Synchronising threads

Threads can be synchronised by invalidating a register before forking and using that register to pass data between the threads. This data would ideally be useful data but could be null data used solely to force a synchronisation:
8.3. Data dependency on LOAD result

0: inval r2 ; r2 := invalid
1: fork 2, 7
2: mov r1, r2 ; r1 := r2
3: nop
4: nop
7: mov r2, 99 ; r2 := 99
8: end

The pipeline behaviour is shown in 8.3. In cycle 4, the mov r1, r2 instruction’s attempt to read r2 is aborted because r2 contains invalid data. The thread is flushed and the instruction converts to a mov r2, PC instruction. The first thread is now sleeping on r2. Valid data is written to r2 when the other thread’s mov r2, 99 instruction reaches the register write-back (w.b.) stage in cycle 9. The PC held there (2) is pushed out into the thread queue, and the now-awoken thread gets rescheduled in cycle 10.

<table>
<thead>
<tr>
<th>cycle</th>
<th>thread queue</th>
<th>i.f.</th>
<th>r.r.</th>
<th>a.l.u.</th>
<th>d.mem</th>
<th>w.b.</th>
<th>r2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>inval</td>
<td>r2</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>fork 2, 7</td>
<td>inval</td>
<td>r2</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>mov r1, r2</td>
<td>fork 2, 7</td>
<td>inval</td>
<td>r2</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>flush</td>
<td>mov r2,pc</td>
<td>fork 2, 7</td>
<td>inval</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>mov r2,99</td>
<td>flush</td>
<td>mov r2,pc</td>
<td>fork 2, 7</td>
<td>inval</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>end</td>
<td>mov r2,99</td>
<td>flush</td>
<td>mov r2,pc</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>end</td>
<td>mov r2,99</td>
<td>flush</td>
<td>mov r2,pc</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
<td>end</td>
<td>mov r2,99</td>
<td>flush</td>
<td>mov r2,pc</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>10</td>
<td>2</td>
<td>end</td>
<td>mov r2,99</td>
<td>flush</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>2</td>
<td>end</td>
<td>mov r2,99</td>
<td>flush</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>mov r1, r2</td>
<td>end</td>
<td>99</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td>nop</td>
<td>mov r1, r2</td>
<td></td>
<td></td>
<td></td>
<td>99</td>
</tr>
</tbody>
</table>

Table 8.3: Synchronising two threads
8.3.3 Two threads awaiting the same result

A problem occurs if two threads await the data of the same register, as shown here:

0: inval r1 ; r1 := invalid
1: fork 2, 7
2: add r7, r1, r2 ; r7 := r1 + r2
3: nop
4: nop
5: nop
6: nop
7: add r8, r1, r2 ; r8 := r1 + r2
8: nop
9: nop

The invalid register, r1, can only hold one PC and there are two threads that wish to sleep on it. The end result is that one thread sleeps but is awoken by the other thread writing its PC to r1. This continues as shown in table 8.4. Of course, this is not a very sensible piece of code, but a real example might be when r1's invalidation is the result of a load suffering a cache miss.

8.3.4 Two threads waiting on two shared results

This code is similar to the last piece of code, but with a key difference:

0: inval r1 ; r1 := invalid
1: inval r2 ; r2 := invalid
2: fork 3, 7
3: add r7, r1, r2 ; r7 := r1 + r2
4: nop
5: end cont...
8.3. Data dependency on LOAD result

<table>
<thead>
<tr>
<th>cycle</th>
<th>thread queue</th>
<th>i.f.</th>
<th>r.r.</th>
<th>a.l.u.</th>
<th>d.mem</th>
<th>w.b.</th>
<th>r1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0: inval r1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>1: fork 2,7</td>
<td>inval r1</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>2: add r7, r1, r2</td>
<td>fork 2,7</td>
<td>inval r1</td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>3: flush</td>
<td>mov r1, pc</td>
<td>fork 2,7</td>
<td>inval r1</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>7: add r8, r1, r2</td>
<td>flush</td>
<td>mov r1, pc</td>
<td>fork 2,7</td>
<td>inval r1</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>8: flush</td>
<td>mov r1, pc</td>
<td>flush</td>
<td>mov r1, pc</td>
<td>fork 2,7</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>2: add r7, r1, r2</td>
<td>flush</td>
<td>mov r1, pc</td>
<td></td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>3: flush</td>
<td>mov r1, pc</td>
<td></td>
<td></td>
<td></td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td>7</td>
<td>7: add r8, r1, r2</td>
<td>flush</td>
<td>mov r1, pc</td>
<td></td>
<td></td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>8: flush</td>
<td>mov r1, pc</td>
<td></td>
<td></td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>17</td>
</tr>
</tbody>
</table>

