A single-chip multiprocessor architecture with hardware thread support

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy in the
Faculty of Science and Engineering

January 2001

Gregory M. Wright
Department of Computer Science

Contents

List of Figures

Abstract

Multithreading and single-chip multiprocessing are two promising approaches for future microprocessor design. This thesis introduces an architecture which combines them with several novel features to support fine-grained dynamic parallelism. Hardware support is provided for locating idle processors and forking threads to them. This allows finer grained tasks to be used than is possible with a conventional software run queue, and it is shown how these features can be used by high-level Java code. Heap-allocated register windows allow fast procedure calling and flexible sharing of the register file between threads.

To connect the processors, a pipelined split-transaction cache coherence protocol has been developed. A simple enhancement to the usual load-locked/store-conditional mechanism allows efficient spinlocks in a multithreaded environment.

A possible implementation of the architecture is studied through simulation, using several simple benchmarks and two multimedia Java programs.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.

Copyright

Acknowledgements

Thanks are due to Ian Watson and Ahmed El-Mahdy, for many helpful discussions about this system, and for producing parts of the runtime software described in Chapter 8. The results presented here would not have been possible without their work. Also to Ian, for supervising me over the past three years, and encouraging me to think about it/get on with it, as appropriate; to Mashhuda, Nadia, Ahmed and Ian R., for keeping me sane in the department with lunches and coffee breaks; and to James, Nilton, Se Won, Dmitry, Richard and the Amulet badminton people for taking me away from computers completely. Finally, but certainly not least, to Miranda – I hope it was worth it in the end.

CHAPTER
1. Introduction

This thesis introduces the Jamaica single-chip multithreaded multiprocessor architecture, which is designed for a Java-based desktop personal computer: its most novel feature is hardware support for load balancing and thread distribution. In this chapter, the motivation for such a system is discussed, and some of its features are introduced.

1.1 Motivation

Computer users have always demanded better performance from their machines, and there is no sign that this trend will stop as the possibilities of graphical user interfaces and multimedia applications are explored. To meet the demand, designers have relied on better circuits (more integration, faster logic) and parallel execution (e.g. superscalar processing). Unfortunately, current architectures are approaching the limits of known technology in both respects.

1.1.1 Superscalar architectures

The first problem – clock rate – is related to wire delays inside chips. The shrinking scale of CMOS ICs has allowed more logic to be packed into the available chip area, and also for it to run faster. However, when wire delays become more significant than gate delays, this will change. As wires become thinner, and resistance per unit length increases, the delay through a wire (the RC time constant) will only decrease linearly with its length, which depends

on the size of the gates. Thus, since the gate delay also decreases, the number of gates reachable in a single gate-delay-time will decrease: shrinking a wire-delay-limited circuit will not make it run any faster. It has been estimated that in a 0.1µm process, only 16% of the die length will be reachable in a single clock cycle [Mat97]. Current machines are based on a single complex CPU, but, by the above rule, large circuits will not make best use of the faster logic available in the future. Instead, a more partitioned design will be needed, where signals are localised to small areas of the chip.

Secondly, there is the desire to execute several instructions at the same time, and indeed most current microprocessors are superscalar. They find instructions to execute in parallel at runtime by searching a limited 'window' for them. Although some improvements have been had from this technique, there are steeply diminishing returns when increasing the number of instructions which can be issued per cycle. For example, a simulated 8-issue processor averaged less than 1.5 instructions per cycle on the SPEC92 benchmarks [TEL95]. Superscalar processing, and alternatives for parallel execution, are covered in Chapter 2.

One particularly important form of parallelism is hiding memory latency, i.e. continuing to execute other instructions whilst a memory load or store completes. Current machines rely on out-of-order execution to rearrange instructions around pending memory operations. As processor speed increases faster than memory speed, the latency problem becomes relatively worse until eventually the memory system totally determines performance [WM95]. The parameters assumed here put memory latency at hundreds of processor cycles in ten years time, and out-of-order techniques are unlikely to be able to hide this. Chapter 2 and Chapter 4 discuss processors and memory systems in more detail.

1.1.2 The Java software environment

The Jamaica architecture is designed with the Java language and multitasking operating systems with graphical user interfaces in mind, to support the network-transparent applications which will be required in future. The Java environment is quite different to that of serial C or Fortran, for which existing processors have been optimised.

Java execution is by run-time compilation from bytecodes. This means that highly complex optimisations cannot be performed because the compilation must happen quickly. Thus, any architecture which needs extensive optimisation will not perform to its full potential; ahead-of-time compilation is not appropriate in general, because class files can be loaded into a running virtual machine. Also, object-oriented programs employ frequent method calls with dynamic binding; this does not suit deep pipelines, which rely on branch prediction to keep them filled. Conventional machines are optimised for scientific code, with very repetitive behaviour.

In addition, the workload will be different. Language-level support is provided for multithreading, and multimedia applications will be particularly important.

1.2 The Jamaica architecture

The Jamaica architecture is designed to tackle the problems outlined in the previous section. Rather than searching at run-time for instruction-level parallelism, it can be represented naturally in program code at the process and thread level. Jamaica is therefore a *multithreaded single-chip multiprocessor*: multiprocessing is the basis of performance and scalable design, and multithreading improves throughput in the presence of memory and synchronisation delays. Each processor may be small-way superscalar (e.g. 2-issue like the Stanford Hydra – see §2.1.3), but for simplicity and to investigate the novel threading mechanisms alone the initial design uses only simple in-order single-issue CPUs[1]. It is not intended that each processor be wide-issue superscalar; instead, extra simple CPUs should be added[2]. The partitioned design of a multiprocessor helps to solve the wire-delay problem, since signals are localised; it also makes design and verification easier, since a simple circuit can be replicated [ONH+96].

To assist frequent method calls and fast compilation, Jamaica uses a windowed register file. To combine this with multithreading, though, the win-

1. There is a cost-benefit trade-off to be considered here.
2. In the longer term, as a billion transistors or more are possible on a chip, there may be so many simple CPUs that they cannot cooperate effectively; more complex CPUs should then be considered.

dows are heap-allocated rather than forming SPARC-style circular buffers. Chapter 3 reviews register file organisations, and the origins of this technique.

The Jamaica architecture seeks to use threads of a finer grain than might be usual in Java applications, so that inner loops and recursive algorithms can be parallelised efficiently.

1.2.1 Fine-grained thread-level parallelism

For efficient and scalable performance, parallel sections of code are indicated by threads rather than found implicitly by the CPU. For good load balancing and latency tolerance, the threads should be fine-grained and plentiful: this means they must be *cheap to create and manage*. In addition, irregular algorithms like tree walks and optimisation problems can be parallelised more easily if the decision about partitioning the work into threads is taken at run time; this is termed *dynamic parallelism*.

For this multithreaded multiprocessor, the philosophy is that tasks will be divided into cooperating threads as a matter of course, whether by the programmer (supported by modern thread-aware languages like Java), or automatically by compilers or source transformation tools (§8.6). The performance of the program as a whole then depends on the throughput of the system. Individual processor utilisation (idleness at any given time) is no more important than is functional unit utilisation for a superscalar machine.

1.2.2 Load balancing

A conventional multiprocessor application will create one thread per processor and then handle load balancing in software, either by keeping a queue of waiting tasks or by subdividing a task on demand (as done by guided self-scheduling). This scheme will not cope with a large number of fine-grained tasks, as the overhead of the software management is too great, and the centralised scheduling (whether a work queue or a manager task) presents a bottleneck.

Distributed scheduling systems should scale better. They can be classified as *sender-initiated*, where overloaded processors attempt to give away work, and *receiver-initiated*, where idle processors are responsible for finding work.

A receiver-initiated scheme requires the maintenance of a significant amount of information, so that idle processors know what work can be taken; on the other hand, the most communication occurs when the system would otherwise be idle. In contrast, sender-initiated schemes can potentially be more efficient, because decisions on splitting the workload remain with the sender; unfortunately, most communication occurs when the system is busiest.

The Jamaica architecture uses sender-initiated work distribution by creating new threads on idle processors. The crucial aspect here is deciding quickly whether a new thread should be created, so that this test can be performed frequently (§5.2.3). For example, a function recursing over a binary tree could test, in each invocation, whether to fork off one of the recursive calls. To work efficiently on fine-grained tasks, the thread creation should then be fast also. The Jamaica architecture therefore provides hardware support for deciding, in a single cycle, whether there is an idle context available. Passing parameters and then starting the remote thread take only a single bus transaction.

1.2.3 Integration with Java

Whatever features are provided at the hardware level must be usable from software. However, Java threads have more features than the simple fast threads described above: they can be suspended, acquire locks, and wait on objects, and these all require software management which means that the runtime system must be aware of them. Ideally, use of the fast hardware mechanism should not be slowed down by introducing software management layers.

There are three possibilities for integrating dynamically-created threads at the Java language level. Firstly, they could be separated from normal Java threads, say by introducing a `BasicThread` class which does not support the normal `Thread` manipulation functions. This is a change to the Java semantics, and introduces some problems since much of the standard Java class library assumes that real threads are running. For example, how should the `Thread.currentThread()` function respond? The second option is to promote the light hardware threads to genuine Java threads on demand, so that the runtime system only intervenes if one of the management functions is called. The third is that the lightweight threads are real Java threads all along,

which means pre-creating them and keeping them ready for use, then re-using them when they terminate.

Here, the third solution is adopted: a Java `Thread` object may as well be provided, since the lightweight threads would anyway need stacks and other private data. The Java runtime system and automatic parallelisation tools are described in Chapter 8.

1.2.4 The memory system

Even with a billion transistors available, it will not be possible to fit the entire main memory on the same chip as the processor (since that allows only 128MB of single-transistor DRAM[1]). Therefore, the memory system must retain off-chip DRAM, supported by on-chip caches. A Rambus-style interface [Ram99] is assumed.

As discussed in §4.2, private first-level caches and a shared second-level cache are used, connected by a shared bus; the L1 caches must supply the bandwidth to the individual processors, and reduce the shared bus traffic to a manageable level. A write-back policy is employed, since that results in more efficient bus use (fewer requests each of a greater size) than with write-through. The bus itself must be able to handle the bandwidth requirements (memory requests and coherence traffic) of all the threads, and most existing protocols were not designed with single-chip multiprocessors (CMPs) in mind; a pipelined, split transaction protocol has therefore been developed (§7.2). For simplicity of programming, sequential consistency is used between the L1 caches.

To enhance the performance of spinlocks in a multithreaded processor, so that a spinning thread does not monopolise the pipeline, an extension to the usual load-locked/store-conditional method has been introduced. The `WAIT` instruction puts a thread to sleep until its locked cache line has been written by another thread.

1. This neglects the increased density and hence more transistors that may be possible with a DRAM array. Also, fabrication processes are optimised for either logic or DRAM, and merging the two results in compromised performance of both.

1.2.5 Summary

The Jamaica system has the following features, all of which are presented in this work:

- Single-chip multithreaded multiprocessing.
- Hardware mechanisms for finding idle contexts and forking new threads to them, to support fine-grained dynamic parallelism.
- Heap-allocated register windows, for fast procedure calling and flexible allocation among multiple threads.
- A pipelined, split transaction cache coherence protocol on the on-chip shared bus.
- An extension to the usual load-locked/store-conditional synchronisation method, to allow efficient spinlocks in a multithreaded processor.

The ideas behind many of these are not new. For example, the concepts of single-chip multiprocessing, heap-allocated register windows and split transaction protocols have been studied before by other researchers. Jamaica's novelty lies partly in the way these features have been brought together for the first time, and partly in the implementations, which are new for this system. The thread distribution mechanism and `WAIT` instruction are unique to Jamaica.

1.3 Design parameters

A very simple analysis of the design parameters can be done, to estimate roughly how many processors and threads are necessary for a balanced system.

Two configurations will be considered: the kind of processor which could be constructed out of current or near-future technology (C), and the projections for ten years in the future (F). The parameters, except for the memory access time, are derived from the Semiconductor Industry Association technology roadmap [SIA98], and summarised in Table 1.

The memory access time, in processor cycles, is therefore 70 (C) rising to 224 (F). If one instruction in 3 is a memory access [HP96], with a cache hit ratio of 95%, then a memory reference will be required every 60 cycles, implying that

Table 1: Technology parameters

	Current (C) (2000)	Future (F) (~2010)
Local CPU clock	1 GHz	10 GHz
Global chip-wide clock	1 GHz	2.5 GHz
Memory bus data rate	800 MHz	2.5 GHz
Memory (DRAM) read time[a]	70 ns	22.4 ns
Logic transistors/chip	25 M	1000 M

a. Current figure from Rambus data sheet [Ram99], based on a 40ns core access time plus control and bus delays; Future figure obtained by scaling with the data rate.

3 (C) to 5 (F) threads per processor will be required to hide the latency. (A more detailed analytical model along these lines is given in [SBCvE90].)

Assuming a 128-bit wide on-chip bus, the global clock restricts the bandwidth to 14.9 GB/s (C) to 37.3 GB/s (F). Each RDRAM channel transfers data at 16 bytes in 8 data cycles, for a bandwidth of 1.49 GB/s (C[1]) or 4.66 GB/s (F). If the second-level cache has a 50% hit ratio, four RDRAM channels will suffice for bandwidth. Each transfer takes 10ns (C) or 3.2 ns (F), so seven-way pipelining will saturate a channel, for a total of 28 outstanding transactions.

A 256-bit L1 cache line every 60 processor cycles represents a bandwidth requirement of 0.497 GB/s (C) or 4.97 GB/s (F) per fully-saturated processor; thus the bus can support 30 (C) or 8 (F) such processors. (In the former case, the total bandwidth used between the processors and their L1 caches will be 10 words per processor cycle, impossible to sustain with a conventional superscalar design and single data cache). Each L1 line fill takes two 128-bit RDRAM transfers, so the four RDRAM channels will need a total of 16 outstanding line fills, which is easily satisfied.

Therefore, with current technology, a 30-processor system, containing a total of 60 threads, will perfectly saturate the ALUs, on-chip bus and memory channels, for a balanced design. For the future parameters, only 8 processors and 32 threads can be satisfied by the memory and bus bandwidth; this illus-

1. This equals 1.6×10^9 bytes/second, which the Rambus documentation refers to as 1.6 GB/s.

trates the global communication problem and increasing gap between local and global clock rates. The L1 cache hit ratio assumed here is conservative; in practice, more processors may be supported. However, unless more external memory bandwidth can be found, there is no need to replace the shared bus with a point-to-point network.

This analysis is extremely simplified: no allowance has been made for irregularity in memory access patterns, cache interference (sharing or collisions), writebacks, bus contention, memory bank conflicts, cache-to-cache transfers, and so on. Contention, and hence delay, is unavoidable if the bus is to be used efficiently (i.e. without large spare capacity), and these latencies must also be hidden. The detailed simulation work will investigate these aspects. It nevertheless demonstrates as an approximation that the level of integration, and the bus and memory bandwidth/latency requirements, are feasible.

1.4 Research aims

This work aims to show that the Jamaica architecture offers a promising approach for future systems design. In particular, the hardware thread distribution should allow fine-grained threads to be used for parallelisation, and the memory system must offer reasonable scalability.

The system's features are evaluated with the help of three sets of benchmark programs. The first consists of simple assembler programs, which test out the memory system (array operations and lock contention), thread distribution and register windowing. This uncovers the bounds on performance imposed by each subsystem.

Secondly, simple Java programs are used, to investigate the performance of the threading mechanisms from a high-level language, with its associated runtime system, and to find out what granularities may be used efficiently.

Thirdly, two 'real' Java programs are used: MPEG encoding and decoding are performed, with semi-automatically parallelised inner loops. The threading mechanisms are compared against more traditional software solutions.

1.5 Organisation

This thesis is arranged in three parts. Part one reviews the ideas upon which Jamaica is based: Chapter 2 examines the ways in which programs may be expressed within a computer, from conventional control-flow architectures to dataflow: multithreading can be considered a combination of the two. Chapter 3 goes inside the processors, to consider the structure of register files, and then Chapter 4 looks at memory systems.

In part two, the Jamaica architecture and runtime system are described. The instruction set, Chapter 5, introduces most of the novel features listed in §1.2.5. Chapter 6 then describes a possible implementation of the architecture, which is modelled by the Jamaica simulator.

Most existing cache coherence protocols are not designed for single-chip multiprocessors, so a new protocol has been developed for Jamaica. The memory system is simulated in detail, and is presented in Chapter 7.

Finally, the system software is in Chapter 8. The thread-management parts of the runtime system, to make the hardware threading mechanisms usable at the Java language, are the main contribution here. An assembler and linker were also developed to support the simulator. For completeness, some work by other researchers is also described: the modified Javar parallelising compiler (§8.6) and the Jamaica back-end to the C compiler were produced by Ian Watson, and the jtrans Java bytecode to assembler translator (§8.2) was developed by Ahmed El-Mahdy. They both worked on the remainder of the runtime system.

In part three, the benchmark programs are run on the simulated system to detemine the effectiveness of Jamaica's new features over a range of problems.

I. Design space review

CHAPTER
2. Instructions and threads

2.1 Control-flow parallelism

The fundamental measure of computer performance considered here is the execution time of programs[1]. One can decompose this time into $\text{time} = \frac{\text{IC} \times \text{CPI}}{\text{Clock rate}}$, where IC is the instruction count, and CPI is the average number of cycles per instruction. Architectural features can affect all of these factors; if it is assumed that the instruction set and program code remain constant, only the variation of CPI and clock rate need be considered. CPI can be decreased either by reducing the cost of a single instruction (for example, caches reduce the average time for memory operations) or performing several instructions at the same time - *parallel* execution. Parallelism can be subdivided into *pipelining*, where successive instructions are partially overlapped in time but only one instruction is at any particular stage[2], and true parallelism (for want of a better term), where functional units are duplicated and more than one instruction can be executing at one time. From now on, 'parallel' will refer just to this form.

Several levels of parallelism can be distinguished at the software and architectural levels: the differences are in how the parallelism is expressed in the executable code, and the granularity of the parallel sections produced.

1. In certain applications price/performance, power consumption, electromagnetic emissions or physical size might be more important.
2. so that the marginal time for each instruction is equal to the time for a single pipe stage, ignoring pipeline stalls and bubbles caused by fetch limitations or branch mispredictions, for example

2.1.1 Instruction-level parallelism

At the lowest level and finest granularity is *instruction-level parallelism* (ILP), where several instructions can be executed at once within the same processor. To illustrate, the starting point (Figure 1a) is a pipelined control-flow load-store architecture with the pipeline split into three stages[1]. The Fetch stage

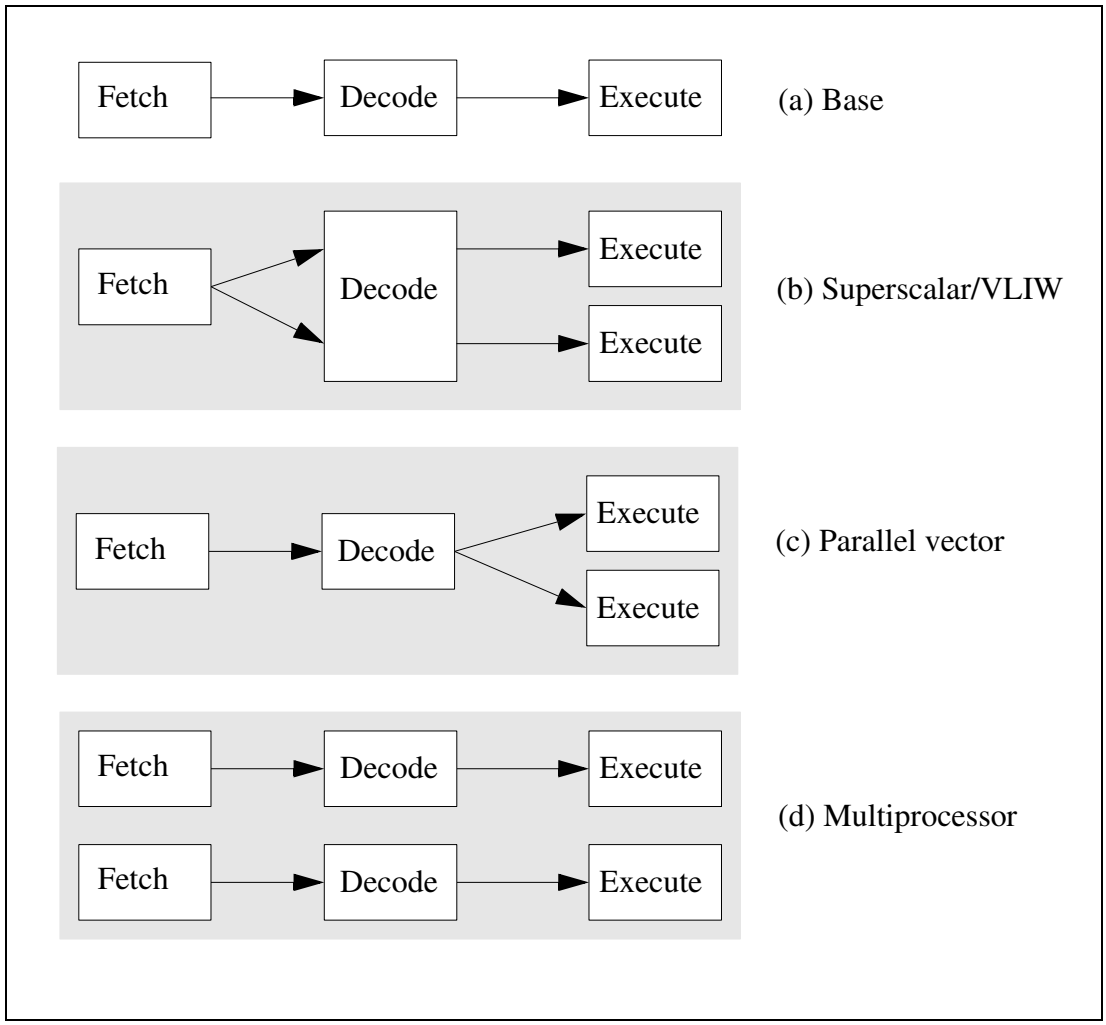


Figure 1: Parallel pipelines

holds the program counter and brings instructions in from memory; the Decode stage performs some preliminary work, for example fetching register contents and determining when the instruction can issue; then the Execute stage actually performs the operation (whether in the ALU or with the memory). In recent microprocessors each of these blocks can be split into several pipe stages, but the overall structure is still there.

1. The original ARM had its pipeline organised in exactly this way [Fur89].

The processor in Figure 1b issues several independent instructions per cycle, from the same instruction stream. If the bundles of instructions to issue together are marked by the compiler, a *VLIW* (Very Long Instruction Word, referring to the encoding) processor results[1] [Fis83]. If the Decode stage can determine the dependences between instructions and decide which can be issued together, it is termed *superscalar*. Superscalar processors can be subdivided into *in-order* machines, where the instructions must be issued in the same order as they appear in the instruction stream, and *out-of-order* (OOO), where the Decode stage is free to issue an instruction whether preceding instructions have been issued or not (assuming there are no dependences on them). The processor performs *static* and *dynamic* scheduling respectively. VLIW machines are inherently statically scheduled. The additional run-time flexibility, from VLIW up to OOO, means that better performance can be achieved, but it comes at the cost of significant hardware complexity since the hardware must resolve the instruction dependences.

The introduction of ILP in this way is essentially transparent to software written in high-level languages (and even, in the superscalar case, to machine code). Compilers may generate better code if they can optimise for a specific design [LPU97][Muc97], but the granularity is below the level of source code expressions; if the compiler does schedule instructions then it must be automatic.

The big advantage of superscalar designs is that no change is required to the instruction set, so that new processors can execute code compiled for earlier machines. Their disadvantage is the restricted scope in which they locate ILP: any superscalar machine has a limited number of instructions (maybe 64) under consideration in its issue window at any one time. For example, independent iterations of a loop whose body is larger than that size will not be located. Studies have shown many complex techniques are required even to achieve an average parallelism of 10 [Wal93].

Other systems indicate potential parallel execution more directly to the hardware.

1. Warning: here 'instruction' is used to mean the individual operations encoded within the bundle; some authors use it to mean the bundle as a whole.

2.1.2 Vector parallelism

A single *vector* machine instruction causes, say, 64 additions to be carried out; the parallelism here is implicit, and an implementation can use this information (Figure 1c). Some 'multimedia' instruction sets (e.g. Intel's MMX and SSE, Sun's VIS) hold short vectors, e.g. four 32-bit floats (SSE) or four 8-bit integers (MMX), in a single conventional register; this is termed *subword parallelism*. Vector operations fit best with numerical applications dealing with matrix and vector arithmetic; many graphics and signal-processing algorithms can be expressed in this form [KP98].

Vector parallelism is less general than ILP, but much easier for the hardware. The complexity of finding the independent operations is moved to the compiler: high-level languages may allow vector operations to be specified directly, or else a vectorising compiler can transform inner loops. Many multimedia kernels are written directly in assembly language.

2.1.3 Threads & processes

At the highest level come threads and processes. The *process* is a familiar entity in multitasking operating systems; it consists of a control-flow sequence of instructions executing in some protection domain (virtual memory space and set of permissions). Since each process executes within its own independent virtual machine, *process-level parallelism* is automatic[1].

More recent operating systems distinguish threads and processes [Tan95]; the process continues to be the unit of protection and resource allocation, and a *thread* is a control-flow abstraction within a process. A traditional process is thus equivalent to a process containing a single thread. Threads have the advantage that they can simplify programming when concurrency is naturally present but awkward to express in serial languages (for example, a web server responding to multiple client requests can more cleanly separate the state of each transaction). Transaction processing is a typical, commercially-important, workload with an abundance of threads [BGM+00][Die99].

1. Assuming that the processes don't deliberately become aware of each other, by sharing files or memory areas for example.

Since process-level parallelism can be seen as a special case of thread-level parallelism (i.e. where the threads just happen to belong to separate protection domains), the distinction will not usually be made. One difference is in the frequency of synchronisation: threads in independent processes are less likely to be cooperating and communicating, so the granularity of work is greater for processes.

In hardware, thread-level parallelism corresponds most directly to *multiprocessing*, where the whole pipeline is duplicated, resulting in several independent processors in the same machine (Figure 1d). Communication between processors is done through shared memory or explicit message passing. Since communication is typically much slower than computation, the units of work scheduled on the processors must be large (coarse-grained parallelism). Multitasking operating systems already have a supply of independent processes, which would have been time-sliced on a single CPU in any case, but if performance on a single task is required then it must be divided into threads in order to benefit.

Parallelising compilers transform outer loops in high level code so that iterations can be distributed to the processors; alternatively, explicit threads can be used at the language level. Compilers for some declarative languages can also generate threads automatically (e.g. the LLNL Sisal compiler).

Any system can of course exploit parallelism at several levels simultaneously, as when superscalar processors are used to build a multiprocessor machine. Of particular interest is a *single-chip multiprocessor*, where integration can reduce the cost of communication to allow better cooperation between the threads.

The Stanford Chip Multiprocessor (CMP)

The Hydra project at Stanford University has proposed a chip multiprocessor [HNO97]. They estimate that eight 2-way superscalar processors could be integrated in the same area as a single 12-issue processor. Partitioning the machine in this way has several advantages, as discussed in §1.2: communication is more localised, reducing problems with wire delays and allowing higher clock rates, the first-level caches (one instruction and one data cache per proc-

essor) can be smaller and faster, and design and verification effort is reduced, as a simpler core can be replicated to make the multiprocessor.

In the simulations reported in the above paper the second-level cache and memory system are also integrated onto the chip, and are the same in each experiment. (The L2 cache access time is slightly increased for the multiprocessor, since it must communicate with several different request sources).

Running a benchmark with no thread-level parallelism ('compress', from the SPEC95 suite), the 12-issue superscalar machine gains only a 43% performance increase over the single occupied 2-issue element in the multiprocessor. On benchmarks which feature thread-level parallelism (mpeg and tomcatv), the CMP performed 2.4 and 5.3 times better.

This illustrates the advantage of systems where parallelism is indicated in software, rather than relying solely on the hardware to locate it at run time. The Hydra CMP is described in more detail and compared with Jamaica in §5.5.1.

2.2 Alternative programming models

The above processor architectures are based on *control flow* - the program code consists of sequences of instructions which are to be (or should appear to be) executed in order by some abstract machine. A thread is the abstraction of this control-flow sequence; each thread has some associated state (its *context*), including a program counter, register contents, and other control information. Instructions read and write either registers, which are assumed to be local to a thread[1], or memory shared between threads; thus the result of an operation (for example, adding the contents of two registers) depends on the register contents at the time of execution – getting the right answer depends on executing the instructions in the correct order. Two alternatives to control flow use the availability of data to drive the computation: dataflow and graph reduction.

1. In fact, some systems do provide registers accessible to all threads, for example shared queues in Threaded Windows (§3.3.2 and [QDT88]), or allow a thread to write into another's register set ([CDK+98]).

2.2.1 Dataflow

The first alternative programming model is *dataflow* [GKW85]; whereas a control flow program specifies when an instruction is to execute, with the data being implicit in the register and memory contents at the time, a dataflow program describes what should happen to particular data items and the timing is implied - an instruction can execute as soon as its operands are available. This flow of data is the real purpose of a control-flow program; a dataflow program just makes it explicit. (There is a similar relationship between imperative and declarative programming styles.)

A dataflow program is represented as a directed graph. Nodes represent computations; they accept some data and produce a result. The arcs show the flow of values from producing nodes to consuming nodes; when a node has data on all necessary inputs, it can execute. Dataflow architectures attempt to model the data flow directly, with nodes being single instructions. There is no program counter, and no single 'correct' sequence in which to execute the instructions. Instead, executable code gives the dependences between the instructions, and an instruction can (theoretically) execute as soon as its operands are available.

One problem with dataflow architectures is managing the potentially vast pool of instructions waiting at any given time[1]. Some implementations have tried a hybrid approach: each node in the dataflow graph becomes a (control-flow) thread of several instructions, with dataflow sequencing between threads. The flexibility of full dataflow is not always needed; the more efficient control-flow style can be used, for example, when a result is immediately consumed by only one instruction. The advantage is that the grain size of the managed tasks has increased from one to several instructions, and the number of tasks is correspondingly reduced. In the above hybrid approach, each thread starts execution with some parameters (taken from the input arcs), produces some results for output arcs, and then dies; the data flow is still driving the computation, since it is only the data which determine when a thread should execute. The MIT Monsoon [TPB+91] made use of this idea.

1. The out-of-order dataflow core does not suffer from this problem because the instruction window is of a limited size.

Dataflow implementation techniques can be used inside control-flow processors; this is the basis for out-of-order execution. A window of instructions is filled in the normal control-flow manner (loading from the program counter address), and then the data dependences are analysed and made explicit (by *register renaming*). Instructions can then be issued independently to the execution pipelines as soon as their operands are available. This is of particular value when some operations may have long (and variable) latencies; in particular, a cache miss on a load instruction can cost hundreds of cycles, and executing instructions which do not need the loaded value during this latency period is potentially beneficial. However, the control path is dependent on the results of some instructions (conditional branches), so at any point only a limited number of instructions are certain to be executed, and to hide latencies and find parallelism it is best to have a large number of instructions available in the window. It is possible to execute some instructions speculatively (for example, using branch prediction) if they will not have any permanent side-effects (the results can be thrown away), which adds some improvement, but the benefits decrease as the proportion of correctly-speculated work reduces.

2.2.2 Graph reduction

The dataflow model described above corresponds to an eager evaluation of the program graph [AG94]: a node is activated when its inputs are ready, and arcs in the graph indicate to where results are sent. *Graph reduction* turns the directions around: the program is represented as a graph of expressions, and the arcs indicate from where arguments come. The computation proceeds by transformations (*reductions*) on this graph. Transforming subgraphs is purely functional; a reduced subgraph has exactly the same meaning as the original, and can be substituted anywhere in its place; indeed, graph reduction is used to implement functional languages. If desired, the reduction of an expression can be delayed until its value is required (which gives lazy semantics; the execution is demand-driven). The most important transformation (termed β-reduction) is function application: the graphs used as arguments of the function are logically substituted into the function's graph in place of its parameters. Peyton Jones [PJ87] describes the subtleties of this in detail.

At any point, there can be many subgraphs awaiting reduction: this provides implicit parallelism. A multiprocessor system can work on reducing many subgraphs simultaneously, with a granularity of work corresponding to a function body in an imperative language. In general, each task (reduction) will spawn off several more tasks as new reducible expressions are discovered; thus an implementation must consider how to manage a (potentially large) number of medium-grain threads, and distribute them to processors efficiently.

Rediflow

Graph reduction machines provide support for this model in hardware. One such machine is the University of Utah's Rediflow[1] [KL84][KSL87].

The Rediflow system is a NUMA (Non-uniform memory access) shared-memory multiprocessor (i.e. each processor has direct access only to a local memory and non-local accesses are routed through a switching network, although all memories form a single global address space). Each processor is a conventional control-flow machine; the interesting parts lie in the switching network. Each switch maintains (for its processor) two pairs of queues. The IN and OUT queues hold messages to implement the globally-visible memory. The APPLY and LOCAL queues manage the work distribution; the APPLY queue holds pending function applications (i.e. reduction operations), and these could be executed by any processor in the system. The LOCAL queue holds work which should only be executed by this particular processor.

Load balancing is achieved automatically by the switches. Each switch calculates a 'pressure' for its processor, based on the length of the work queues and the amount of free memory remaining. Work should 'flow' from high to low pressure processors; switches exchange pressure information with neighbouring switches, and at high pressure points take packets out of their APPLY queues and place them on the network. The work packets are routed along pressure gradients to a local pressure minimum, and placed in the APPLY queue of that processor. The distribution takes no account of memory locality and Rediflow does not use data caches, so moved tasks will suffer a larger number of remote memory accesses.

1. The name derives from a combination of 'reduction' and 'dataflow'.

In a normal multiprocessor system, the task of scheduling threads would lie with the operating system. Rediflow has replaced this with a hardware mechanism, and is thus unlike the other architectures considered in this chapter which only consider thread and instruction scheduling within a processor.

A disadvantage of the graph reduction model, inherited by Rediflow, is that the parallel tasks are produced whether or not they will be used. The structure of the computation is maintained in the graph and is an overhead when compared with a control-flow implementation, where the structure is partially in data and partially implicit in the code and program counter. Some other parallel function language systems have reduced the cost of representing all these tasks (§8.6.1), but this overhead remains.

2.3 Multithreading

The convergence of control-flow tasks and dataflow scheduling can also be approached from the other direction. If a program consists of a number of threads which communicate by explicitly passing messages (in the style of 'communicating sequential processes' [Hoa85]) then a different hybrid results. Typically, some threads will be available for execution, whilst others are blocked waiting either to transmit or receive data. The computation consists of processing interleaved with communication, and the threads have a lifetime and state of their own. This is a control-flow design (the program code says when data transfers are to take place), but the data flow determines what threads are available for execution at any given time.

Such a system could be implemented on a conventional (multiprocessor) machine using software scheduling of the threads, but it is far more efficient if the processor is modified to hold several thread contexts internally; this is a *multithreaded architecture* (MTA). The Fetch stage is logically duplicated, but the instructions are then fed through a common pipeline (Figure 2e). The communication of most interest is not usually between threads but between a thread and the memory system, and MTAs are particularly concerned with hiding memory latency. The technique of context-switching when a particular thread cannot immediately make progress is more general, though, and can also be used to hide functional units' latencies, if the context switch is fast

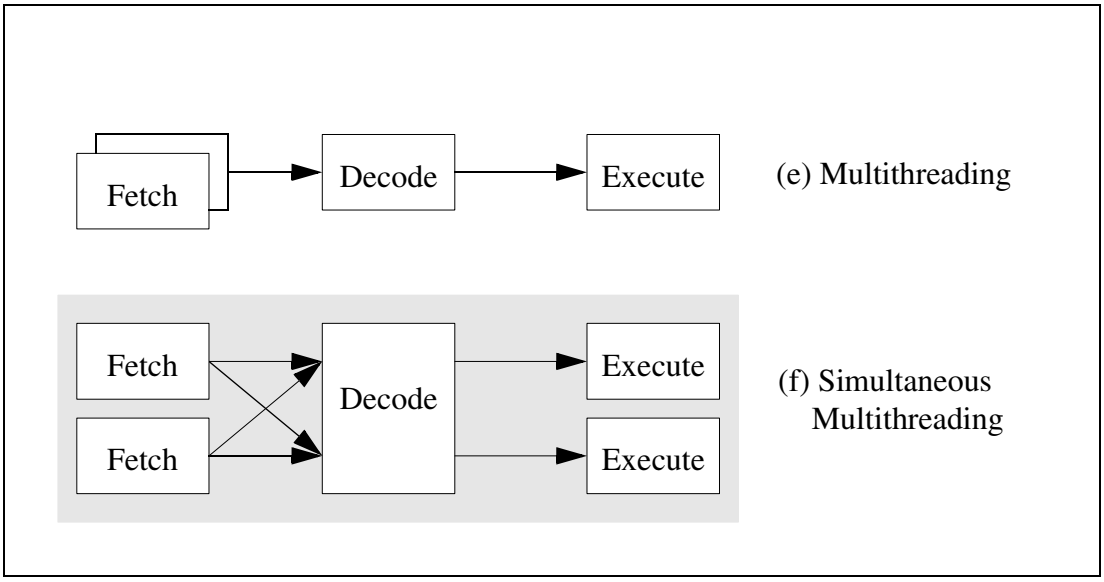


Figure 2: Multithreaded pipelines

enough. It should be noted that multithreading does not directly improve the execution time of an individual thread; rather, it improves the throughput of the system as a whole. In this respect it is similar to multiprocessing, and can be viewed as multiplexing several virtual processors onto one physical processor.

Multithreaded architectures differ depending on the policy determining when to switch threads. *Fine-grained* MTAs can switch to a different thread each cycle; the usual alternative is to switch on a cache miss or an explicit instruction, termed *coarse-grained* or *block* multithreading. Cycle interleaving provides better performance than block switching for deeper pipelines; for short pipelines (5 stages), they behave similarly [LGH92].

The Tera MTA (see below) and MIT Alewife [ABC+95] are particularly control-flow based multithreaded architectures; the MIT *T [NPA92] and Moore's Anaconda [Moo96] are more influenced by dataflow.

2.3.1 The Tera MTA

The Tera MTA [ACC+90][AAC+94][AKK] is a multithreaded machine aimed at the supercomputing market. The basic philosophy is that shared-memory models are easier to program than message-passing, that predictability is important to automatic parallelisation, and that sufficiently many threads can be created that multithreading can be used to hide memory latency. Each VLIW processor can hold 128 threads, and issues a bundle of three in-

structions from one thread each cycle; threads are interleaved cycle by cycle. The processors have no data caches; instead, a high-bandwidth network is used to connect all the processors to the memory modules. Thus all memory references have predictable latency, and no compiler optimisations like blocking need be performed. It also simplifies scheduling, since a thread will get equivalent performance whichever processor it executes on, and cache interactions (sharing or collisions) are not considered.

The disadvantage is that many threads are necessary to hide the memory latency; the designers estimate that up to 80 threads per processor will be required. In certain cases these can be produced automatically by the Tera Fortran compiler - for example, for matrix operations. Here, the primary selling points of the machine are its high memory bandwidth and easy programming. Although the designers have tried to make parallel execution as easy as possible (e.g. full/empty bits on each memory word), inherently serial tasks will execute very slowly under this architecture. It could be argued that the absence of caches achieves predictability by making every thread predictably slow.

Scheduling within a processor is handled by the hardware (selecting which thread will issue each cycle) and runtime system (allocating thread contexts to processes). Creating threads on a different processor requires interaction with the operating system. Thus, although threads are central to the machine, only coarse-grained ones can be used for parallelism.

2.3.2 Simultaneous multithreading

One problem with wide-issue superscalar processors (with 8 or more instructions issued per cycle) is that the pipelines cannot be kept occupied: the combination of memory latency and dependences between instructions (including control-flow dependences) severely restricts the average IPC. *Simultaneous multithreading* (SMT) architectures issue instructions from more than one thread each cycle. The Fetch stage of a superscalar pipeline is extended, so that instructions can be fetched from more than one thread at the same time (Figure 2f); these instructions are fed into a normal dynamic-scheduling Decode stage. After that, the Execute stages can work on the instructions without

regard to which threads they came from, multiplying the number of independent instructions available for execution.

To avoid an explosion in complexity, SMT processors must restrict the number of threads used for fetching each cycle, and the number of instructions fetched from each. A variety of scheduling algorithms has been considered [TEE+96].

An SMT processor executing a single thread looks just like a dynamic superscalar machine, and no performance is 'lost' when only one thread is available. SMT is therefore a natural incremental technique to improve the throughput of an out-of-order pipeline without injuring single-threaded code, and has the generality to exploit ILP and multithreading at the same time; these advantages are often phrased in terms of 'utilisation of resources' (presumably meaning execution units). However, it does not necessarily follow that it provides the best absolute performance. The superscalar design upon which the SMT processor is based has a quadratic area cost, arising from the bypass paths which move results between the pipelines: each ALU may need the value just produced by any other ALU [PJS97]. Therefore two n-way pipelines will occupy *less* area than a $2n$-way pipeline. The actual cost of the execution units is small compared to that of the logic required to keep them occupied, and better performance might be achieved by having more ALUs with lower individual utilisation [HNO97].

This trade-off has already been considered within superscalar designs: the high cost of fully-connected execution pipelines led the designers of the Alpha 21264 to split the ALUs into two clusters, with more limited bypassing between clusters than within them [Com99]. A penalty is imposed on any instruction which refers to a result just generated within the other cluster; to avoid this, more complex scheduling must be used.

There is thus an argument in favour of multiprocessing, rather than pursuing wide-issue ILP at all costs. Some SMT proponents [EEL+97] argue that multiple superscalar processors on a chip would not be effective because, in addition to the lower ILP than an equivalent-area SMT processor, performance suffers when there is little thread-level parallelism. This argument might equally well be inverted: the extra issue width of the SMT processor provides

little marginal benefit to single-thread ILP, and (because of the fewer total ALUs) performance suffers when there is much thread-level parallelism. The costly out-of-order capabilities may also provide little benefit over in-order issue when multiple threads are present [HS99]; multithreading and OOO issue are tackling the same problems of latency and dependences, and are not orthogonal. An in-order SMT would sacrifice the single-thread performance.

Therefore, when much thread-level parallelism is available, SMT is inferior to single-chip multiprocessing; for a single thread, SMT is no better than superscalar, and only marginally better than CMP. It is more complex than either, so its only advantage is flexibility under differing workloads.

2.3.3 Finding threads

Multithreaded architectures, whether coarse-grained, fine-grained or SMT, need 'enough' threads to work efficiently; the exact number depends on architectural features and program behaviour. For example, in the Tera, every memory reference suffers a long latency, so many threads will be required (up to 80) per processor. Multithreaded multiprocessors with caches will have a much better average memory latency, and need fewer threads per processor.

These threads can come from three possible sources: the programmer, the compiler, or the processor itself.

If the threads are explicit, a programming language may support them directly (e.g. Java [LY96]) or through run-time libraries (e.g. C with Posix threads).

Parallelising compilers may identify threads automatically; for example, Fortran compilers and graph reduction (§2.2.2) implementations of declarative languages may do this. This approach is also taken in the Javar restructuring compiler [BG97], which transforms loops in Java code into threaded equivalents.

A novel alternative, simultaneous subordinate microthreading (SSMT) [CSK+99], does not rely on parallelising the application at all (although this could be done); instead, threads hand-written in an internal microcode format are spawned (at compiler-selected points) on behalf of the main thread to modify the processor's behaviour – for example, by providing a custom branch-

prediction algorithm. An SMT pipeline executes the primary and subordinate threads together.

Speculative threads

If insufficient threads are provided (in particular, when maximum performance on a single-threaded program is needed), some systems will attempt to parallelise automatically by *speculatively* creating new threads, executing portions of the program in parallel in the hope that there are no dependences between them. If a dependence is detected, the speculative thread must be terminated (*squashed*) and restarted serially. The two problems are detecting these dependences, and then undoing the effects of the speculative thread. The Wisconsin Multiscalar processor [JBSS97] uses this technique to run 'serial' code on what is effectively a multiprocessor; another project ('dynamic multithreading') used SMT instead [AD98]. The Atlas CMP combines speculative threads with aggressive value prediction [CW99a][CW99b].

Java has some helpful properties for this activity; in particular, a method cannot modify anything in its caller's stack other than retrieving its own arguments and (possibly) returning a single value; the only communication may be through global objects. Therefore, method calls form natural boundaries for speculative execution. Sun's MAJC [Sun99][Tre99] and the Stanford Hydra CMP [CO98][HHS+00] have both used this.

When non-speculative threads are available, speculation must be a poor alternative. However, it may be a useful extra technique for 'expanding' low TLP to fill a multiprocessor, and speculation techniques may be a good complement to Jamaica's threading features. However, for now, Jamaica focuses on user-supplied threads.