Table 8.4: Two threads awaiting the same result

7: add r8, r1, r2 ; r8 := r1 + r2
8: nop
9: nop

The difference is that there are two invalid source registers to the add instructions, and either can be chosen to hold the PCs of both threads — the choice is arbitrary, but if one contains the PC of a sleeping thread then the other invalid source register is chosen. This is illustrated in table 8.5.
8.4. Performance

The simulator was run with an instruction stream consisting of a random assortment of LOADS and NOPs. The probability of a LOAD was such that 1 in 32 instructions was a LOAD. The latency of the queued memory system attached (the same system as used in previous experiments) was 32 cycles, and the memory throughput was 1 request per 32 cycles. This means that the application exerted a full load on the memory system.

Unlike the earlier experiments on a simpler processor model, a LOAD instruction did not result in a thread being put to sleep. Instead, a LOAD was followed by a number of NOP instructions and then an instruction that was dependent on the result of the LOAD. The number of NOPs was randomly chosen with a uniform distribution between 0 and some maximum.

The performance results are shown in figure 8.2. The performance for the simpler model (used in earlier chapters) with a switch-on-miss strategy and the same number of
pipeline stages is shown for comparison. It can be seen that the performance is better than with the simple model. This is partly due to the non-dependent instructions following the LOAD, and the processor efficiency should increase as the [maximum] number of dependent instructions following a LOAD increases. The more non-dependent instructions that follow a LOAD, the fewer instructions that will need flushing from the pipeline.

Performance is also enhanced due to the fact that it is the register-read pipeline stage that detects that an instruction's source operand is invalid and that the current thread needs to be flushed from the pipe. Since the register-read stage is early in the pipeline, fewer pipeline stages need flushing than with the earlier results, where the need to flush was detected later in the pipeline (in the memory-access stage).

The performance when running a non-random application will depend on the static scheduling of non-dependent instructions subsequent to a LOAD.

![Figure 8.2: Performance of simulated multithreaded processor](image.png)
Chapter 9

Multiprocessor efficiency

The efficiency of processors in multiprocessor systems is affected greatly by the long latencies associated with accessing data held at remote nodes across a network. For a synchronous, symmetrical network in which a message makes a journey between adjacent nodes in one network cycle, [Bolychevsky96] shows that a shared memory multiprocessor can achieve a processor efficiency of 80% with just two or three threads per router channel.

In order to experimentally verify the results given in [Bolychevsky96] for a networked memory system, a simulation of a simple multiprocessor system was implemented. The multiprocessor that was simulated consisted of \( n \) nodes connected in a bi-directional ring structure, illustrated in figure 9.1. Each node consists of a processor and a router with incoming and outgoing connections to both of its neighbours. The simulator is described in more detail in the following sections.

9.1 Network type

The network is a \( n \) node bi-directional ring which runs synchronously and can carry a complete packet in one flit. A flit takes one cycle to be communicated from one node to an adjacent node. Every node holds a processor which owns a fraction of the memory of the whole shared memory.
The maximum distance between any two nodes, assuming a direct route is taken, is \( n/2 \) — since packets can travel in either direction around the ring, a packet will never need to travel more than half the circumference. This distance is the 'diameter', \( D \), of the network. The average number of hops made by a packet is \( D/2 \). A memory request by a processor attached to a node will require two communications, one for the read or write request to a remote node, and one to return the data or acknowledge the write. The average total distance travelled to satisfy a memory request will then be \( D \).