2.4 Scheduling: VLIW to SMT

Each of these architectures considered so far forms a subset of a wider design space. At the highest level is the degree of multiprocessing: the number m of independent pipelines present. Within each pipeline is some number e of execution units, to which the instructions are scheduled (whether by a VLIW, or static/dynamic-issue superscalar) after being fetched from t resident threads.

Fetching can be in blocks from a single thread, or else from (up to) some number *f* of threads per cycle.

Conventional superscalar processors therefore occupy the slice (1,*e*,1,1), while the multiprocessor Tera architecture spans the range (*m*,3,128,1). The examples considered so far are summarised in Table 2.

Table 2: Design space examples

	Multiprocessing	Multiple-issue	Threads	Thread Fetching
Superscalar	1	*e*	1	1
Simple MTA	1	1	*t*	block or 1
CMP	*m*	*e*	1	1
SMT	1	*e*	*t*	*f*

2.4.1 Scheduling axes

There are three levels of scheduling in operation. At the highest level, the operating system or runtime library assigns threads from its ready queue to one of the processors. Each processor (if multithreaded) then selects the threads for fetching, and finally the instructions within the threads are scheduled. The achieved performance depends on the quantity and speed of execution units, and the efficiency of the schedule.

Figure 3 illustrates these levels; it depicts a multithreaded VLIW multiprocessor. The OS is responsible for placing threads onto the processors; the fetch stage gathers instructions from one of the threads, and then the issue stage presents them to the execution pipelines. If the issue stage were superscalar then the scheduling here would be performed by the hardware; with VLIW, the binding to physical execution units has already been done by the compiler.

The objective of the schedule is maximum performance, i.e. maintaining processor utilisation. There are two axes for scheduling: deciding *where* to perform an operation (the *horizontal,* or space axis), and *when* (*vertical*, or time). The horizontal direction (instructions issued simultaneously) must have no dependencies, but the vertical can tolerate a certain amount: a dependent operation can begin when its prerequisites have reached a certain point. Threads and instructions are candidates for scheduling in either direction, and by either

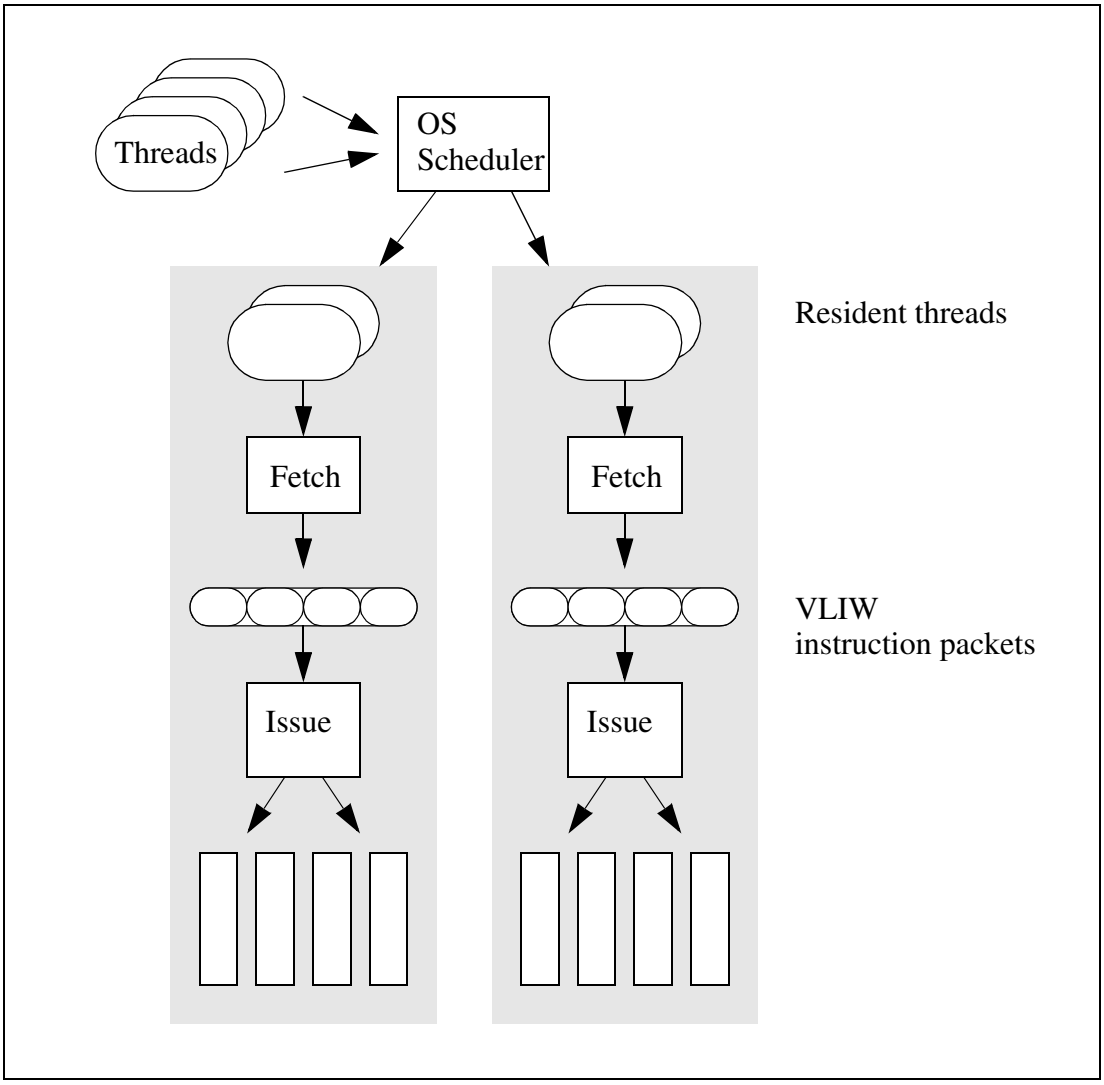


Figure 3: Three levels of scheduling. (A multiprocessor MTA)

hardware or software. In Figure 3, the operating system is performing both horizontal and vertical thread scheduling (i.e. allocating threads to the separate processors, and time-slicing among the available hardware contexts); the Fetch unit is selecting VLIW instruction packets cycle by cycle (vertical), and then the instructions are issued in parallel (i.e. the horizontal instruction scheduling has been performed by the compiler).

2.4.2 Vertical scheduling

Perfect vertical scheduling may fail for a number of reasons and on differing timescales. For example, a thread waiting for input may remain idle for several milliseconds (awaiting data from a hard disc), that is, millions of processor cycles. Threads waiting to acquire a lock will be idle for tens of cycles (assuming a fast communication system, e.g. a shared L1 cache) if the lock protects a simple variable update, up to millions if some complex transaction is in progress.

At the instruction level, an operation may have to wait on a cache miss for tens or hundreds of cycles, and a pipeline bubble of a single cycle may be caused by structural or data hazards.

Thread management in software (OS or library) is efficient on timescales of milliseconds, where blocked threads consume no hardware resources and context switches are relatively infrequent. Compilers can also schedule instructions for some functional unit latencies (as when delayed branches or loads are exposed by the pipeline); to do this, though, the delays must be predictable, the compiler must have detailed knowledge of the pipeline structure, and large enough regions of code (basic blocks) must be available[1]. Hardware vertical thread scheduling (i.e. multithreading) can hide memory or functional unit latency, on small timescales. Hardware vertical instruction scheduling is nearly always carried out: exceptions are some early RISC machines which exposed pipelining effects (delayed loads and branches). Out-of-order machines also allow instruction reordering by the scheduler.

2.4.3 Horizontal scheduling

Horizontal scheduling of threads is easy, since independence is guaranteed (but more complicated schemes, like gang scheduling, may get better performance by considering interactions between the threads [AKK]). Horizontal scheduling of instructions results in VLIW (software) or superscalar (hardware) architectures, which were discussed in §2.1.1.

2.4.4 Classification of existing architectures

The five major architectures presented (VLIW, superscalar, CMP, MTA and SMT) take differing positions within this design space. SMT has the most hardware-oriented view: as much as possible is scheduled within the processor. VLIW places most emphasis on the software: instruction and thread scheduling are both handled by the compiler and operating system. In between, various compromises can be found. Table 3 lists the scheduling mechanisms, and the directions in which they apply. Hardware squares are shaded.

1. Trace scheduling was developed to enable reasonable scheduling for VLIW machines [Fis83].

Table 3: Examples of scheduling methods

	Threads to processors	Threads within processors	Instructions to execution units
VLIW	Runtime system (V)	-	Fixed at compile time (H)
Superscalar	Runtime system (V)	-	Static or dynamic scheduling in hardware (H & V)
Multiprocessor	Runtime system (H & V)	-	Static or dynamic scheduling in hardware (H & V)
Tera MTA (multiprocessor)	Runtime system (H & V)	Hardware interleaving (V)	VLIW - fixed at compile time (H)
SMT	Runtime system (V)	Hardware selects several threads each cycle (H & V)	Static or dynamic scheduling in hardware (H & V)
Rediflow	Hardware assisted work queues (H & V)	Remove from work queue (V)	-

The trade-off between software and hardware depends on the granularity of the objects being scheduled. Software is cheaper to build and maintain, and offers more flexibility in its algorithms, but would not be suitable for instruction scheduling at run time. The Jamaica system, with its hardware support for mapping threads to processors, considers the neglected left-hand side of Table 3, so that fine-grained threads may be used for parallelisation.

2.5 Conclusions

This chapter has reviewed the spectrum of processor design, with particular emphasis on scheduling and parallelism. Instruction-level parallelism within a single thread provides some benefit, but has problems with scalability and

latency tolerance. Multiple threads, whether used with multithreading or multiprocessing, make parallelism more explicit and increase its availability.

The Jamaica system is therefore a single-chip multiprocessor, with multithreading and support for distributing threads. Automatic task management is more common in dataflow and graph reduction machines, which schedule finer-grained threads than conventional multiprocessors. Multithreaded architectures generally only perform vertical scheduling of threads; simultaneous multithreading adds horizontal scheduling within a single superscalar pipeline. The Jamaica system extends this to (limited) horizontal scheduling within a multiprocessor.

CHAPTER
3. Registers

The previous chapter considered several different ways of scheduling computation within a machine, i.e. specifying what it should do and when. This chapter looks in more detail at how data are stored within the processor. This is tightly connected with the instruction set of the machine, and the platform independence offered by Java bytecode means that new organisations can be considered for Jamaica.

Computers conventionally divide data between a large main memory and a set of registers inside the processor. There are two reasons for this: firstly, the small number of registers can be built out of a fast technology (allowing high power dissipation and multiporting) and kept close to the execution units; main memory is usually concerned more with density than speed. Since most references will be served by the register file this results in much greater execution rates. Secondly, registers may be identified in a small number of bits, so that instruction encodings are dense. Instructions referencing memory may either use a longer format (CISC designs), or build up the memory address using a register (RISC).

This division creates two independent address spaces within the machine, with different capabilities. The main memory is more flexible: locations are identified using a variety of addressing modes (e.g. by a constant value, or as an offset from the contents of some register). This allows complex data structures to be constructed, since memory locations can be calculated at run-time. High-level languages may allow these *pointers* (variables whose values indi-

cate memory locations, indicating the address of more data) to be manipulated directly (as in C or Pascal), or indirectly (as in Java).

On the other hand, the flat register address space (as found in the ARM, MIPS and Alpha architectures) has only one addressing mode: a constant index into the register file is specified as part of the instruction encoding, and is fixed when the code is generated. Pointers to registers cannot therefore be created, as there is no easy way to dereference them, so data structures based on pointers (e.g. lists and trees) must always be allocated in main memory. Individual values must still be brought into registers to be operated on, and frequently-used values should be maintained in registers to reduce memory traffic. Deciding which registers to use for particular variables (or compiler-generated temporaries, which have the advantage of being invisible to the language) is the task of the *register allocation* phase of the compiler.

High-level languages like C present an abstract model to the user where all variables reside in a single address space, so that any variable can have its address taken and used as a pointer[1]. Register allocation must therefore be transparently handled by the compiler.

Having two address spaces causes some difficulties: for example, a variable allocated in main memory but (temporarily) brought into a register will have a name in each of the spaces, a problem known as *aliasing*. Referring to the variable by one name (for example, storing into its memory location) will not affect the value in the other location (the register). Thus compilers must perform an analysis based on data flow or type information to determine when aliasing can possibly occur, or else severely restrict register allocation when pointers are used [Muc97]. Some hardware can help this problem: for example, if a register is holding the contents of a memory location then it could be tagged with the memory address, and loads and stores automatically checked against the tag. This turns the register file into a small specialised cache. The Berkeley RISC architecture specified a restricted form of this: the registers aliased a consecutive block of memory addresses, reducing the test to two comparisons [Kat85].

1. An exception would be C variables declared `register`.

3.1 Variables

Variables in high-level languages are usually assigned memory addresses according to one of three schemes [ASU86]. *Static* variables have an address assigned at compile (or link) time. *Stack* or *automatic* variables are created when control enters a particular function in the program, and remain in existence until the function returns. The LIFO nature of function-call nesting means that they can be allocated on a stack (as part of the function's *stack frame*), hence the name. *Heap* variables are created on demand[1] from a pool of available memory, and exist until explicitly destroyed (or garbage-collected when they can no longer be referenced). This classification (of lifetime or *duration*) is independent of the language policy determining when a variable can be named (*scope*).

Static variables are the most restricted: their number must be known at compile time, so recursive functions and dynamically-created data structures cannot be handled. Heap variables are the most flexible, and could simulate the other two, but must always be referenced through pointers (because the allocated address cannot be predicted in advance), and the allocation and deallocation require time.

Stack variables have received the most attention from computer architects because they are common, being closely tied to function calls, the building-blocks of high-level programs, and cheap (allocation requires only adjustment of the stack pointer). Functions are most likely to refer to their parameters or local variables, so stack frames have good locality of reference (both temporal and spatial), which can be exploited by caches. Some languages (in particular Java) do not allow local variables to have their addresses taken, and therefore they cannot be aliased – this makes register allocation very easy.

3.1.1 Stack caches

Some designs have tried to exploit this locality by constructing caches specifically tailored to the task of caching stack frames [DM82][Lop94]. The properties of stacks can allow optimisations not possible with ordinary caches: for example, when a cache line is selected for eviction from a write-back cache, the

1. Languages do not typically allow variable *names* to be created at runtime, so heap variables are anonymous and can be manipulated only through pointers.

copy back to main memory can be avoided if the evicted line was from the stack frame of a function which has since returned [Lop94]. Extra instructions must be inserted to communicate with the cache, which reduces the advantage over ordinary caches. A conventional cache would have captured much of the locality in any case.

3.1.2 Register allocation

Even if stack frames can be cached efficiently, it is still worth removing as many memory references to stack variables as possible to reduce the instruction count. Compilers usually support calling conventions which allow function parameters to be passed in registers, thus eliminating a store in the calling function and a load in the callee.

There are two other situations in which function calls may cause memory references. The first is the memory aliasing problem again. A caller cannot in general know which memory locations a callee might use or modify, so any registers holding globally-accessible data must be written back to main memory. The second is when the caller does not know which registers the callee will use. Any registers which might be overwritten must be saved, either to the homes of the variables which they are holding, or to temporary locations on the stack.

Calling conventions sometimes distinguish two classes of registers. *Caller-saved* registers may be overwritten by any function, so a caller must always preserve them if necessary; a clever compiler may allocate them only between (and not across) function calls. A *callee-saved* register is guaranteed not to be modified by a function call, and so no save and restore is done if the child does not in fact use the register. On the other hand, the callee does not know whether the register does contain any data, so the save and restore must always be done if the register is used. Caller-saving wins if the caller doesn't use the register; callee-saving wins if the callee doesn't use it.

Interprocedural register allocation attempts to allocate registers by considering the call graph of the program and examining several functions together. Effectively, functions can negotiate an improved calling format on an individual basis, so that saves and restores are avoided and arguments do not need moving

to fixed locations. To be most effective, a large number of registers needs to be available, otherwise the callee will end up saving and restoring the caller's registers anyway. The call graph of the function must be relatively simple, and the functions to be analysed must be compiled together. It is also computationally intractable, so realistic implementations must be based on heuristics.

3.2 Stack processors

Some machines don't use a flat register file at all, but replace the two- or three-address instructions with zero-address instructions. The operands are instead taken from a stack inside the processor, which may parallel or replace the call stack in memory. For example, an `ADD` instruction may pop the top two items from the stack, add them, and then push the result.

The stack can be in a separate memory space; in that case it must either have a limited number of elements (e.g. 4 for the Inmos Transputer [Inm88]), or else must be spilled to memory when it overflows by a software trap-handler, or by some dedicated hardware. Alternatively, it can be considered to reside in main memory, in which case a stack cache as considered above can be used (but perhaps not be kept consistent with main memory).

Stack processors are usually the targets of stack-based languages, like Forth [Koo89], or Java bytecode for Sun's PicoJava [MO98][Sun97]. The addressing is even more restricted than with a normal register set; extra instructions are required to shuffle operands around on the stack which would not be needed with a register-based architecture – the 'stack overhead'. A stack-based design was considered for Jamaica, but rejected for this reason [WWEM99]. On the other hand, zero-address instructions can allow dense encodings [AP98]. Stack processors are therefore more appropriate for embedded applications where low power and low memory requirements are important, and performance is less significant.

3.3 Register windows

Windowed register files attempt to use the stack properties (easy allocation, reduced register saving on function calls) whilst retaining the advantages of a flat register file (easy addressing) and allowing a large number of registers to be used with no increase in the number of bits required in the instruction encodings. They have shown a clear benefit in previous object-oriented systems, where method calls are frequent [Ung87].

3.3.1 Sun SPARC & Berkeley RISC

In the Berkeley RISC design, which was later adopted for the SPARC, a large[1] register file is split into fixed-size windows, of eight registers each[2]. The register field in instructions is interpreted as relative to the contents of a Current Window Pointer (CWP) register (Figure 4), apart from a single Global

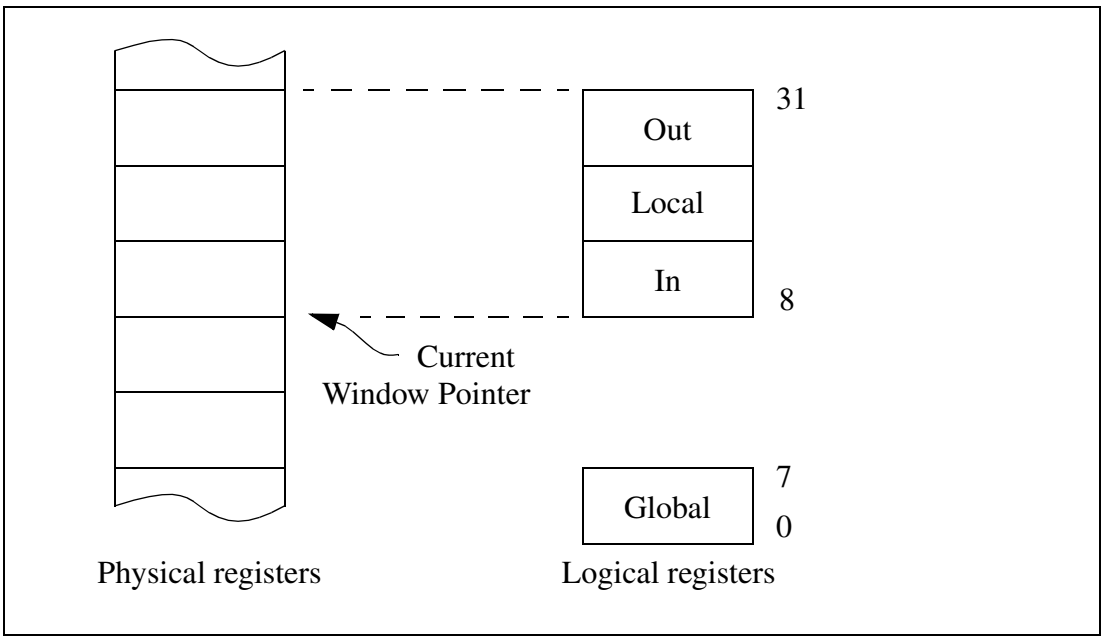


Figure 4: Berkeley RISC register windows

window which remains constant.

The other 24 visible logical registers are divided into In, Local and Out windows. On a function call (`SAVE` instruction) the CWP is increased by two windows; this makes the new In window map onto the previous Out window,

1. Of implementation-defined rather than architecturally specified size.
2. The original documents refer to overlapping windows of 16 registers each; here, I use the Jamaica terms.

with two new empty windows available as Locals and Outs (Figure 5). The

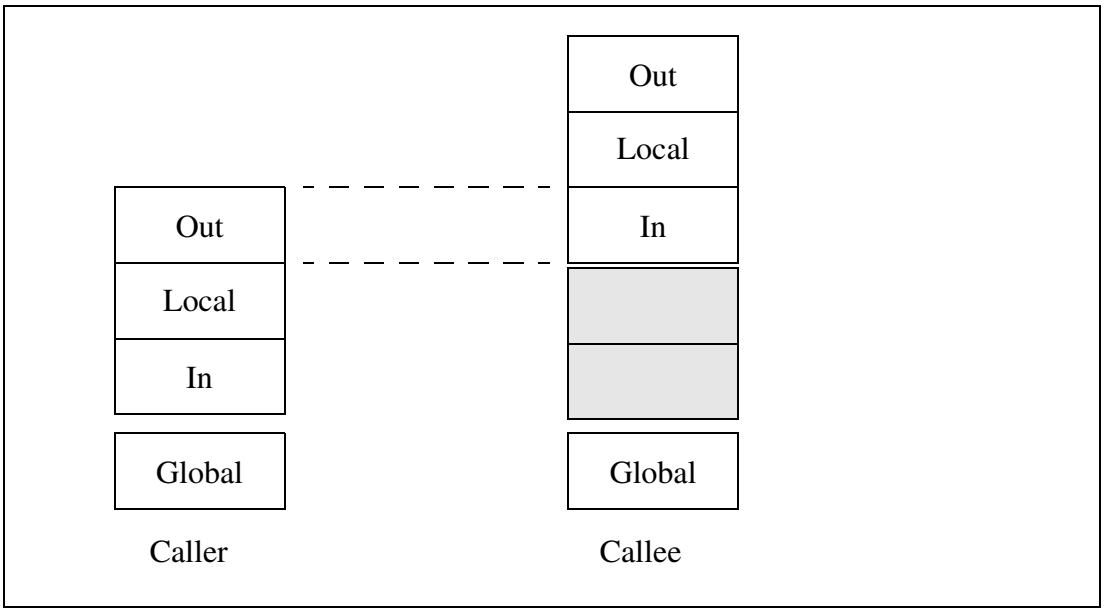


Figure 5: Windows on a function call

register file is logically circular; window spilling or filling is managed by a software trap handler.