9.2 Router

The router has been designed to scale to a 2-d or 3-d torus and uses a fixed routing strategy whereby each node routes packets in the x-dimension first until the column of the network that holds the destination node is reached, and only then does it start moving the packet in the y-dimension, and then the z-dimension for a 3d torus. This is commonly known as 'first x, then y' or 'dimension-order' routing. For the ring, which is a 1-d torus, routing is required in only one dimension. A received packet will always be routed to the channel which will result in the shortest journey to the packet's destination, even if that route is congested.

The router has two outgoing links, one to each of its neighbouring nodes, as illustrated in figure 9.2. For each outgoing link there are two queues - one to hold packets received on incoming links that need further routing, and one to hold packets newly injected.
by the CPU attached to this router. These are the 'transit' and 'injected' queues, respectively. Since a packet always travels in the same direction from its injection until its consumption, the only time the transit queue for an output grows is when a packet reaches its destination node and is returned in the opposite direction from which it came. If a packet in transit is received on the opposing channel then there will be competition for the output channel and a request will need queueing.

![Router Diagram]

Figure 9.2: Router

When ready to transmit a packet, the outgoing link will always select a packet from the transit queue in preference to the injected queue. The injected queue can only be emptied when the transit queue is empty. Since the average number of hops made by a packet is $D$, an average of $1/D$ packets is consumed at each node. This means that the average throughput available to packets in the injection queue is also $1/D$. Since there are two channels, the throughput available to the CPU connected to a node is $2/D$.

As well as the transit and injected queues, each router also has a 'to-CPU' queue which holds packets whose destination is that router. One or two received packets could be destined for a given node on each network cycle, but only one packet can actually be
passed to the attached CPU/memory-system during that cycle. The to-CPU queue buffers excess packets.

Packets are received from the CPU at a rate of two packets per simulated clock cycle — accepting two rather than one simplifies the simulation by removing the need for a queue before the router. Many of the packets received will be void, however, and the average number of packets received per cycle will be less than unity in a sensible system (since the available throughput for new packets is only $2/D$).

There are no restrictions on the maximum size of any of the router’s queues. Although this is unrealistic, it simplifies the implementation a little, and the results in [Bolychevsky96] are based on a model where queue sizes are unrestricted. Limits on queue size could be imposed later, however.

9.3 Clocking

Although the whole simulation is synchronous, the processor and router can be clocked at different frequencies. This is achieved by having the simulation run at the higher [CPU] frequency and restricting the routers to communicating on just one in every $m$ simulation cycles, where $m$ is the clock division ratio.

Although the router can communicate only once in every $m$ simulation cycles, it can receive packets from the CPU and feed the CPU with packets from its to-CPU queue every simulation cycle.

9.4 Nodes

A number of different types of node are available for connection to the network. For test purposes, idle nodes can be connected with one node being able to inject a packet at a specified time. This allows the routing strategy to be observed and checked.

For running a simulation, there are two types of node in use. The multithreaded processor is identical to that used in the uniprocessor simulation, namely it features a
simple microthreaded processor alongside the memory system. Only one node in the
network is of this type, the node whose performance we are interested in measuring.

All other nodes are idle nodes. They accept packets from the network and consume
them but do not generate any packets. The random traffic in the network for the
simulation is generated by the router.

9.5 Uniform random traffic

To compare the results with [Bolychevsky96] the traffic in the network must be uni­
form random traffic. This is achieved by each router creating a newly injected packet
whenever it finds an outgoing link’s transit queue to be empty. The packet makes a
single journey of a randomly chosen distance, the distances being uniformly distributed
between 1 and $2D$.

9.6 Simulator implementation

The same approach as for the uniprocessor simulation was taken — Occam processes
synchronise every simulation cycle through a hierarchy of fanout clocks which stem from
a central clock process. Diagnostic information is passed along the channels connecting
the simulation processes to the top clock process.

A communication is made on every Occam channel in every simulation cycle - this is
to simplify the design and eliminate the risk of deadlock, at the cost of performance.