In this design, the calling convention is fixed in hardware; the In and Local registers of the caller are automatically saved. Registers do not have to be spilled and filled individually, reducing both instruction count and time required for each function call. The possible disadvantages of register windows are: the larger register file and increased complexity of decoding (adding the CWP to the immediate register field[1]) may slow the cycle time of the processor, the cost of spilling registers when the hardware register file overflows, and the amount of state maintained inside the processor may increase the context switch time. Usually, context switches are infrequent, so the latter is negligible.

Two multithreaded machines based on the SPARC have chosen opposite solutions for combining multithreading with its register windows. The MSparc processor [MD96] replicated the register file, so that each context had a separate circular buffer. The Sparcle processor [ABC+95] from the MIT Alewife instead used the existing windowed file to hold a flat register space for each context (so that the CWP selected the context), and abandoned windowing within the contexts.

1. The Berkeley RISC II used AND-OR decoding rather than addition [Kat85].

3.3.2 Multi-windows

The MULTRIS project [QDT88][QMT89] proposed a more flexible register window organisation, where the available frames, rather than being managed as a circular buffer, are allocated arbitrarily from a 'free list'. This allows several threads to reside in the processor at the same time and allocate their register windows from the same physical register file; the design was targeted at real-time applications requiring fast context switches, and is also appropriate for multithreading. Also, certain common data structures (stacks and queues) could be allocated directly in the register file and used for communication between threads. Multi-windows [SS95] are based on this design, but improve it by simplifying the control information.

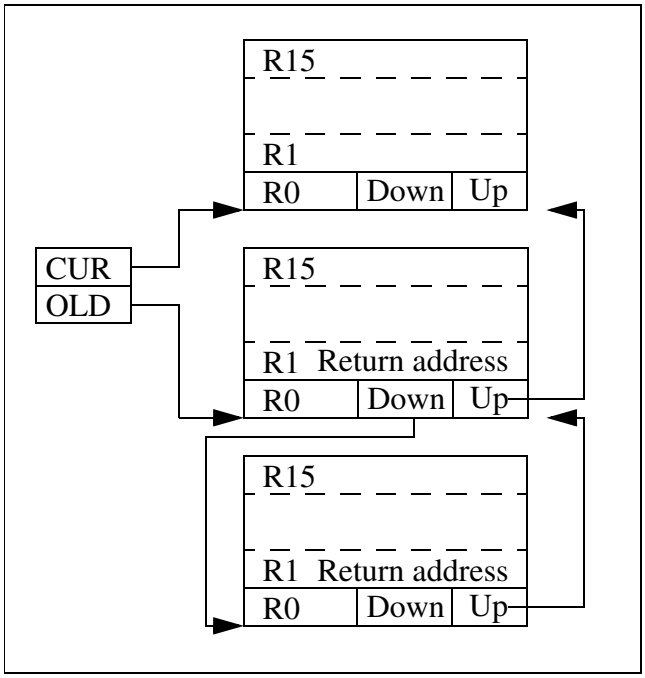


Figure 6: Multi-windows

Figure 6 illustrates how the frames are managed. Each thread has four windows visible at once: one each of 'global' and 'extra' (not shown), and two which contain the procedure stack. The 'cur' window corresponds to the Berkeley 'outs', and 'old' to 'ins'. There are no separate locals. The stack is maintained as a doubly-linked list in the first register of each window.

A function call allocates a free frame and links it onto the end of the list by writing the down and up fields of R0 appropriately. As an optimisation, the down link of the newly allocated frame is written with its up link on the next function call, since the 'old' field holds the back link anyway. The return address is also written. Notice that this requires two register writes, a disadvantage of this implementation.

The advantage of heap-allocation is that threads which can make best use of the registers (i.e. those with frequent procedure calls) may allocate more. The alternative static design (like MSparc) has an independent register file per con-

text; any individual thread is therefore restricted to a fixed fraction of the registers, and if not every context is occupied then some registers will go unused.

3.4 Conclusions

This chapter has considered three different register organisations: a flat register file (as used in most modern machines), a register stack, and register windows. Of these, register windows offer the smallest code size and memory traffic when programs use frequent method calls, which is important for object-oriented languages like Java. Multi-windows provide a flexible way to share a single register file on a multithreaded processor. The Jamaica processor therefore adopts a structure similar to Multi-windows; it is described in detail in §5.1.

CHAPTER
4. Memory systems

The previous chapter examined the small amounts of storage directly accessible within the processor; the next consideration is the main memory. The Jamaica system is a shared-memory multiprocessor, meaning that communication between CPUs takes place through a single memory space. The memory system must take into account this communication as well as simple storage.

4.1 Main memory

The main memory of a computer has many conflicting requirements:

- Capacity: it must be large enough to hold the working set of all applications, since swapping out to disc is very slow.
- Density: the memory must be physically reasonably sized, and of low power and cost.
- Bandwidth: it should supply sufficient quantities of data to keep the processor occupied.
- Latency: the memory should respond rapidly when a request is made of it, to minimise idle time when waiting for the reply.

The first two are driven by the requirements of the application and the costs (in money, volume, and power consumption) of the system; to date, the market has demanded maximum capacity at minimum cost, with the result that single-transistor dynamic RAM is almost universal. These considerations will not

be discussed further; it is assumed that 'sufficient' memory is available at a reasonable price.

The final two determine the computational performance of the resulting system. Bandwidth must be sufficiently high and latency sufficiently low that the processor does not stall. Latency is determined by memory technology and interconnection delays; improving latency is very expensive. In a system with multiple outstanding requests, the latency observed for any particular transaction can be increased by contention or bandwidth limitations. Total memory bandwidth can be increased by widening the path to memory (which includes using multiple memory banks), or operating the path at a faster rate. Caches, i.e. small, fast memories interposed between the processor and main memory, can improve both average bandwidth and latency and also allow the use of higher bandwidth connections.

4.1.1 Caches

A processor running an application with no memory stall cycles will request a certain number of memory loads per second; say 100 million 4-byte words, for a bandwidth requirement of 400 million bytes/s. A main memory with a cycle time of 100ns can respond to only 10 million word requests, for a bandwidth of 40 million bytes/s.

Caches rely on two kinds of locality of reference which may be present in programs. If data which have been referenced once are soon used again, the program exhibits *temporal locality*; the second and subsequent references will (hopefully) hit in the cache. A cache with a 2ns cycle time could provide up to 2000 million bytes/s of bandwidth in this way.

Spatial locality occurs when a program references data which are 'close' to each other. It makes it advantageous to fetch more than one word from memory in the same request: the nearby words are also stored in the cache, and their use will save a memory access which would otherwise have been required. A multi-word cache line can take advantage of wider memory paths – e.g. a 16-byte wide bus could fill a four-word cache line in a single 100ns cycle, for a memory bandwidth of 160 million bytes/s. If these adjacent words are

never used, this extra bandwidth has been wasted. Filling multi-word cache lines and hoping for spatial locality is a form of speculative prefetching.

Notice that caches (or line buffers, effectively a form of small cache) are necessary in order to take advantage of memory paths which are wider than the data demanded by the processor in a single request.

4.1.2 Multiple outstanding requests

Another approach is to tolerate latency by allowing multiple outstanding memory requests; this requires that the processor support out-of-order execution, multithreading, or multi-word (vector) instructions. The main memory may be internally pipelined, or divided into separate banks to which independent transactions are directed. Two-way pipelining and dual banks are illustrated in Figure 7 and Figure 8 respectively. The Control lines transmit the

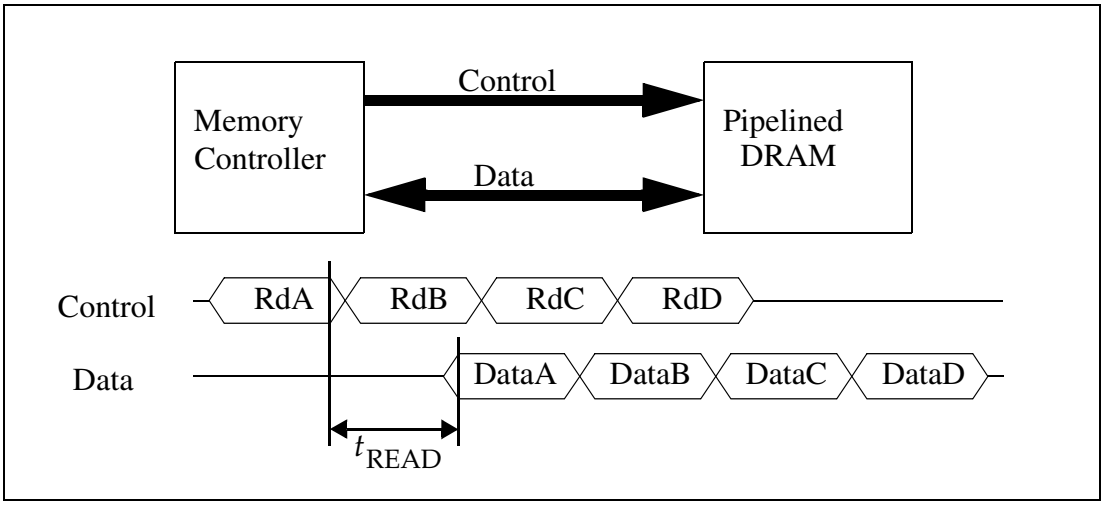


Figure 7: A pipelined memory system

address required and any other command signals (for example, whether to read or write). In each case, four read operations are illustrated, with two in progress at once. The pipelined system can commence the next operation before the previous one has completed; the banked design can perform operations simultaneously, but only when they are on different banks (which depends on the addresses requested; some mapping of addresses to banks is used). Pipelining requires more complex control within the memory modules, but reduces the number of connections (IC pins, circuit board tracks) within the machine as a whole.

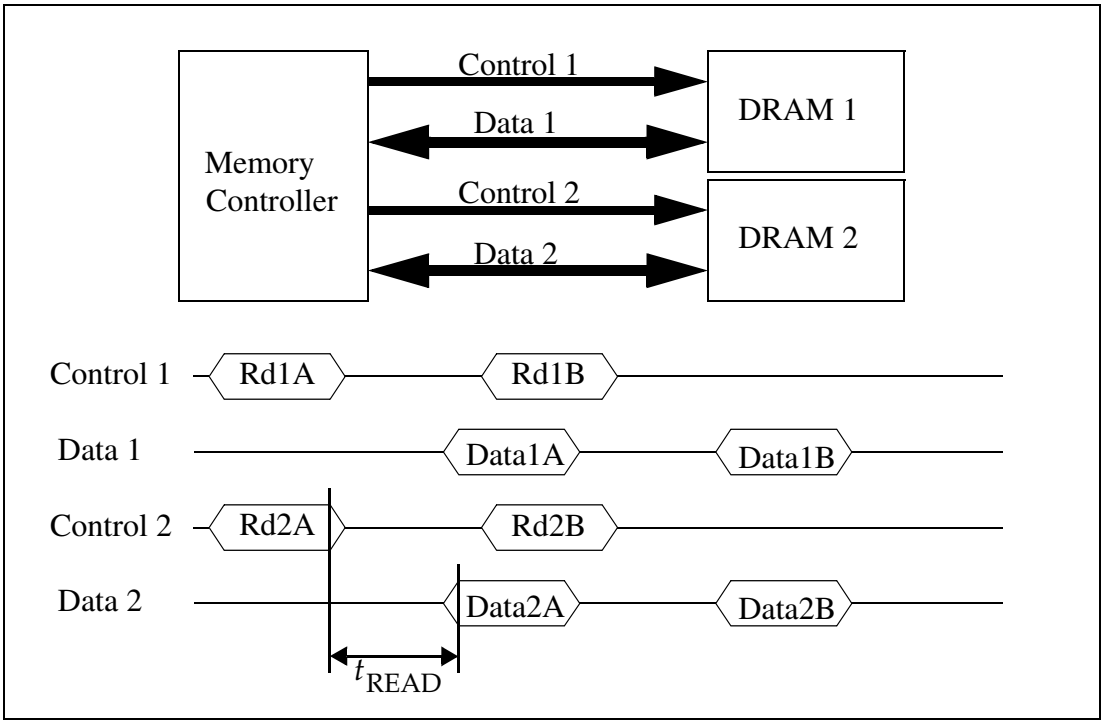


Figure 8: A banked memory system

If caching is used with multithreading, a single shared cache generally performs better than a separate partition per thread [CGL94]. The cache must allow other threads to proceed whilst one is waiting on a cache miss, i.e. it must be lockup-free [Kro81].

The Direct Rambus system connects internally-banked DRAM devices [Ram99] using pipelined control and data buses (termed a *channel*) [Cri97], and offers the highest peak bandwidth currently available from commercial DRAMs: the 16-bit wide data bus operate on both edges of a 400 MHz clock, giving a peak bandwidth of 1.6×10^9 bytes/s. However, the latency is larger than that of more conventional synchronous DRAM, which reduces its performance advantage [CJDM99]. Nevertheless, high-bandwidth pipelined memory is exactly what is required for a multithreaded processor, so the Jamaica simulator models Rambus DRAMs (§7.5).

4.2 Multiprocessing and buses

A multiprocessor system behaves, to the memory, like a multithreaded single processor: a stream of interleaved requests is generated.

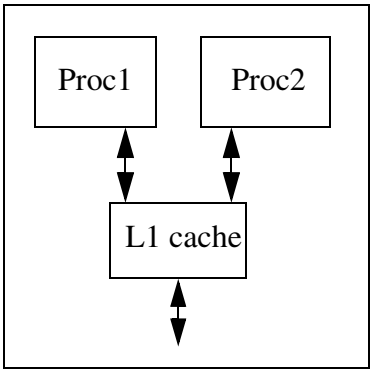


Figure 9: Shared L1

A shared first-level (L1) cache is conceptually the simplest design; the entire memory system is shared, no coherence problems exist, and communication between processors occurs at L1 speed. Unfortunately, the cache must be multiported (costing area and speed) to avoid contention between the processors. The distance between the cache and processors will also increase latency, compared to a private cache. Shared L1 is therefore only appropriate for very small scale multiprocessing (e.g. two processors, like Sun's MAJC 5200 [Sud00]).

If the sharing occurs further from the processor (e.g. shared L2, or only shared main memory) then a bus or point-to-point network must be interposed to maintain the single memory space. If private caches are used then the problem of coherence is introduced, and the bus protocol must solve this. If the bus is further from the processor, then the intermediate caches will reduce bus traffic and contention, at the expense of increased interprocessor communication time. Studies have shown that a shared L2 cache provides the best overall performance for a single chip multiprocessor [NHO96].

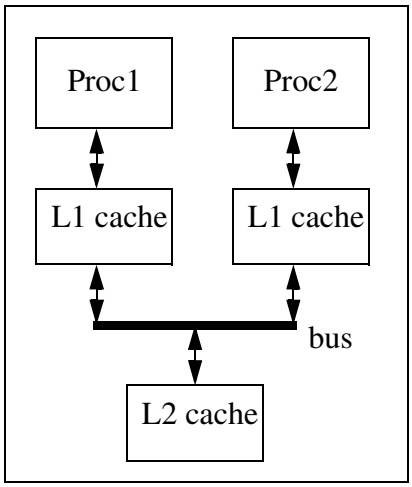


Figure 10: Shared L2 cache

4.2.1 Coherence protocols

From now on, a shared bus will be assumed, for simplicity of implementation.

The design space of coherence protocols is large; the main aspects are:

- The consistency model: sequential consistency means that the memory behaves as it would on a time-sliced uniprocessor. Relaxed consistency may allow optimisations (for example, write buffering) and potentially greater performance [AH93] [AG95][CH95].
- Write-through vs. write-back, at each level of the cache hierarchy.

- Write-allocation: whether a write miss causes a line read (the alternative is to write through).
- Whether cache to cache transactions are allowed, or all have to go through memory. If they are allowed, whether memory is simultaneously updated.
- The state of each line: common protocols distinguish some subset of Modified, Owned, Exclusive, Shared and Invalid (MOESI) [SS86]. M or E means that the line is writable, since it is the only visible copy of data present in the system. M or O means that the cache is responsible for writing the data back to main memory.
- Whether control information (the state of cache lines) is distributed through the system, or centralised.

Some existing protocols are briefly reviewed – a good survey is [AB86], and it is from here that the definitions of Berkeley, Firefly and Dragon are taken.

Illinois

A typical writeback protocol was invented at the University of Illinois [PP84]. Any cache line is in one of four states, the MESI subset of those mentioned above[1].

A read transaction can be answered either by another cache or by the main memory. If another cache has the data, it provides them and sets its own state to S or I, depending on whether the requester wants the line to be exclusive. Memory is updated if the line was modified. If the memory provides the data (i.e. no other cache has them), then the requesting cache receives them in state E; other caches which may have been requesting the same line are not satisfied in the same transaction. Writing to a non-exclusive line causes an invalidation in other caches. A writeback goes straight to memory, and also does not answer pending requests by others.

Berkeley

The Berkeley protocol uses a different subset of states, namely MOSI. It differs from Illinois in that cache to cache transfers are allowed without updating memory, hence the dirty-shared state O. Unfortunately, only an owned line

1. The original Illinois paper defines the S state as being consistent with main memory, but this is not true in general for a MOESI protocol [SS86].

can be supplied in this way, meaning that memory must still supply the line in the case where it is only shared (S) in other caches. Berkeley is inferior to Illinois in its treatment of exclusive lines, since it cannot detect exclusivity on read misses. Thus, the first write to a line by a particular cache will always cause an invalidation.

Firefly & Dragon

These two protocols are very similar, and use write-broadcasting rather than invalidation. Firefly is the simpler. It uses the MESI states, and treats read misses in the same way that Illinois does. When writing, if the line is shared, the write is broadcast on the bus to other caches and memory; the other caches then respond to say whether they still hold a copy, and if not then the original requester changes state to exclusive. A further write would change the state to M rather than being broadcast. A write miss is treated like a read followed by a write.

Of particular interest for CMP design is the implementation of Firefly. Both it and Illinois allow any cache holding a line to supply it, but whereas Illinois arbitrates between them for a unique respondent, Firefly instead fixes the timing and all caches respond at once. The fixed timing is easy to arrange in a CMP, and so Jamaica adopts this approach (§7.2). In the past this choice between arbitration or fixed timing for cache to cache transfers has, according to Archibald & Baer, "led some designers to conclude that it is more efficient to obtain the data from memory whenever possible." [AB86]. This is the solution used by Berkeley, but the same assumption will not hold for a CMP.

The Dragon protocol finally uses all the MOESI states, acting like Firefly with Berkeley-style cache to cache transfers (but only for dirty lines). Write-broadcasts only update caches and not the main memory.

CRAC

The CRAC write-update protocol [TTKS96] was developed for a single-chip multiprocessor, and fixes some of the problems with the previous protocols by taking the best features from each. In particular, it has Dragon's five states and delayed writing to memory, and Firefly's ability to transfer clean lines from one cache to another. It also introduces a new capability: when an Owned line

is replaced, the responsibility for writing it to memory (i.e. the ownership) can be transferred to another cache already holding the line, so that an external memory write is not generated.

The choice of write-update achieves fewer cache misses at the expense of increased bus traffic (over write-invalidate). The processor for which CRAC was designed had a target clock rate of "over 500MHz", and the simulations in the paper assumed a 16 or 32-cycle memory latency with no second-level cache, and a four-cycle bus transaction time. A relatively small number of processing elements (up to 8) was simulated.

These parameters are quite unlike Jamaica's, where a larger number of processors must be supported with a greater memory latency and transaction time (§1.3); since the processors are multithreaded, they will each have greater bandwidth requirements too. However, Jamaica does have on-chip L2 cache, so many L1 misses may actually be cheaper than assumed for CRAC. Jamaica's protocol therefore uses write-invalidate, to reduce bus traffic and increase scalability, even if this increases L1 cache misses.

The CRAC protocol was implemented using a central coherence unit (CCU), a single controller which checks the (dual-ported) tags for each cache simultaneously. This centralised control is the reason for the four-cycle transaction time. However, given future chips' wire delays, this is also inappropriate for Jamaica, and the more usual distributed control is used instead.

To reduce the number of bus transactions, a single data response in Jamaica can answer several outstanding requests for the same line, a feature of no protocol here. This should be particularly valuable in a multithreaded machine where there may be a large number of outstanding transactions for the same line, particularly in the case of contention on a single lock. The details are in §7.1.

A recent survey considers several other enhancements to the Illinois protocol [Mil00]. Some of them rely on the software's communicating more information about its intentions, e.g. by loading or prefetching data in an exclusive state, to save a later invalidation. Others change the coherence properties based on heuristics: for example, passing shared data may be allowed to update an invalid line (known as *read snarfing* [AB95] or *read broadcasting* [EK89]),

or new states may be added to identify migratory data [DSS95]. These techniques are not considered further for Jamaica, but could be the object of future studies. In particular, the Jamaica system does not suffer from some of the problems which reduced the effectiveness of read broadcasting in Eggers's study.

4.2.2 Split transactions

In the uniprocessor illustrations, the DRAM devices were purely passive; the memory controller has knowledge about their timing characteristics, and asserts the appropriate control signals when necessary. The memory system as a whole (i.e. caches, memory controller, buses and DRAM) has much more complex behaviour; some requests will complete in different times to others. A high-performance system must allow other operations onto the bus in between a memory read request and the data reply. This is accomplished with a *split-transaction* protocol, where the memory interface becomes an active participant on the bus.

Each request on the bus must be tagged somehow, with a serial number or requester identifier; when the memory is ready to respond it gains control of the bus and provides the tag so that the reply can be matched against the request.

4.2.3 Inclusion

A (two-level) cache hierarchy is termed *inclusive* if every line present in the L1 cache must also be present in the L2 cache [BW88]. The advantage is that coherence traffic (for example, from I/O devices, or other processors not sharing this L2 cache) need check only the L2 tags.

Maintaining inclusion means invalidating corresponding lines in the L1 cache when a line is removed (replaced or invalidated) from the L2 cache. The problem for the shared-L2 CMP is that the combined first-level caches can act like a single highly associative cache; for example, assuming 16 processors with 4-way associative L1 caches, there may be up to 64 entries which map to a single L2 line[1]. Hence, if the L2 is less than 64-way associative, then maintain-

1. This is assuming that the L2 has at least as many lines as the L1, which will usually be the case.

ing inclusion may require invalidations of large numbers of L1 entries. With a write-back protocol, this causes needless thrashing to memory; a write-through protocol would perform more operations across the bus. In either case, the efficiency of the L1 cache is reduced.

The Jamaica processor therefore does not maintain inclusion between the L1 and L2 caches; some consequences of this are discussed in §7.4.2.

4.3 Conclusions

This chapter has briefly examined modern memory systems, caches and protocols suitable for building a single-chip multiprocessor. Direct Rambus DRAM is adopted as the highest performance main memory technology available; its pipelining and internal banking make it a good choice for a multi-threaded machine, where many requests can be handled simultaneously. A shared bus will connect the processors between the L1 and L2 caches. Jamaica's cache coherence protocol fills in some gaps in existing protocols.

The Jamaica simulator models in detail a memory system meeting these requirements; it is described in Chapter 7.

II. The Jamaica architecture & simulation

CHAPTER

5. The instruction set architecture

An initial study examined several Java programs to determine their register (stack and local variable) usage and function call patterns, and concluded that a conventional register-register RISC instruction set with register windows would provide the best performance[1], compared with a non-windowed file or a hardware stack [WWEM99]. The Jamaica instruction set is based on an existing RISC design: the Digital Alpha [Dig92]. Jamaica has, however, only a 32-bit word length (suitable for most Java data types), so the 64-bit Alpha instructions are omitted. The full instruction set is listed in Appendix B.1.

The significant modifications are to the function calling mechanism and register set, and in the provision of multithreading and thread distribution. This chapter presents the new features at the abstract architectural level; Chapter 6 describes how the simulator models their implementation.

5.1 Function calling and register windows

The register window[2] structure and function calling is based on Multi-windows (§3.3.2 and [SS95]). However, the windows are reduced to 8 registers each (to agree with the Alpha's 32 visible registers), and the provision of stacks

1. This study used a relatively simple translator from Java bytecode, and did not perform function inlining or complex register allocation. Given a more sophisticated compiler the performance advantage of register windows would be reduced.
2. Although a formal distinction will not be made, this document uses *window* to mean the abstract programmer-visible registers, and *frame* for the underlying set of registers which implements a particular window. Cf. *thread* and *context*.

and queues is abandoned on the grounds that high-level Java code is unlikely to use these features. Also, the linking information is no longer kept in the first register of each frame; this state is not required by user programs, and merely compromises security and stability.

Up to four windows are visible at any one time: they are named Global, In, Out and Extra, and named `%[giox][0-7]`. The Global registers belong to a particular thread context, and are not switched on a function call. Register `%g0` is hard-wired to zero. The Ins and Outs act in the usual register-window fashion, Ins being automatically saved on a call, with Outs then being mapped to become the new Ins. The Extra window normally acts as more global registers; however, some system threads may have to inspect the call stack of user threads, for example when handling an exception or garbage collecting. In that case, the intention is to allow the system thread's Extra window to map onto one of the user's windows, and be moved up and down the user's call stack[1].

The depth of the call stack may exceed the number of frames available in a particular implementation, so some means must be provided for spilling to and filling from memory. The SPARC uses a software trap handler for this [WG94]. Jamaica instead performs it in hardware; the runtime system must designate a *spill area* as part of a thread's context. If a spill area overflows (or underflows, since there may be more than one per thread) then an exception is taken so that the runtime software can extend it or provide a new area. The hardware to perform this spilling and filling is shared with the task distribution mechanism.

Figure 11 gives a logical view, with Figure 12 showing the underlying mechanics. The former illustrates the window stack; the frame numbers are marked at the right side to make it easier to relate to the underlying structure. The latter shows how this is implemented; the `next` links and some unused data fields have been omitted for clarity.

5.1.1 Function calls

The SPARC decouples the actual call of a subroutine (`CALL`, `JMPL`) from the shifting of the register window (`SAVE`). This allows a leaf function (i.e. one

1. The simulated system does not yet require these capabilities, as the RTS does not use them, so they have not been implemented.

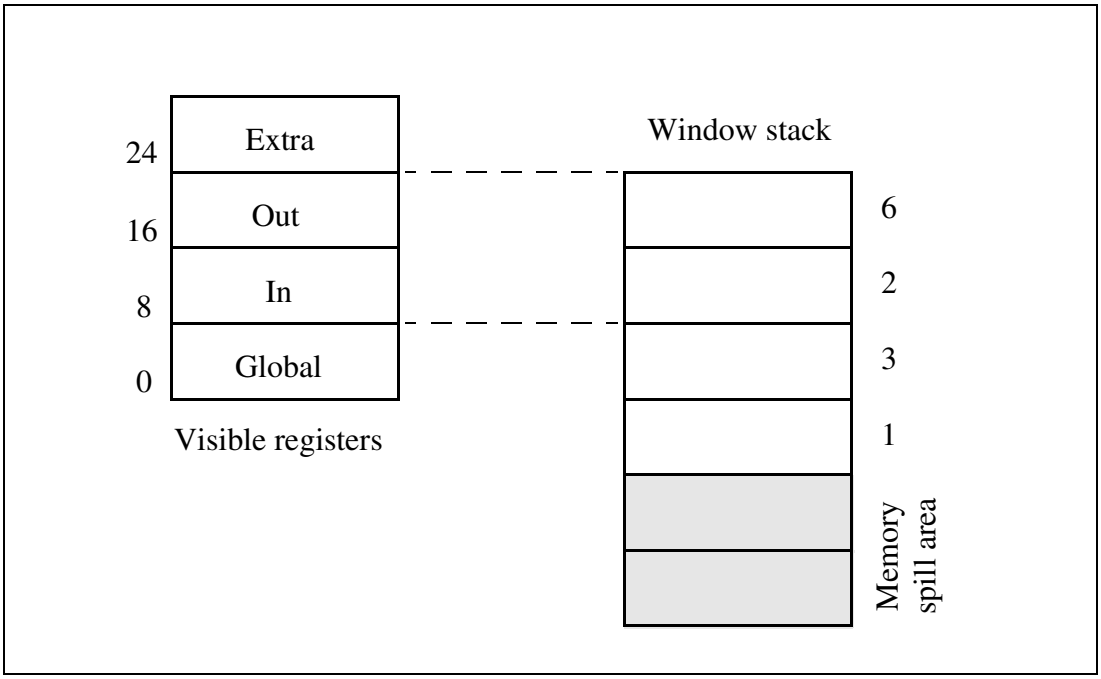


Figure 11: Window structure (logical)

which performs no further calls) not to use a new window if it can instead operate directly on the Out registers, thereby saving a possible window spill. The compiler must detect leaf functions statically and change its register allocation for them.

Jamaica, instead, always shifts the windows on a subroutine call (`JSR`, `BSR`). However, the allocation of a new Out frame can be delayed until an Out register is first referenced (*lazy* allocation); leaf functions may then operate directly on the In registers without requiring a new frame. This has the advantage over the SPARC method that the compiler does not have to detect leaf functions or modify its register allocation, and indeed invocations will also benefit which are dynamically but not statically leaves (for example, of a function which performs a test on its arguments and may either return immediately or call a second function).

A function call (`JSR`, `BSR`) to destination D proceeds as follows:

1. Check that an Out window is allocated; if not, allocate one.
2. `In_frame := Out_frame`
3. `Out_frame :=` (none)
4. `%i7 := PC+4`, saving the return address
5. `PC :=` D

The Out frame will be allocated when an instruction first references it.

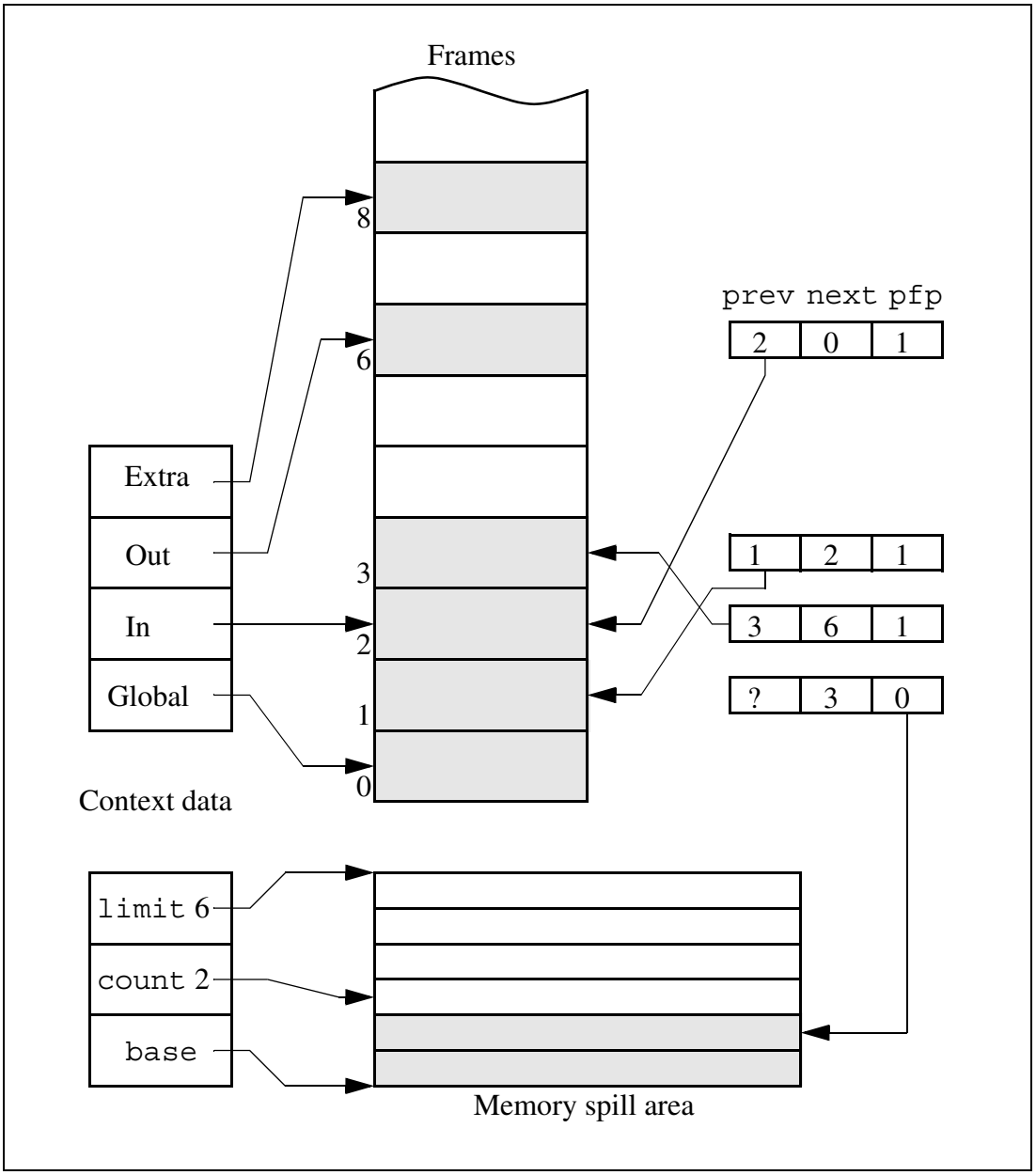


Figure 12: Window structure (underlying)

A return (`RET`) works in exactly the opposite way, except that a thread will terminate if it attempts to return from its first window:

1. If an Out window is allocated, free it.
2. If the In window is the first of this thread (§5.1.2), terminate the thread. Otherwise,...

3. `PC := %i7`

4. `Out_frame := In_frame`

5. `In_frame := In_frame.prev`, filling from memory if necessary (§5.1.3).

5.1.2 Window management

To maintain the doubly-linked list of frames within a thread, several fields are stored in a separate data structure (shown at the right side of Figure 12). Each frame has the following information:

- `prev`: the index of the frame belonging to this function's caller, i.e. the frame previously allocated to this thread.
- `next`: the index of the frame belonging to this function's callee, i.e. the frame allocated after this one.
- `pfp`: previous frame present, a single bit, set to 1 if the previous window is resident in the processor. If the bit is 0 then returning from this frame will require a fill operation.
- `alloc`: a single bit, 1 if this window is allocated, 0 if it is free.

The combination `prev = 0` and `pfp = 1` indicates that the frame has no caller, so the thread should terminate when this function returns[1].

To allocate a new frame as a successor to frame `P`:

1. Find a frame `F` such that `F.alloc = 0` (presumably by removing it from a free list). If none exists, a window must be spilled first.

```
2.F.prev := P ; F.next := 0 ; F.pfp := 1 ; F.alloc = 1
3.P.next := F
```

Similarly, to free the frame when returning from F:

```
1.F.prev.next = 0
2.F.alloc = 0
```

Each of these operations requires a write to the control fields of two frames. In Multi-windows, since the links were held in the main register bank, an extra write port was needed [SS95]. Jamaica's links are held in a separate structure, so this is not nearly as expensive. One advantage of the SPARC's circular window management is that only adjacent windows may ever be referenced simultaneously; this allows a physical layout where bit cells are packed more tightly within the register file [TJS95].

1. In the simulated system, frame 0 is always a global window, and hence cannot also be a previous frame. Any distinguishing value would suffice.

5.1.3 Filling and spilling

The spill area, part of a thread's context, has four parameters:

- `base`, a (cache-line aligned) memory address for the start of the area.
- `limit`, the maximum number of entries allowed in this area.
- `count`, the index of the next free entry within the area.
- `last`, a single bit indicating if this is the final spill area for this thread.

To spill a frame `F`, which must be the bottom resident window of a thread:

1. If `count >= limit`, then raise an exception since the spill area is full.
2. Write `r0..r7` out to memory, starting at address `base + 32*count`.
3. `count := count + 1`.
4. `F.next.pfp := 0`.
5. `F.alloc := 0`.

Then to refill frame `F`, when frame `N` is the current bottom frame of a thread:

1. If `count = 0`, then raise an exception as the spill area is empty.
2. `count := count - 1`.
3. `F.next := N ; F.alloc := 1`.
4. If `count = 0` and `last = 1`, then `F.prev := 0 ; F.pfp := 1` otherwise `F.pfp := 0`.
5. `N.prev := F ; N.pfp := 1`.
6. Read `r0..r7` from memory, starting at address `base + 32*count`.

The `last` flag allows the runtime system to use several non-contiguous save areas for the same thread; only if the final area has just been emptied should the thread terminate on this frame's return, and the handler should set the flag appropriately.

5.2 Multithreading and thread distribution

Each processor in the system can hold several (up to `T`) threads; the resident state of a thread is termed a *context*. The context structure, indexed by a number from `0` to `T-1`, holds the following information:

- The program counter (`PC`).
- State flags, indicating whether the context is runnable:
  - An `empty` flag, set when the context is free.

  - A `memwait` flag, set when the context is waiting for a memory operation to complete.
- Pointers to the Global, In, Out and Extra windows.
- A spill area in memory for window spilling (§5.1.3).
- A token (§5.2.1), if one is allocated to this thread.
- A 32-bit *thread ID* for this context. The processor does not interpret this information directly, and the runtime system can use it as it wishes.
- Flags which modify the context's behaviour:
  - Interrupt mask.
  - Release token on exit.
  - Handling exception.
  - Low priority.
- A 32-bit *irq_flags* register, indicating pending interrupts for this context.
- Some extra internal state, such as the saved program counter during interrupts and exceptions.

A context is available for execution when the state flags are clear. The exact method of multithreading (the context switching mechanism and policy, for example fine-grained interleaving or switching on a cache miss) is not architecturally specified.

5.2.1 Locating idle contexts: Tokens

As discussed in §1.2.1, supporting efficient fine-grained thread level parallelism will need quick methods for finding an idle context and then creating a new thread in it. To begin with, the former will be considered.

Finding an idle context requires three distinct operations. Firstly, is there such a context available? If not, the requesting thread should continue its serial execution with minimal overhead. Secondly, on which processor is the empty context located? Thirdly, the empty context should be locked, so that another requesting thread does not also find it available.

A simple wired-or signal across the chip, with each processor asserting the line if it had a free context, would satisfy the first point, but not the other two. A query broadcast to all the processors, with an arbitration mechanism (as on a shared bus) to choose one idle context in the event of multiple responses,

could satisfy all three, but would not be fast. A single centralised controller would suffer contention and cross-chip wire delays, and a distributed scheme in which each processor maintains information about all the others would have problems with broadcast state updates and locking.

The solution adopted solves all three problems, and is distributed. An idle context is represented as a single *token*, containing a thread identifier. Possession of the token by another thread confers permission to give work to the named context. Since there is only one token per idle context, the token both locates it and ensures locking.

Tokens are passed around the processors, to distribute the idleness information. A ring structure is illustrated here (Figure 13): each cycle, every processor passes a token on to the processor to its right, and receives a token from the left. Since there are possibly more tokens than processors, a small pool of tokens is kept at each processor, and these are inserted into gaps to keep the ring as full as possible. Each processor's Thread Interface Unit (TIU) manages its tokens.

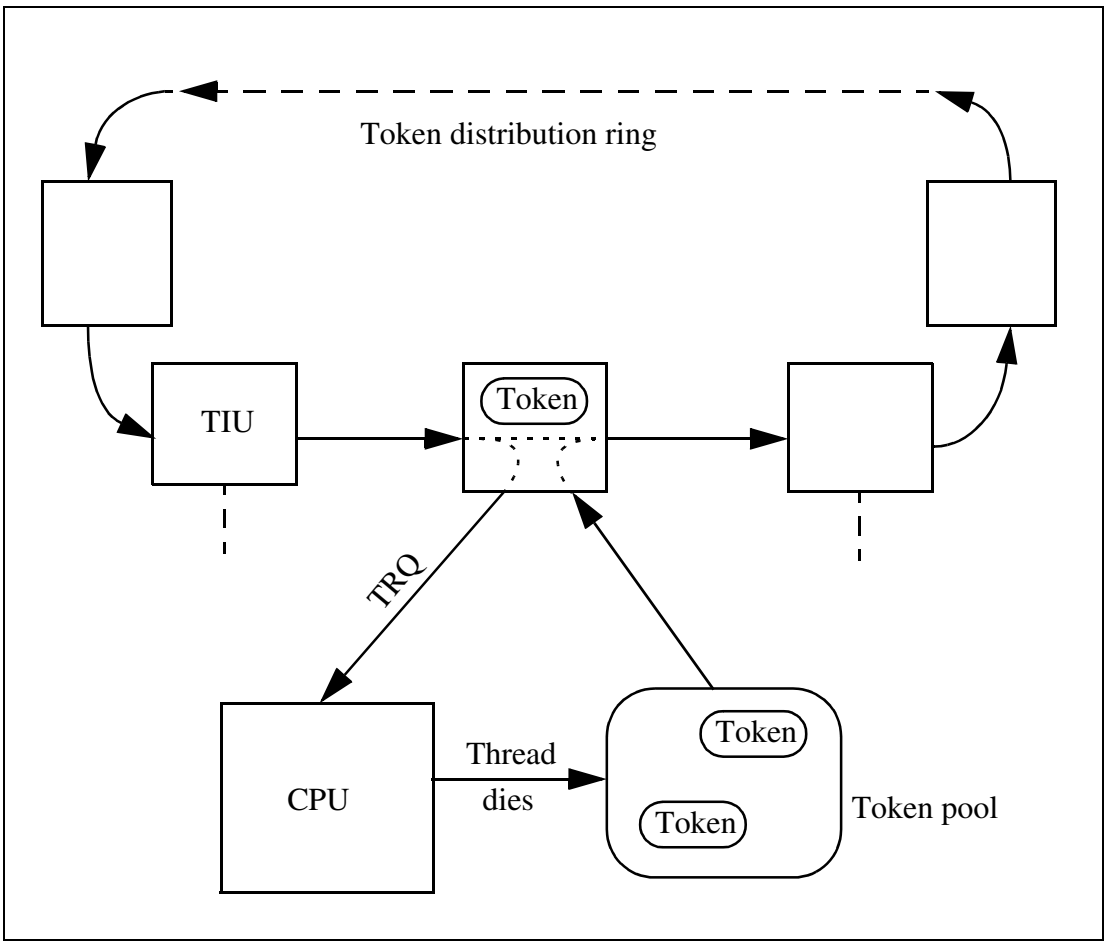


Figure 13: Token distribution by a ring

The token request instruction (`TRQ`) returns 1 if the processor has a token available, and writes that token into the thread's context. (A subsequent `TRQ` will also return 1 but not allocate another token). The thread can now partition its workload in preparation for forking.

When a thread dies, if the 'release token' flag is set, a token is created and inserted into the local pool, with its thread ID taken from the context. Eventually it will find its way onto the ring, and be available for allocation by another thread

5.2.2 Creating a new thread

Having allocated a token and partitioned its workload, a forking thread must start the remote thread and pass its parameters. The fork (`THB/THJ`) instruction is designed to look like as much like a function call as possible: the eight Out registers are copied to the In registers of the new thread, and the PC is set to the argument of the instruction. The destination processor and context is identified by the token belonging to the forking thread. The token's thread ID is broadcast across the bus, and each processor compares the value against the thread IDs of its idle contexts. On a match, it accepts the work.

It is worth mentioning the software stack at this point: by convention, a runtime system will usually designate an In/Out register pair to use as a frame pointer (FP) and stack pointer (SP), for example `%i6` and `%o6`. The window shift on a function call will then make the caller's SP into the callee's FP. Here, the fork/function call analogy must break down, since the parent and child cannot share a stack. There are two possibilities: either the parent can find a stack for the child, place any arguments onto it[1], and then pass the new stack; or the parent can pass its SP, the child extract arguments from it, and then switch to its own stack[2]. The former allows the child to be ignorant of its new creation, but needs more significant changes to the parent. In practice, the Java runtime system uses a native function to fork the thread and a small wrapper on the child, so it adopts the latter solution (§8.5); a fixed set of arguments is passed in registers.

1. If there are more than can be passed in registers.
2. Some locking may be required here!

5.2.3 Performance

Jamaica employs two hardware techniques to help with dynamic thread manipulation: finding idle processors, and sending work to them. This section explains why these two operations are important for effective parallel execution, and hence why hardware support should be considered.

The life-cycle of a token/context proceeds in four stages. Firstly, the token is released and travels around the ring. Secondly, another thread tests for and acquires the token; thirdly, the forking thread must arrange its workload to make use of the token, which includes sending the register contents across the bus. Finally, the new thread works until it terminates and releases the token again.