9.7 Experimental parameters

The simulator was run until the microthreaded processor had executed 100,000 instruc­
tions divided into between 1 and 10 threads. This meant that there were between 0.5
and 5 threads per outgoing router channel (per node). As with earlier simulations,
the probability that an instruction was a LOAD was chosen so as to exert a load of
one-half-, one-, and two-times the maximum available throughput. For this network
the available throughput to a CPU is $\frac{2}{D}$. The network size was 16 nodes, giving an available throughput of $\frac{1}{4}$. The clock division ratio between the CPU and the router was 1.

9.8 Results

The results are shown in figure 9.3. Also shown on the same graph are the expected values from [Bolychevsky96]. The measured and expected results vary greatly with a small number of threads, the efficiency being lower than expected. This is in some part due to the lack of available instructions to cover latency — for the three levels of load, the probability that an instruction is a LOAD is one in eight, four, or two for the half, full, and double load respectively. In the doubly loaded scenario, there is only one non-LOAD instruction to cover the latency of the LOAD. Unless the latency of the LOAD is small then this is never going to provide good latency tolerance. Even if there was no waiting in any queues, the average time taken for a packet to travel through the network would be $D$ cycles — 16 in this case. When there is just one thread the average execution time of an instruction would be $1 + (1/2)D$, giving a maximum efficiency of $1/9$ when $D = 16$. 

9.8. Results

Figure 9.3: Processor efficiency in a multiprocessor
Chapter 10

Conclusions

The research presented in this thesis has investigated the effect of the latency of a queued memory system on the efficiency of multithreaded processors.

A simulator for a simple multithreaded processor was implemented and it was discovered that just three threads were sufficient to achieve near-optimal performance when the desired memory throughput was half, equal to, or double the maximum physical throughput. This number of threads is slightly lower than that predicted in [Bolychevsky96]. This was discovered to be because the average time that memory requests spent queueing for access to memory was lower than their model expected, due to the number of outstanding memory request being limited to one per thread. Furthermore, it was found that, contrary to the results in [Bolychevsky96], the actual value of the latency of the memory system did affect the overall processor efficiency, though the queueing behaviour was still the dominant factor.

The situation where the memory throughput was not the inverse of the latency of the physical memory was then considered. It was found that increasing the latency by a few cycles to model the cost of implementing multithreading in a processor had a small but noticeable effect on performance. However, the effect was sufficiently small with three or more threads to allow a relatively large number of cycles to be used to make a thread available for rescheduling. This knowledge might ease the implementation of multithreading on a processor.
Reducing the latency to less than the inverse of the throughput was found to give a significant increase in processor performance when just a few threads were available. Thus, by careful design of the cache line fill mechanism to ensure that the data requested by a LOAD instruction is returned before the remainder of the line, multithreaded performance can be improved when there is a temporary narrowing of available parallelism.

A number of strategies were considered to determine when to switch between threads. Because the cost of flushing the pipeline was high with some strategies, it was found that executing just one instruction from a thread before switching to another thread provided the best performance.

Having found that just a few threads were necessary to achieve near-optimal performance, the design of a lightweight multithreaded processor was then explored. Several implementation choices and their costs were considered. A lightweight multithreaded RISC processor that used its register set to synchronise threads was then simulated. The simulated processor was found to tolerate latency effectively with just a few threads, despite a number of inefficiencies in the implementation.

A simulator of a shared memory multiprocessor was implemented to determine the ability of multithreading to hide the large latencies resulting from queueing in the network. It was found that a small number of threads were required per router channel to reduce the effect of latency.

**Summary**

This research has shown that a small number of threads can effectively provide tolerance of memory latency. Given the small number of threads required, a multithreaded processor that can exploit threads derived from an application's instruction-level parallelism has been designed and simulated. The results were encouraging and suggested that adding multithreading capability to a processor can cost little in hardware and provide significant performance improvements.
Bibliography


[Bolychevsky96] Dynamic scheduling in RISC architectures, A. Bolychevsky, C.R.


Bibliography


[Hockney88] Parallel Computers 2, R.W. Hockney & C.R. Jesshope, The Institute of Physics, 1988


[Intel01] Introduction to Hyper-Threading Technology, Intel Corporation, document number 250008-002, 2001


Bibliography