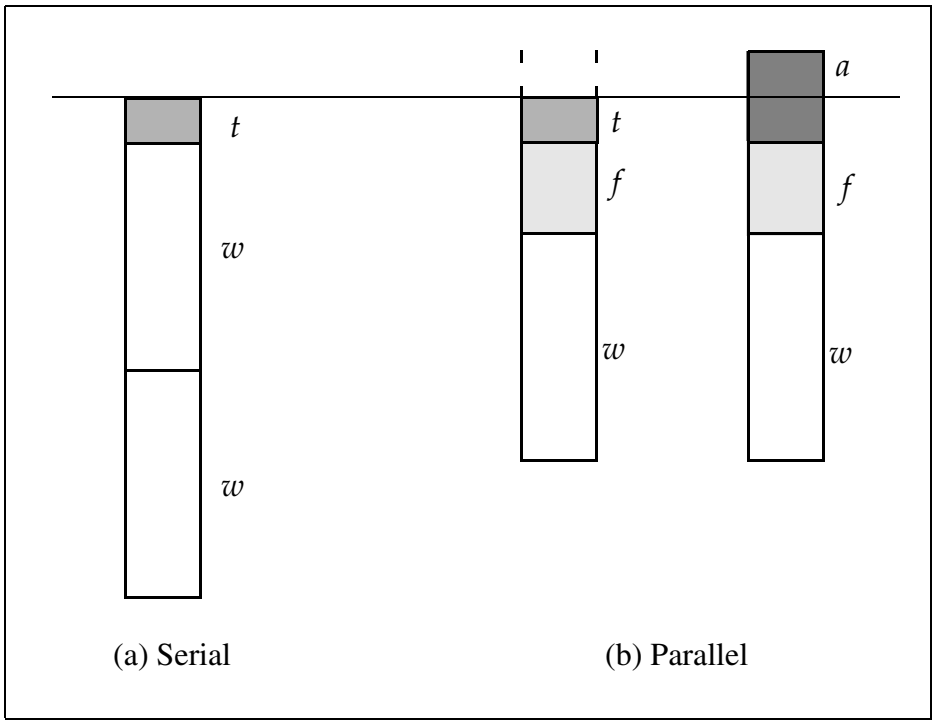


Figure 14: Serial vs. parallel executions

Figure 14 shows outlines of these times[1]; in (a), a single processor executes two work units of time w each after first testing whether one could be forked; the test takes time t. In (b), the test succeeds, and after time f the processors execute one work unit each.

The release–acquire stage (time a in the figure) consumes no processing resources, and thus does not slow down the computation at all. During this time,

1. These values are not independent of one another; for example, increasing w would result in an increase in a if it meant that token requests happened less frequently.

at least one thread (somewhere) is performing useful work, although the creator of the token remains idle; this may be inefficient, but it is not directly harmful.

The second phase, testing for a token, is overhead by a running thread, and occurs whether or not a token is found. In (a), the 'speedup' is $2w/(t+2w) < 1$.

The third phase, acquire–fork, also costs a running thread, but is only paid when a fork is about to take place; this time should therefore be made up once the workload starts executing in parallel. Finally, the fourth phase consists of useful work by both threads. The speedup in (b) is $2w/(t+f+w)$.

Of these four, only the 'test' operation adds to execution time in the worst (serial) case when no token is available; if the available parallelism is much larger than the number of contexts then most of the tests will fail, and potentially parallel code will actually execute serially. This test, then, must be very fast, because it directly slows all executions of the program.

The acquire–fork stage is less critical, but also affects the granularity of work which it is worthwhile forking off: it must take less time than the work (phase four) for there to be any speedup, i.e. $t+f<w$.

The release–acquire stage is the least important, because it never slows down the program; instead, it limits the maximum speedup. The efficiency of the system depends on the ratio of the idle time to useful work. In (b), for example, if the right-hand processor repeatedly releases a token, waits time $a > t$ for it to be allocated and then acts as in the figure, then its utilisation will be $w/(a+f+w)$, and the maximum possible speedup will be this multiplied by the number of processors in the system.

The token mechanism, therefore, seeks to minimise t even at the expense of increasing a (since its inaccuracy could result in missed forking opportunities). This results in the best performance when there is a lot of excess parallelism, so that most token requests are bound to fail anyway. This is exactly the Cilk project's 'work first' principle [FLR98]. The alternative, minimising a, for example by using a more accurate system, may result in better performance when the available parallelism is comparable with the number of processors. When w is large compared to the others, the results approach 'perfection'. Making t frequent reduces a.

Finally, in the presence of multithreading the situation is more complex, because the idleness or otherwise of a context has a less direct effect on processor utilisation. On one hand, it may hide some of the release–acquire time a; on the other, it may result in an increased number of forks without a proportional increase in utilisation. The best degree of multithreading will vary depending on the workload.

5.3 Interprocessor interrupts

Having placed a new thread on another processor, the runtime system may want some way to interrupt it asynchronously. For example, software scheduling and pre-emption, the Java `Thread.stop()` method, and squashing of speculative threads (if supported), all need to gain control of a thread which is executing elsewhere.

The software interrupt request (`SIRQ`) instruction uses many of the same mechanisms as the thread distribution. It accepts as an argument a thread ID and an interrupt number (0–31); the request is broadcast across the bus; any matching context will mark the indicated interrupt as pending by setting the appropriate bit of the irq_flags register.

Two interrupt handlers are used: one for contexts which are idle, and one for those which contain an active thread. The idle handler does not need to save internal state or worry about part-completion of blocked instructions. A return from interrupt (`RTI`) clears the interrupt mask, putting the thread back to the idle state or resuming execution.

5.4 Locked memory operations and WAIT

Following the Alpha instruction set, Jamaica uses load-locked (`LDL_L`) and store-conditional (`STL_C`) instructions for building atomic sequences like test-and-set or fetch-and-add. The `LDL_L` instruction marks a cache line as locked by that context; any other processor's writing to that line before the next `STL_C` will set a flag and cause the store to fail. If the store completes, then to the rest of the system the entire sequence between the load and store appears to com-

plete atomically. The store instruction returns a bit indicating whether it completed; if it did fail, it is usual to branch back to the load and retry the sequence.

In a conventional multiprocessor system, spinlocks will be implemented with these primitives using code like Figure 15 (a slightly edited version of the code from the Alpha architecture manual [Dig92]). On a multithreaded system like Jamaica this sequence would also work correctly, but whilst spinning it is consuming execution cycles which could be available to another thread; at an extreme, if another thread on the same processor is the current owner of the lock, then it may delay the release for which it is waiting. The Alpha code allows only for possible software rescheduling at the `already_set` point, since that is the only other option on a single-threaded processor.

```
acquire_lock:
  LDL_L R1, lock_var
  BNE R1, already_set
  OR R1, 1, R2
  STL_C R2, lock_var
  BEQ R2, stl_fail
  RET
already_set:
  <block/reschedule/test for too many iterations>
  BR acquire_lock
stl_fail:
  <test for too many iterations>
  BR acquire_lock
```

Figure 15: Alpha acquire_lock()

The Jamaica system solves this by introducing a new instruction, `WAIT`. It uses the cache line address previously set by an `LDL_L`, and blocks execution of a thread until another processor writes to that line, presumably to release the lock. The processor will wake up 'accidentally' if a word other that the lock is written in the same cache line; it is recommended that lock flags be placed in cache lines of their own, or with mostly read-only data (the Alpha manual makes the same recommendation). Also, observe that the `acquire_lock` routine never writes to the lock unless it succeeds, so another processor's attempt will not terminate a `WAIT`.

The Jamaica version of the `acquire_lock` routine is given in Figure 29 (§8.3.1 on page 121). Using the `WAIT` instruction, the hardware multithreading may hide some of the latency of acquiring locks.

The implementation of the `WAIT` instruction requires only minor changes to the existing `LDL_L` mechanism.

5.5 Comparisons

Several existing systems were introduced in Chapter 2, as part of the design space discussion. Here, the Jamaica architecture just described is compared with them. Inevitably, comparisons are also made with the memory systems of each: Jamaica's is described in Chapter 7.

5.5.1 Chip multiprocessors

The most obvious comparison is with the Stanford Hydra CMP (§2.1.3). The earlier Hydra papers (e.g. [HO98]) considered modestly superscalar processors; the present Hydra ([HHS+00]), like the Jamaica simulation, uses simpler single-issue pipelines. Multiple-issue is an obvious extension for both. The systems diverge in respect of thread support. Hydra's support for threads is limited to speculation and threads are managed by software ([HHS+00][CO98]), whereas Jamaica focuses on hardware multithreading and load balancing: this should allow it to run much finer grain threads than Hydra manages. However, the Hydra speculation support may allow threads to be created with less programmer effort.

These two designs have much in common, and the thread mechanisms are essentially complementary. It would be an interesting exercise to consider a composite of the two.

The Hydra memory system is similar in outline to Jamaica's: each processor has a private L1 cache connected to a single L2 cache. However, coherence is quite different: their L1 caches are write-through, with an extra word-wide bus to take writes from the processors directly to the L2, invalidating the line in other L1s simultaneously; they also maintain L1–L2 inclusion. The simulated Jamaica system uses a more conventional one-bus protocol, with write-back primary caches, but does not enforce inclusion (Chapter 7). The Hydra write-through bus may become a bottleneck in larger systems. The cache parameters

used are identical, although they were decided independently: 16 KB 4-way associative I and D caches, with a 32-byte line size.

Another CMP is the recently introduced Sun MAJC 5200 [Sud00]. This has two 4-issue VLIW processors sharing a single L1 data cache, with an on-chip graphics processor, I/O, and a Rambus memory controller. The processors support vertical multithreading, with four threads per processor [Sun99].

Again, this system would be a good match for Jamaica's thread management capabilities, and its existing aims are complementary. Its main target is multimedia applications, where inner loops compile well to the VLIW instruction set. Enhanced TLP would enable outer loops to be parallelised too.

The Piranha CMP [BGM+00] is a more aggressive design than either of these. It is based on eight single-issue Alpha processors, and like Hydra has separate I and D caches for each processor sharing a single L2 cache. However, the L2 is divided into eight banks, each of which has a separate memory controller and Rambus channel; the L2 banks are connected to the processors by means of a (logical) crossbar switch. As in Jamaica, L1–L2 inclusion is not maintained; each L2 bank is responsible for the coherence of cache lines which map to it, and holds a duplicate copy of all the L1 tags.

The target for Piranha is commercial workloads – database transaction processing. The designers comment that there is an abundance of thread-level parallelism in such programs. However, there is no hardware support for threads beyond simple multiprocessing.

Aimed at the same market, IBM's Power4 SMP will 'probably' consist of four CMPs mounted in a multi-chip module (MCM) [Die99]. Each chip has two 5-issue superscalar processors with private L1 and a shared L2 cache; the L2s are connected in a ring within the very complex MCM.

Both Piranha and Power4 are designed to have huge memory bandwidths, necessary for these applications; the costs of this, in packaging and interconnections, may be borne by the server market but would be too great for Jamaica's target area. However, transaction processing typically consists of many independent queries, so the TLP comes for free. Neither system offers any support for finer-grained threads. Also, although both have such great bandwidths, the ability to hide memory latency is limited: the Piranha cores are in-

order, and although the Power4 cores support OOO execution this will not hide main memory latency. Weaker memory consistency models may help, as may the Power4's software-initiated prefetching. However, if TLP is so abundant, multithreading is an obvious alternative which places less responsibility on the software.

5.5.2 Rediflow

A comparison of Jamaica with Rediflow is also interesting. Although both have hardware assistance for load balancing, the approaches taken are almost exact opposites. In Rediflow, spare work units migrate around from processor to processor; when a CPU wants a task, it removes one from its queue. In the Jamaica architecture, spare processing capacity, in the form of tokens, is distributed; when a thread wants to fork, it takes the capability from its TIU.

Rediflow, then, assumes that work units are plentiful; given an oversupply of work, the network sees to it that processors remain fully occupied. The cost of creating so many tasks is hidden by the graph reduction model – they would be produced anyway, and the overhead is distributed throughout the computation.

Jamaica, on the other hand, is based more firmly in control-flow, where the default must be not to fork new threads. The assumption is that creating too many is more harmful than allowing some contexts to be temporarily unused: the 'work first' principle again. There are other advantages to distributing tokens rather than work units, not least that their number is known in advance, so that buffer space is not an issue.

5.6 Conclusions

This chapter has introduced the interesting features of the Jamaica instructions set. For improved performance with frequent method calls, as well as easy JIT compilation, register windows are used; for maximum flexibility in a multithreaded processor, the windows are heap-allocated rather than being partitioned statically among the threads.

Fast hardware mechanisms are also provided for finding idle contexts and distributing work to them; an idle context is represented as a token which is passed between the processors. When acquired by a thread, a token grants it permission to send work to the remote processor.

The `WAIT` operation allows spinlocks to be used in a multithreaded environment without taking processor time from other threads.

Finally, the Jamaica architecture was compared with four recent CMPs; although they all use threads for multiprocessing, and two (Hydra and MAJC) have some support for speculative threads, only MAJC is multithreaded within the processors. None offers any assistance for managing threads, which remains the responsibility of software. Jamaica should have a clear advantage for fine-grained threads.

The next two chapters describe in more detail how these features are modelled by the simulator, and then Chapter 8 shows how the system software makes use of them.

CHAPTER

6. The simulated processor

Any architecture is only as good as its implementation, so a Jamaica simulator has been developed in order that the architectural features presented in the previous section can be evaluated. There are low-level design choices, and certain assumptions must be made about a VLSI implementation which can only be assessed by actual circuit design and post-layout simulation. Thus the simulation is intended to be realistic rather than totally accurate.

The simulator itself is written in C, and works mostly at the register-transfer level. That is, every clock cycle, each unit in the system 'simultaneously' examines its inputs and produces an output, which is then latched for use in the next cycle. The dynamic nature of the computation means that a program does not have a unique execution path, so the potentially more efficient trace-based simulation cannot be used.

6.1 Processor outline

Each processor on the chip is identical, and implements the ISA described earlier. A simple single-issue in-order five-stage pipeline, similar to the MIPS R2000 [Kan89] or TOOBSIE [Gol96] is modelled: Figure 16 depicts the main components, although control structures (e.g. the context data) and bypass paths are not shown.

At any given time, one context is identified as current. The instruction fetch (IF) stage works with the Icache to fetch a 32-bit instruction from the address

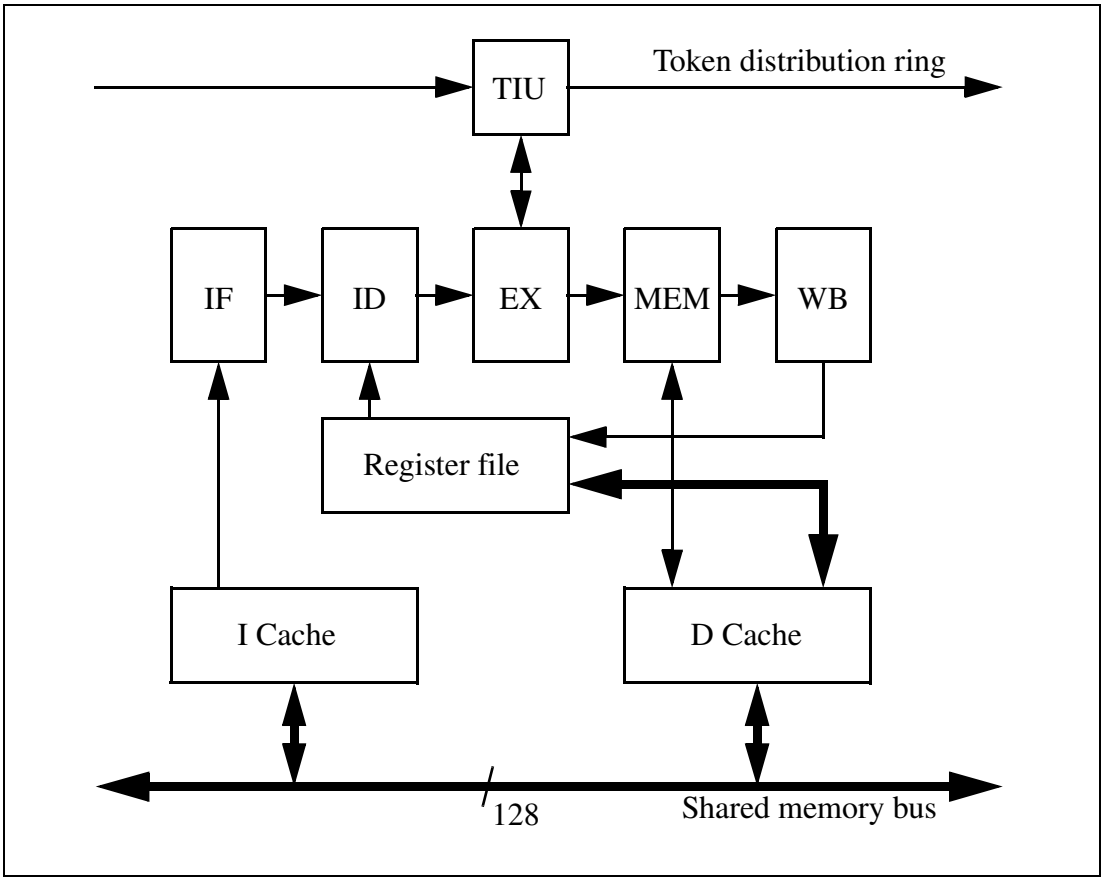


Figure 16: Processor pipeline - outline

indicated by the context's program counter. The IF unit is also responsible for context switching, and will initiate a switch after retrying an Icache miss once, when the quantum timer expires, or when instructed to by the memory stage.

The instruction decode stage (ID) then fetches register operands required by the instructions. It also handles exceptions and pipeline hazards, calculates branch targets, performs branch prediction, and detects unconditional branches and jumps.

The execute (EX) stage contains the ALU, which performs arithmetic, calculates memory addresses, and resolves conditional branches. The memory (MEM) stage communicates with the data cache if needed, and then the write-back (WB) stage commits values back to the register file. It is assumed that the pipeline is fully bypassed, so that following instructions can use results without delay, apart from a one cycle penalty when immediately consuming the result of a memory load (all other results are produced in the EX stage).

In general, every instruction commits its results by changing the processor's permanent state (registers or context data) only when it is guaranteed to complete, i.e. in the MEM stage[1].

The Thread Interface Unit (TIU) deals with tokens and thread distribution, and will be described in §6.6.

6.2 Exceptions and stalls

The pipeline generally advances at every clock tick, passing a normal Jamaica instruction along. The complications arise when something 'out of the ordinary' happens: for example, a data cache miss, or a branch misprediction.

The simulator represents unusual conditions as *exceptions*, which are actually a particular format of instruction passed along the pipeline like any other. It should be emphasised that not every exception represents an error. For example, on an instruction cache miss, the fetch stage passes an `ICACHE_MISS` 'instruction' along to the decode stage, which is ignored by the rest of the pipeline as a no-op. Some exceptions do represent errors, for example `NO_TOKEN`, which is generated when a thread tries to fork without first receiving permission. Raising this exception (in the EX stage) causes the ID stage to be cancelled (the instruction replaced with `EXCEPTION_ANNUL`). The fetch stage then inserts the `NO_TOKEN` pseudo-instruction into the pipeline, and starts fetching from the handler address. This is depicted in Figure 17. This method has the

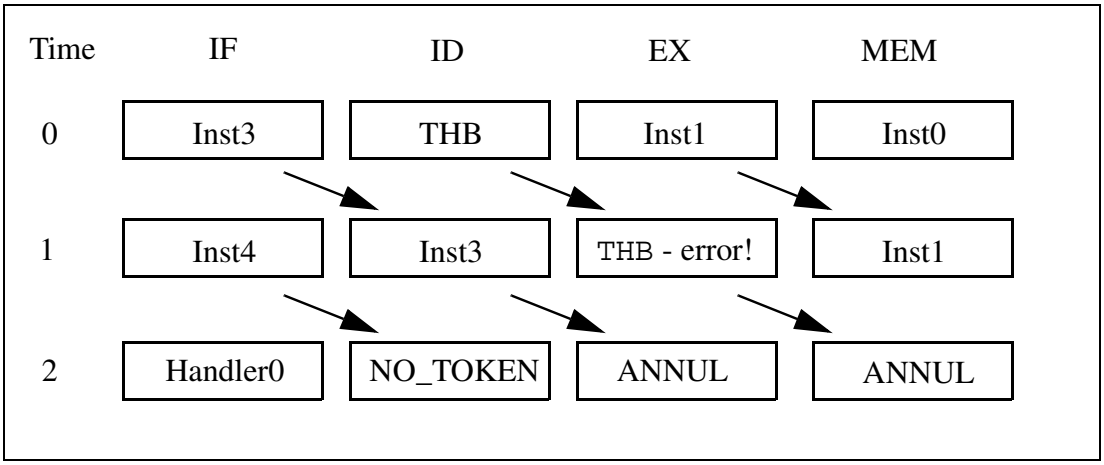


Figure 17: Raising a NO_TOKEN exception

advantage that every execution cycle is attributed either to an instruction or some other reason, and it allows the permanent state to be modified only in the MEM stage. For example, if Inst1 were a memory load and required a context

1. The WB stage is not present in the simulator code, the MEM stage commits results itself.

switch at time 1, the THB exception should only be accounted for when the instruction is restarted.

Some other exceptions (e.g. SPILL1, §6.3.1) do not branch to a handler; instead, they save the program counter, insert the exception pseudo-instruction, and then restart the offending instruction.

A *stall* arises when a pipeline stage cannot complete its work within one clock cycle, but must re-attempt the operation. An example is when the data cache is needed for a memory load, but is otherwise in use (§7.3). There are at least three possible ways of dealing with pipeline stalls. One method stalls the whole pipeline, and prevents any advancement; alternatively, only the units before the offending one may be stalled, allowing those past it to advance. The third, as used in the Alpha 21164 [Dig97], does not allow any unit past the decode stage to stall, but instead flushes the pipeline and restarts the instruction.

The Jamaica simulator does not use many stalls, but when they do occur the advancement EX-MEM is allowed if the EX and MEM operations have finished. Long latency operations (for example, cache misses or window spills/fills) are instead dealt with using exceptions, so that the instruction is restarted (method 3). This is needed for multithreading, so that another thread can take over the pipeline.

The treatment of normal instructions is now considered in more detail.

6.3 Registers

The decode stage is responsible for mapping logical 5-bit register identifiers to physical registers. The high two bits indicate which window (Global, In, Out, Extra) is needed, and the appropriate frame number is taken from the context. Unlike for general overlapping windows, an addition is not required; the low three bits (register within window) are concatenated to form the physical register.

Lazy Out and Extra window allocations (§5.1.1) are performed here. If the instruction references an unallocated Out or Extra register, a frame is mapped in from the free list in the same cycle. This happens at the same time as reads of other registers, since it is known that reading an unallocated register will re-

turn zero. An Out and Extra window cannot be allocated together in the same cycle, however; if both are needed, a stall is generated.

The decode stage also detects the 'load delay' condition, where the result of a memory read is needed by an immediately following instruction; a stall cycle is requested as long as a memory load is in EX and its destination register is a source of the ID instruction. The stalled instruction will be allowed to leave ID only after the load has advanced to MEM (this is one reason for allowing the EX-MEM advancement during a stall).

6.3.1 Spilling and filling

With lazy Out allocation, almost any instruction (not just the subroutine calls JSR and BSR) may allocate a new frame, and hence might cause a window spill. If the ID stage finds the free list empty when attempting an allocation, it generates a `SPILL1` exception and saves the program counter. Spilling can also be requested explicitly: the `EVICT` instruction acts exactly like `SPILL1`, but always spills the bottom frame of the current context, and it also sets a register to indicate whether it succeeded. This is used by the runtime system to empty the window stack out to memory, so that a new thread can be scheduled in that context.

In the decode stage, the `SPILL1` exception selects a window to evict using a pseudo-LRU (least recently used) algorithm[1]; it also requests that the fetch stage insert a `SPILL2` operation before refetching the original instruction. The EX stage calculates the address to which to spill, modifies the count and limit fields of the evicted context, and may also raise a `SPILL_OVERFLOW` exception if the spill area is exhausted. The MEM stage writes out the first four registers to the data cache; the following `SPILL2` writes the second four. The freed frame is allocated to the requesting context rather than being replaced on the free list, and will then be mapped in when the instruction is re-executed. A single cycle stall is necessary after `SPILL2`, so that the window state has stabilised before it is used.

Filling can only be caused by return instructions, and (with lazy Out allocation) may also need a spill. If a spill is required, it proceeds as above; fills them-

1. The evicted window must be at the bottom of a context's resident stack, and not the In window.

selves work similarly, by inserting `FILL1` and `FILL2` operations and then restarting the instruction.

Four registers are read or written in a single cycle using a special 128-bit wide path to the data cache. However, the registers are always aligned to a group of 4, and the address is always a 4-word boundary (half a cache line, here), so this will be achievable without the full complexity of a 4-ported register file. The 128-bit path is not used in normal pipeline operation.

6.4 Control transfers

Control transfers are divided into three categories: unconditional, conditional (comparing a register value with zero), and subroutine calls. The first two comprise ordinary branches and jumps, and these are simpler than the last.

6.4.1 Branches and jumps

Branch and jump instructions are both recognised in the decode stage. For jumps the destination is a register value, whereas branches add an immediate value to the program counter.

For conditional branches, simple static prediction (like the Alpha 21164 [Dig97]) is performed: backwards branches are predicted to be taken, forwards ones untaken. Unconditional branches and jumps work as if predicted taken (and are always correct, of course).

Since the destination is known at the end of ID, a correctly-predicted taken branch (or jump) will cause a one-cycle pipeline bubble; a correctly untaken branch has no penalty. The prediction is not resolved until the comparison in the EX stage, so a misprediction of either kind costs two cycles.

6.4.2 Subroutine calls

The `JSR` and `BSR` operations are both unconditional, and cause a window shift (Outs become Ins) in the EX stage by modifying the context's state; the ID instruction will be annulled whilst this happens because of the branch. Additionally, they write the PC+4 value to `%o7` (which becomes `%i7` after the shift);

this implicit dependence on Outs is checked for lazy allocation in the normal way. Apart from that, they proceed like other unconditional transfers.

Returns (`RET`) act like a jump to the address in `%i7`, but also cause a window shift in the other direction (Ins become Outs). If the thread tries to return from its final window then it terminates (§5.2), by raising a `THREAD_DONE` exception. If necessary, the token is created and released; otherwise, the context is marked idle.

6.5 Multithreading

The fetch stage handles most aspects of multithreading. A context switch request can originate from the memory stage, on a data cache miss, or from the fetch stage itself. The MEM stage's request is similar to an exception, in that later instructions are cancelled and the IF stage inserts a `CTX_SWITCH` pseudo-op (which is treated as a no-op). On an instruction cache miss, the fetch unit performs a context switch only when the previous attempts' `ICACHE_MISS`es have reached the ID and EX stages. In both cases, the pipeline is effectively flushed of the old context before the new one starts executing. A switch therefore costs three cycles.

Apart from this, multithreading does not add much complexity to the pipeline. Once logical registers have been resolved to physical ones at decode time, most instructions do not need to know to which context they belong. The one major difficulty is that a `SPILL_OVERFLOW` exception must be handled in the context of the evicted window's owner, not the thread which caused the eviction. Thus, taking this exception also implies a context switch.

Threads are selected for execution using a least-recently-used algorithm. When a thread is switched in the quantum timer starts counting, and the fetch stage will request a context switch once the timer expires (the default is 1000 cycles). Switches do not occur whilst exceptions are being processed, so a spill or fill must complete before another context starts executing. If spills are frequent and cause cache misses this could be a performance problem; however, the observed program behaviour does not justify the extra complexity of switching during exceptions.

6.6 Thread distribution

The simulated processor directly implements the model described in §5.2, with the additional functions (the token distribution ring and memory bus interface) handled by the TIU. This maintains the token pool and passes tokens around the ring. Since the ring is made of point-to-point connections between neighbouring processors, it is clocked at the core clock speed (not the, possibly slower, memory bus speed).

The token request (`TRQ`) instruction executes in a single cycle; the TIU returns a flag indicating whether the operation succeeded.

6.6.1 Sending a thread

The `THB` and `THJ` instructions calculate the destination PC in the normal way, by an addition or register read, and like subroutine calls have an implicit dependence on the Out registers. These are read four at a time, using the same mechanism as for window spilling, and placed in a line buffer; the TIU stalls the main pipeline whilst performing these reads. If the line buffer is already occupied then the fork instruction stalls until it is free.

Once the data are in the buffer the TIU requests the memory bus, and then broadcasts the data along with the token value and PC in a single bus transaction (§7.2).

The simulator models the TIU as having its own interface to the memory bus, but in practice much of the logic would be shared with the data cache, resulting in the structure of Figure 16.

6.6.2 Receiving a thread

When a thread is broadcast on the bus, the TIU compares its thread ID with those of each idle context. The first match will accept the work; in this case, the TIU stalls the pipeline and writes the register values into the context's In registers, sets the PC, and marks the context as runnable. It also asserts the bus signal indicating that the thread was accepted.

6.6.3 Interprocessor interrupts

The interrupt mechanism is similar to the thread distribution, but does not send any register state. The matching is done in the same way, but a match sets a bit in the context's irq_flags. If the context was idle, it is marked runnable.

When the interrupted context is scheduled, assuming that interrupts are not masked, the fetch stage detects the non-zero irq_flags and inserts an `IRQ_PENDING` exception into the pipeline instead of fetching a normal instruction. When this reaches the execute stage, the interrupt is taken properly by generating a `TAKE_IRQ` exception, which saves the program counter, sets the IRQ mask, and branches to the handler. If the PC was zero then it calls the idle IRQ handler, otherwise the running handler.

This two-stage procedure allows instructions currently executing (which may take multiple cycles, for example when spilling registers) to complete, so that interrupts are taken cleanly on an instruction boundary.

6.7 PALcode

Sometimes, code running on the simulator needs access to 'internal' processor features. For example, the runtime system directly manipulates the context's state (spill area, threadId, etc) when it performs a software context switch. To support this capability without implementing a very specific instruction, whose details may have to change in future models of processor, the simulator follows the Alpha in using 'privileged architecture library' code (PALcode) [Dig92]. The Alpha has a special instruction 'call_pal,' but the Jamaica simulator instead traps subroutine call addresses near the top of memory (above 0xffff0000). The linker knows about the addresses and names of the functions, so the assembly code '`BSR timestamp`', for example, will work correctly. The complete list of PALcode routines is in Appendix B.2.

The aim in the Alpha is to keep implementation-specific operating system code localised, so that future processor models with different internal registers, for example, only need changes to small portions of code. The Jamaica system bends this slightly, and uses PALcode for three purposes. Firstly, as with the Alpha, internal registers may need to be read or modified, and PAL-

code accomplishes this. However, the Jamaica simulator does not include the system-specific instructions which the PALcode routine would itself use – instead, the trapped 'subroutine' is executed inside the simulator in a single cycle. Secondly, it emulates floating-point arithmetic; a real implementation would include a floating-point unit[1]. Thirdly, it performs character input/output operations, since hardware devices are not modelled.

The PALcode routines are called infrequently in the benchmarks considered here, and the inaccuracy in the simulations is negligible.

6.8 Conclusions

This chapter has outlined the Jamaica simulator's modelling of the processor. It uses a simple five-stage pipeline, and handles exceptions, branches and stalls correctly, allowing every execution cycle to be accounted for. The simulator's 'reality' breaks down with PALcode, where some infrequently-used routines and I/O features are not modelled realistically

1. The benchmarks considered here only perform floating-point arithmetic incidentally, and not during the main body of the program.

CHAPTER

7. The memory system

The performance of the memory system will be crucial for any future processor, but particularly for the Jamaica architecture combining multithreading and multiprocessing. Multithreading allows latency tolerance by the system as a whole, but places greater demands on the memory and bus bandwidth [BGK96]; a large number of outstanding transactions must also be supported. Effective multiprocessing needs processor-processor transfers to be handled efficiently. The Jamaica memory system is therefore simulated in detail, and incorporates several novel features.

The Jamaica CMP has private first-level I and D caches, connected over a shared bus to a single second-level cache. The interface to external memory is combined with the L2 cache controller (Figure 18).

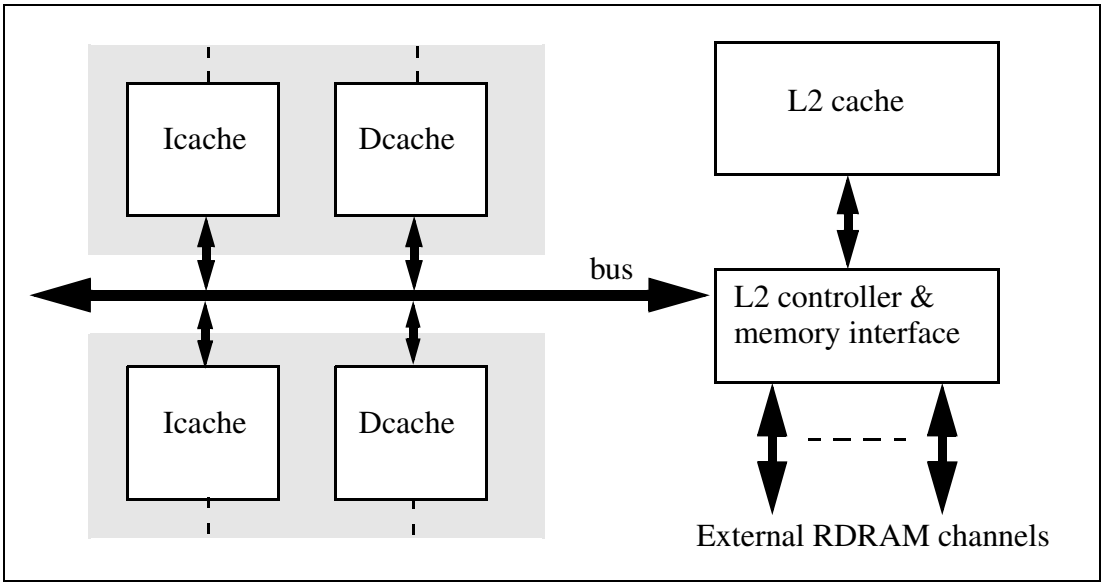


Figure 18: Memory system outline

If external devices require coherent access to the same memory (for example, direct memory access by peripheral controllers, or a board-level multiprocessor built from CMPs) then a separate memory controller would be needed, with another shared bus. This results in a hierarchy of shared buses (Figure 19). The coherence protocol described here can be extended to work

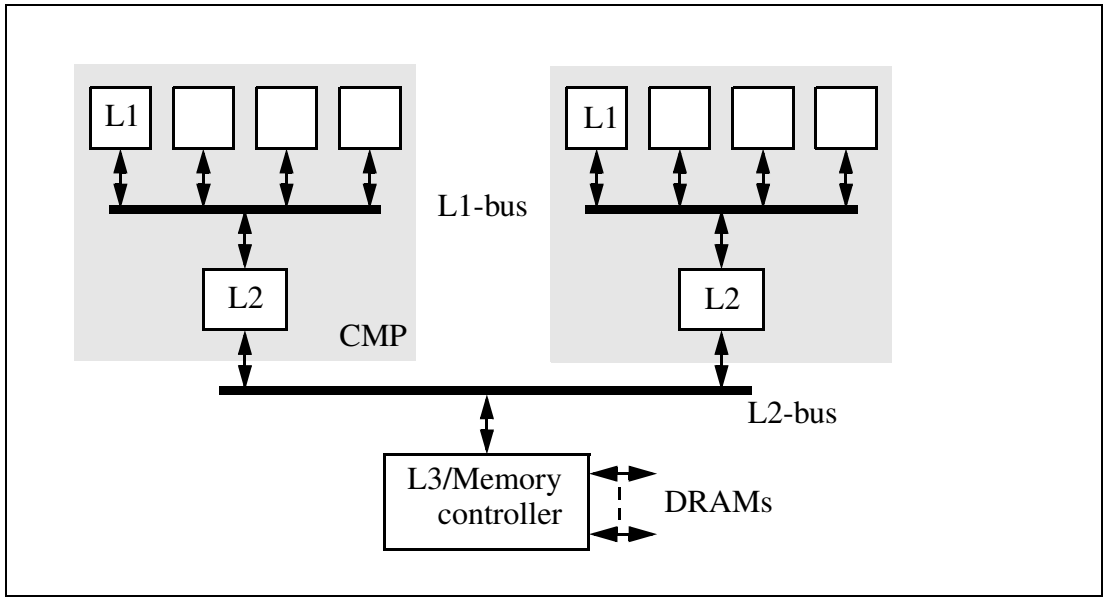


Figure 19: A hierarchy of shared buses

with a hierarchy of buses, although the simulator does not support this; the changes needed to the L2 cache are described in §7.4.2.

7.1 Cache coherence

The simulated processor maintains sequential consistency among the caches. This is for simplicity of programming, initially, and a future implementation could relax this and add memory barrier instructions to the architecture.

7.1.1 Cache states

The protocol used is based on the MOESI states, but is more aggressive than the Dragon or Illinois protocols (§4.2.1). L1 cache-to-cache transactions are allowed without updating memory, and ownership can also be transferred. In general, the main memory and L2 cache are relied on as little as possible, because L1-L2 inclusion is not maintained.

Any cache line can be in one of four states, encoded by two bits: the Exclusive bit, and the Owned bit. The Exclusive bit, if set, indicates that this cache holds the only visible copy of a line, and it is therefore writable. A set Owned bit indicates that this cache is responsible for writing the line back to memory. The four states are named:

- Shared (S) – not exclusive, not owned.
- Owned (O) – not exclusive, owned.
- Exclusive (E) - exclusive, not owned.
- Modified (M) - exclusive, owned.

An entry in the cache can also be invalid (I), indicating that no line is present. This gives rise to the MOESI name. Bus transactions can change one cache state to any other (§7.2.4); a write by the processor will change the E state to M.

In a cache hierarchy like Figure 19, the L2 cache's Exclusive bit covers the L1 caches serviced by that L2 also, effectively considering the L2 and its L1s as a single unit. Thus a line which is exclusive in the L2 cache can be in a shared state in several of its L1s (and therefore not writable by any of them), or exclusive in one of its L1s, or not present in any of them. It cannot be present in any other L2 or L1 cache.

At most one cache in the entire system can have ownership, however. This causes a complication when an L1 cache acquires a line exclusively which is owned by the L2. An L1 line in the E state will spontaneously become owned (M) if the line is written; the L2's retaining ownership would result in the L1 and the L2 cache *both* having ownership. The L2 will therefore write out-of-date data back to memory if it evicts the line. In the simulated system this would be inefficient but would not affect correctness because no active agents ever see this data: the valid L1 will always intervene. In a hierarchy like Figure 19, the wrong data may become visible. There are three solutions:

1. The L2 can give ownership to the L1 cache when the line becomes exclusive. This may cause an unnecessary writeback on the L1-bus if the line is not then modified. The 'at most one' property of ownership is maintained, and an owned line is always up to date.
2. The L2 can query the L1s whenever it evicts an owned line, to check that the data are correct. An L1 may then agree to retain ownership (in which

case the writeback can be aborted), or lose ownership and exclusivity; this transaction is unnecessary if the L1 had modified the line and keeps it. This solution changes the meaning of ownership to treat the L1s and L2 more like a single unit.

3. The L2's writeback can continue, if a subsequent read of this line by any other processor in the system will check the L1s first. This means that the L2's writeback transaction should not be observed by any other agent on the L2-bus. More traffic is also created on the L2-bus, for the unnecessary writebacks and because when writebacks do occur they cannot satisfy other L2s' outstanding requests.

Here, the first solution is taken, for simplicity and to reduce the traffic on lower buses and to memory. Baer & Wang [BW88] also considered the problem of ownership in the L2, but they assumed that the L1's transition to M would be detectable on the L1-bus.

7.2 The shared bus

The bus connects all the first level caches with the L2 controller and memory interface, and is also used for transferring threads. The hardware protocol is pipelined and supports split transactions. However, because the system is integrated on a single chip, the design parameters are quite different from conventional multiprocessor split-transaction buses, for example those of the SGI Challenge or Sun Enterprise [CS98].

All the agents on the bus work together in lock-step; transactions happen with fixed timing. The conventional protocols split the address (request) and data (response) phases of transactions, because the data may be supplied after a variable delay even in the case of cache-to-cache transfers; the address and data buses are separately arbitrated, and requests and responses are related by being tagged with a small integer. In a CMP, however, the tightly controlled timing means that, for cache-to-cache transfers, the data phase can follow immediately after the address phase: as in the Firefly protocol (§4.2.1), several caches can supply the data at once. If the data must come from memory then the data lines will be idle, and a later transaction will supply the response; for a constant stream of memory reads, this effectively halves the peak bus band-

width, but (using the parameters in §1.3) this is still greater than or equal to the peak memory bandwidth. Arbitrating separately would slow the cache-to-cache transfers by an extra two cycles (25%) and complicate the implementation[1].

The protocol introduced here does not require caches to track all outstanding transactions; matching is done only using the cache line's address. Therefore the caches themselves do not limit the number of outstanding transactions in the system as a whole.

7.2.1 Signals

The bus itself has the following signals:

- A 128-bit wide Data bus.
- A 32-bit wide address bus (Addr).
- An identifier bus (ID), wide enough to match the original requester of a transaction with the reply.
- A 4-bit Type signal, indicating what kind of transaction is on the bus (§7.2.4).
- Four wired-or lines: Found, Excl, Own and Shared. The Found line generally confirms the success of a transaction, Excl and Own indicate the state of a cache line, and Shared indicates that another cache will also take the data from this request (a piggyback), or the line is already in another cache[2].
- A single line MAccept, driven by the memory interface to indicate that it could accept the request.
- For each user of the bus, two pairs of request-grant arbitration signals: master (MRq and MGnt), and slave (SRq and SGnt).

1. Another possibility may be to add another set of address and state lines, driven by the L2/memory in cycle 6 along with the data if no slave L1 will use the data lines for the current transaction. The data cycles belonging to a memory request could then provide the response to a previous transaction, and full utilisation of the data lines for a constant stream of cache misses.
2. Two lines are needed to determine exclusivity, Excl and Shared, because any slave cache can indicate either state: in effect, each cache can reply 'exclusive', 'shared', or 'no information', which cannot be conveyed with a single signal.

7.2.2 Timing

A single bus transaction takes eight bus cycles (Figure 20); the bus cycle

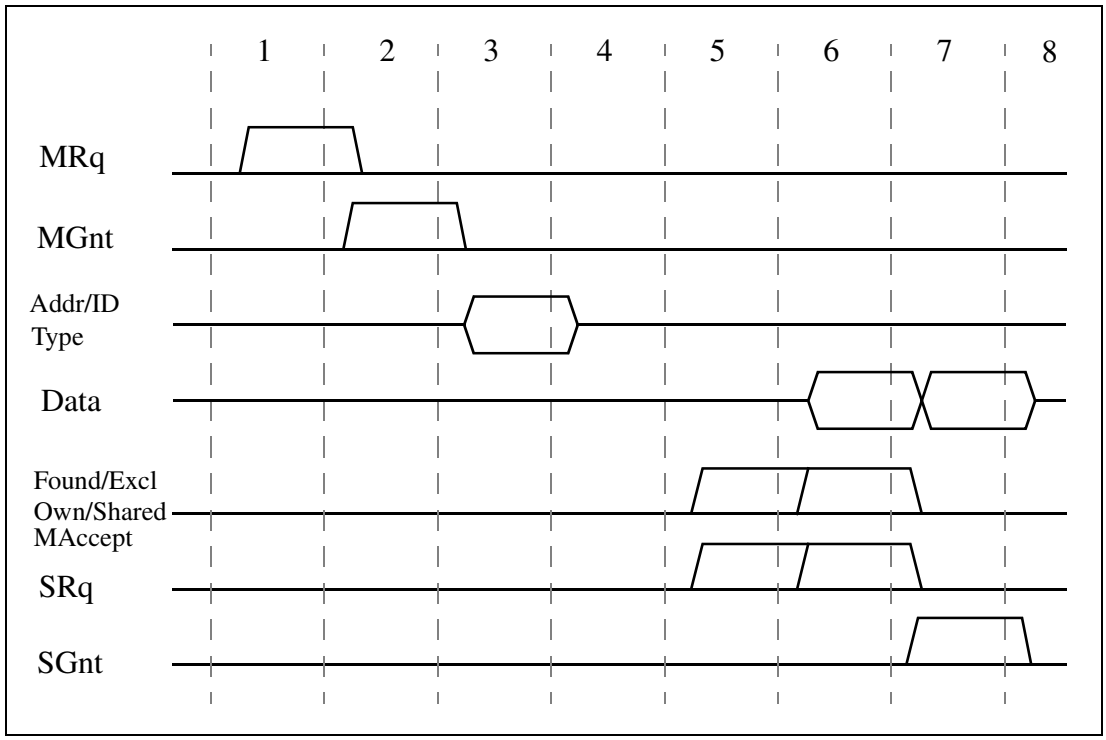


Figure 20: Bus signal timing

time is some multiple of the processor core cycle time. Each signal is given a whole cycle to stabilise on the bus and, apart from the Rq/Gnt pairs[1], the value to be transmitted is determined in the cycle beforehand.

1. *MRq*: Any requester of the bus asserts its MRq line.
2. *MGnt*: The bus arbitrator selects which will be master for this transaction, and asserts its MGnt line. (All other entities on the bus are referred to as slaves).
3. *Addr*: The bus master transmits the type, address and ID for the transaction.
4. *Tag check*: All the slaves check their cache tags and read the line if present, or decide on their response as necessary.
5. *Found*: If the slaves respond to the transaction, they assert the Found/Excl/Own/Shared lines now. They may also assert SRq, to arbitrate for ownership.

1. The Rq/Gnt pairs are point-to-point links to and from the arbitrators; all the other signals (apart from MAccept) are tristate or wired-or connections.

6. *Data1*: The first half of a cache line is transferred, either from an L1 or the L2 cache, and the L2 can also assert the Found/Excl/Own/Shared signals to indicate its state.
7. *Data2*: The second half of a cache line is transferred, and the slave arbitrator asserts the SRq line to one of its requesters.
8. *Write*: All bus users adjust their cache line state, store the data, or otherwise complete the transaction.

Two cycles are required for the Found signal, because the L2 must know whether an L1 can supply the data before providing the line itself.

7.2.3 Pipelining

The above timing is pipelined, with a new operation starting every two cycles. Thus the MRq phase overlaps with the Addr phase of the previous transaction, and so on. Only a single cache port is needed for data, because writes happen in cycle 8, and if the tag check in cycle 4 indicates a hit then the data can be read in cycle 5. However, tag checks and tag updates happen in the same bus cycle, and so may require a read and a write port. If the bus's cycle time is greater than the processor's, one port may be enough.

A cache line's state is in flux between the query in cycle 4 and the write in cycle 8; to simplify coherence, two consecutive transactions are not allowed to refer to the same line. This ensures that, by the time a tag check happens in cycle 4, previous transactions for that line have been completed. Caches must also ensure that coherence is maintained during a transaction, by not allowing a line to be modified once the data are supplied.

The pipelining means that each user of the bus must latch the state of the signals every other cycle (apart from the Data lines, which need latching every cycle), so they can be associated with the right operation.

7.2.4 Transactions

Bus transactions are divided into three categories: cache-originated (i.e. requests), memory-originated (replies), and thread operations (context sends and interrupts).

Caches produce four kinds of request: read-shared (SH), read-exclusive (EX), upgrade (UP), and writeback (WB). The first two are similar: the cache asks for a copy of the data, but EX also requires it to be returned in an exclusive state, so all other caches must invalidate the line. An upgrade also invalidates the line in other caches, but does not require the data to be sent (presumably it is already held but in a non-exclusive state[1]). A writeback is required when a cache evicts a line which it owns, and requests that another cache or the memory take ownership.

Memory replies are similar to the corresponding cache requests: memory-shared (MS) returns a cache line in a shared state, memory-exclusive (ME) in an exclusive state, and upgrade-acknowledge (UA) confirms that any off-chip invalidation has been performed if necessary. There is no need for the memory to reply to a writeback.

The non-exclusive transactions are designed so that listening slave caches will also accept the data, and many pending requests can be satisfied at the same time. However, the bus master is ordinarily guaranteed that its request will complete, which ensures against livelock. Also, the resulting cache state is as accurate as possible: that is, even a read-shared operation can fetch a cache line in an exclusive state, if it is not contained in any other cache. This may save an invalidation if the line is subsequently written.

For each slave, a cache line may be resident in the cache or in a writeback buffer, and may also be the object of an outstanding miss.

Details of the transactions are in Appendix B.3; they are summarised in Table 4 and Table 5. Notes preceded with an asterisk (*) are optimisations, discussed in §7.2.5.

7.2.5 Options and optimisations

Several of the protocol features are optional, in the sense that correctness is maintained if they are disabled. These include:

1. A more advanced processor implementation may determine that a cache line will be completely overwritten, for example on a register window spill or if a write buffer supporting merging were added. In that case the data are not required, and an Upgrade could be used even if the line is not in the cache.

Table 4: Bus master transitions

Operation	State	Transaction	Transition
Read	I	SH	if Found: accept data if (Excl & $\overline{\text{Shared}}$): exclusive (*1) if Own: owned
Write	I	EX	if Found: accept data if Excl: exclusive if Own: owned
	O/S	UP	if Found: change state to... if Excl: exclusive (else wait for UA) if Own: owned
Writeback	M/O	WB Own Excl if [M]	if (Found $\vee$ MAccept): completed
L2/Mem response	-	MS Own if [M/O] (*3) Excl if [M/E]	if Found: not owned
	-	ME, UA Own if [M/O] Excl	

- (*1) A read-shared transaction returns a line in an exclusive state from the L2 to a single requester. An M line in the L2 must also transfer ownership (see §7.1.1).
- (*2) A read-shared or memory-shared transaction allows other requesters to piggyback on the request.
- (*3) A shared response from the L2 cache (of a line in the O state) transfers ownership to an L1 cache. Note that if an M line is to be transferred exclusively then ownership must be given up also, so this interacts with (*1).
- (*4) A writeback gives ownership to another L1 cache, and if so, whether an L2 hit will take ownership in preference (which depends on the SRq/SGnt arbitration).
- (*5) A writeback satisfies another cache's pending miss. Note that this interacts with (*4), as it may or may not also transfer ownership.

Some of these are definite improvements: (*2) and (*5) have no downside, in the sense that they can only save transactions which would have been required

Table 5: Bus slave transitions

State	Transaction observed	Response	Transition
Pending miss	SH	Shared (*2)	if Found: accept data, shared

	WB	Found, SRq, Shared (*4, *5)	accept data
			if (Excl & $\overline{\text{Shared}}$): exclusive
			if (Own & SGnt): owned

	MS	Found, SRq if (≠ID): Shared (*2)	
	ME	if (=ID): Found, SRq	
Pending writeback	SH, EX	Found, Own Excl if [M]	completed
In cache	SH	Found, Shared	not exclusive
	EX, UP, UA	Found Excl if [M/E] Own if [M/O]	invalidate
	WB	Found, SRq, Shared	if SGnt: owned (*4)
Locked (§7.3.4)	SH, WB	Shared	
	EX, UP		set lock_flag
In L2	SH	Found Excl if [M/E] Own if [M/O] (*3)	if Own: not owned
	EX, UP		if Own: not owned if $\overline{\text{Excl}}$: respond with UA
	WB	Found, SRq	if SGnt: owned (*4)
Not in L2	SH	MAccept	if $\overline{\text{Found}}$: respond with MS
	EX, UP		if ($\overline{\text{Found}}$ & $\overline{\text{Excl}}$): respond with ME if (Found & $\overline{\text{Excl}}$): respond with UA
	WB		if $\overline{\text{Found}}$: write through to memory (*4)

anyway. The only question is whether they justify the additional complexity of supporting them.

For M lines in the L2, (*1) and (*3) are trading the cost of an unnecessary writeback if the line is not written against an upgrade if it is. For E lines, (*1) is

cost-free, and indeed E and M lines could be treated differently by the L2 in deciding whether to assert the Excl signal.

(*3) and (*4) are about whether ownership should remain at the L1 level for as long as possible, or whether it should be given back to the L2. Retaining it among the L1s may cost extra writebacks on the L1-bus, but may also allow better use to be made of the L2's limited associativity, since evictions will not require costly writes to memory.

(*4) for L2 misses is another trade-off, of a memory write each time an owned line is evicted from an L1 cache, against possibly several writeback transactions between the L1s plus a single memory write when it is evicted from the final L1. The former is better for write-once data; for items read and written by several processors, the latter is better.

7.3 First-level caches

The L1 instruction and data caches are similar, except that the Icache does not need to allow writes, or the 128-bit accesses for window spilling and filling. From now on, only the Dcache will be considered.

The Dcache is illustrated in Figure 21, which shows the main data paths. The cache is divided into two sides, dealing with the processor and the bus. Coordination between them is handled by the table of outstanding requests. It is assumed that reads and writes each take a single cycle[1].

The bus protocol requires access to the cache tags and data for snooping, but even though it is fully pipelined a single data port is adequate (§7.2.3). In fact, a single port is shared between the processor and bus sides, with the bus taking priority in the case of contention. Simulations have shown the benefit of a dual-ported array to be small.

1. This may require internal pipelining in the data array [HP96], but is not considered further.

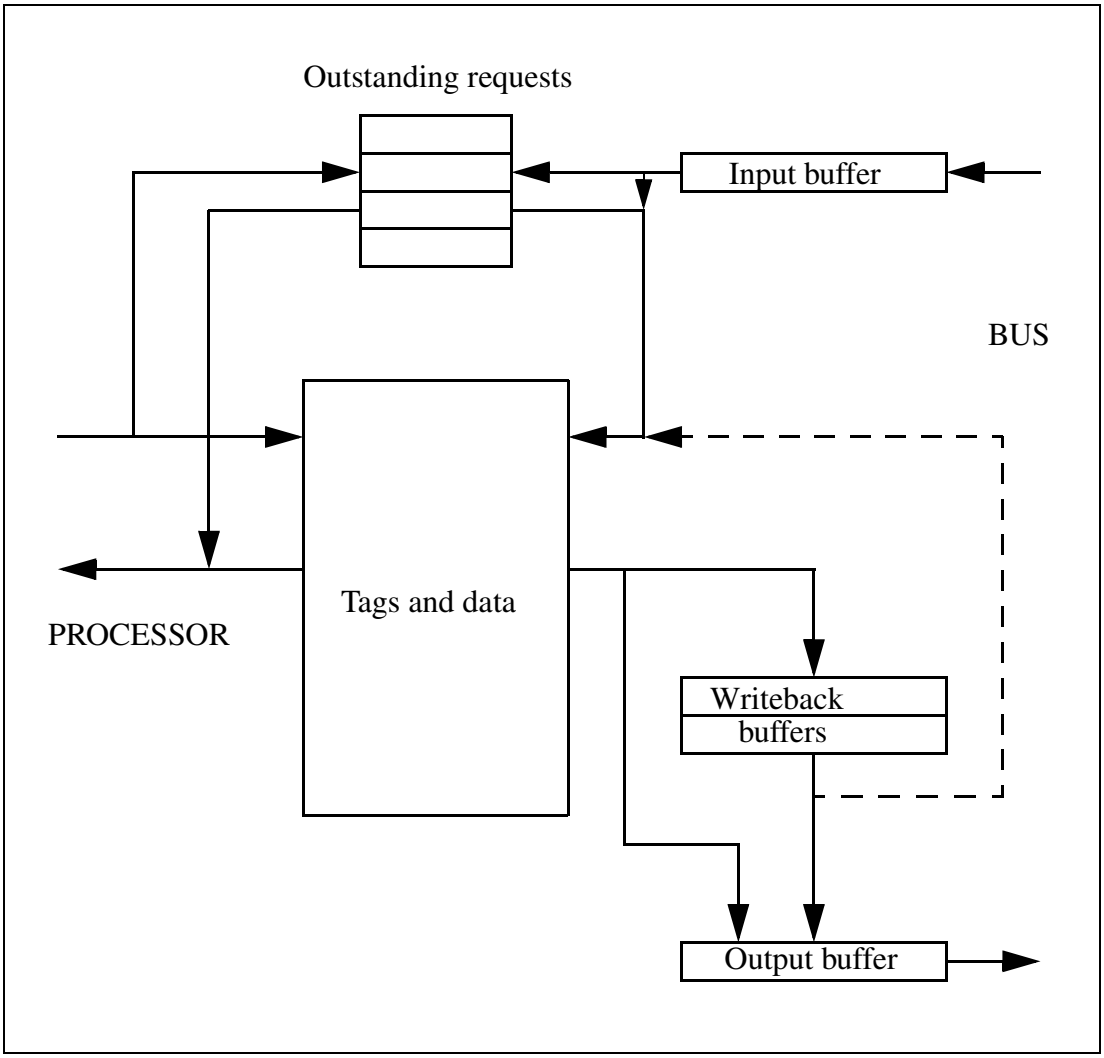


Figure 21: Data cache structure

7.3.1 The outstanding request table

Each context in the processor can have a single outstanding request, and has a dedicated slot in the table. Each entry holds the following information:

- `type`: none, read, write or wait.
- `addr`: the memory (word) address.
- `mask`: a 4-bit mask, to allow byte operations.
- `data`: a 32-bit value.
- `miss_set`: for an associative cache, the set into which this line will be fetched.
- `locked`: whether the current operation is a locked one (`LDL_L` or `STL_C`).
- `state`: recording the progress of the request.

There are also two more fields for the `LDL_L / STL_C / WAIT` operations; they are associated with the context rather than a particular outstanding request:

- `lock_base` and `lock_flag`: indicating a locked cache line address and its flag (§7.3.4).

The state information (in the simulation) is distributed between the `type` and `state` fields, but the combination obeys the transition graph in Figure 22. Unlike Kroft's lockup-free cache [Kro81], there is one slot per context and not one per outstanding line.

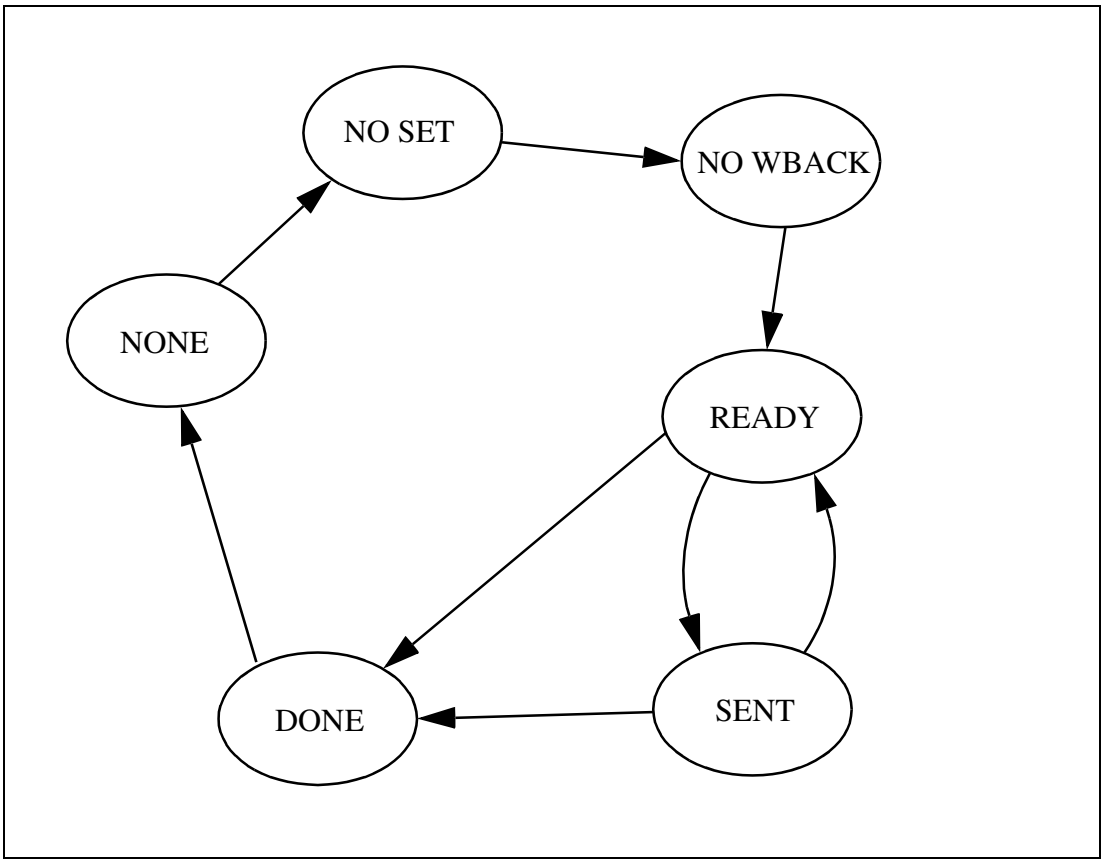


Figure 22: Outstanding request states

7.3.2 Example: a read request

As an example, consider a read request. When the processor attempts to read a word (e.g. an `LDL` instruction in the MEM stage) it passes the word address and context number to the data cache. The context number is the index into the request table; assume that the slot is empty (NONE). If there is a cache hit the data are returned immediately, otherwise the entry is filled in to indicate a (non-locked) read with state NO_SET. At this point, the thread will be

marked as waiting for memory, and a context switch will be initiated by the MEM stage.

A set is now identified to hold the incoming line; if another context has an outstanding miss for this line then that is used, otherwise a pseudo-random or LRU policy selects one. The request state moves to NO_WBACK.

If the selected set needs writing back to memory (i.e., the line is owned) then it must be transferred to a writeback buffer. If there is a buffer available, the line is evicted and the state moves on to READY.

At this point, the request can be sent to memory as there is a vacant line awaiting the reply. The memory bus is requested, and when granted the state moves on to SENT.

Eventually the response comes back from the memory; the data are written into the cache line, and the requested word is also held in the data field of the request buffer, advancing the state to DONE. The thread is woken up again, and the `LDL` will be retried. This time, the request matches the one in the buffer, and the value is returned, with the buffer returning to NONE. The buffering of the value is in case the cache line is invalidated by the time the load is retried; it guarantees that every request will eventually complete. It does mean that the time at which a load appears to take place is between the first attempt and the successful completion of the operation, possibly whilst another context is running on the processor. This does not affect the coherence properties.

Stores happen in a similar manner, but the value is written into the table (along with a mask, to allow byte-wide writes) when the request is made. The write is then merged with the incoming line when it is written into the cache.

7.3.3 The processor interface

In more detail, the processor's interaction with the cache proceeds as follows. The request from the processor carries the word address and the kind of operation required; a response is returned the same cycle. The processor will either retry the operation until it succeeds, or switch to a different context and retry it later. The response from the cache will be one of the codes in Table 6.

Assume that the request is a read:

Table 6: Cache responses to the processor

Code	Meaning
DONE	Request successful
MISS	Line not present in cache – a fill operation is needed
CONFLICT	Cache/tag array occupied by bus interface, so the cache could not be checked
COND_FAIL	On a `STL_C` operation, the lock flag indicates that the write should not complete
TABLE_CONFLICT	This context's outstanding transaction could not be aborted because ownership is being negotiated. This means that the request cannot be placed in the request table, and must be retried before the cache can accept it.
BUS_ERROR	The requested address was invalid (outside physical memory)

1. If the cache is being queried by the bus interface this cycle, return CONFLICT.
2. If the table entry is a read of the same address, and the entry is DONE, then change the entry to NONE, and reply DONE along with the data. This happens when a previous request is being retried and completes successfully. BUS_ERROR might also be returned here, although in practice a response would come from the memory itself via another transaction.
3. If the table entry is a read of the same address, and the request is still outstanding, then reply MISS.
4. Perform the tag check; if the line is in the cache, return DONE and the data.
5. Otherwise, a line fill is needed. If the table entry holds an outstanding request (i.e. READY or SENT), a bus transaction is in progress for that line, and this cache might acquire ownership as a result, then the table entry cannot be modified. Return TABLE_CONFLICT. Any other outstanding request can be aborted (i.e. its table entry overwritten) without affecting coherence.
6. Initialise the table entry for a read of the address, with state NO_SET.
7. Choose a set: if any other table entry is for the same line, with state not NONE, then use that entry's miss_set. Otherwise, select one some other way (pseudo-random or LRU).

8. If that set is the target of another entry's line fill or eviction, return MISS. Otherwise, record it in miss_set, and put the request's state to NO_WBACK.
9. If the line required is already in a writeback buffer, and its writeback transaction is in progress on the bus, return MISS. If it's not in progress, then swap the evicted line (if needed) with the contents of the writeback buffer, and return DONE.
10. If the target set's state requires a write back to memory (i.e. O or M), copy the line into a writeback buffer. If no buffer is available, return MISS.
11. Set the target set's tag, and put its state to I. At this point, the transaction is ready to go on the bus; advance the request's state to READY, and return MISS.

All of these checks can be carried out in parallel: there are comparisons with each entry already in the table, and with each writeback buffer. For an implementation, the only problem is the swap in step 9, which requires a read and write in the same cycle. If this is impossible in practice, the cache's current contents (the evicted line) would be written into the writeback buffer, with the write into the cache happening in a subsequent cycle. The requested data are still available to be returned, but a conflict may arise whilst the write takes place which is not accounted for by the simulator. Based on simulation runs, these swaps are rare.

Occasionally, a transaction may need to be aborted if the processor determines that it no longer needs the result; an example would be an instruction cache miss taken just after an incorrectly-predicted branch. In general, any of the entries in the table can simply be overwritten by a new request; the exception is when a transaction is on the bus and this cache may already be committed to taking ownership. In that case, the TABLE_CONFLICT result is returned, and the request must be retried until the cache can accept it. Aborting other transactions may result in a loss of exclusivity, but that does not affect correctness.

A write request follows the same pattern, except that in step 2 a failed `STL_C` will return COND_FAIL instead of DONE, and in step 4 the check must be for whether the line is writable (state E or M). Similarly, a swap from a writeback buffer may not satisfy the request if the line is not writable.

When any request in the table completes (i.e. moves to DONE), the corresponding context will be woken up by clearing the `memwait` flag (§5.2).

There is another option for the writeback buffers: they may be used as a small victim cache, by storing unowned lines when they are evicted. This only happens if there is no writeback pending for that buffer; the unowned line does not generate a transaction, and is simply overwritten when the buffer is next needed. However, in the intervening time, a cache miss may swap the buffer's contents back in to the main cache.

7.3.4 Synchronisation - LDL_L, STL_C and WAIT

These instructions are used to build atomic sequences in software, for synchronisation: e.g. test-and-set, fetch-and-add. They are distinguished in the request table by having the `locked` bit set.

An `LDL_L`, in addition to acting like a normal memory load, also sets the `lock_base` field to the address of the cache line referenced and clears the `lock_flag` when the value is nominally returned. On a cache hit, this is in the same cycle; on a miss, it is at the point where the returned value is buffered and the request's status goes to DONE. A cache line is locked as long as the `lock_flag` is clear.

Any subsequent write to the same line by any processor will set the `lock_flag`, and the write is detected by observing an EX or UP transaction on the bus. To ensure that these are generated, a line must not be exclusive as long as the line is locked by any other cache, so the Shared signal is asserted for any other transaction involving a locked line.

An `STL_C` fails if, at any time before the value is written into the cache, the `lock_flag` is set. The request then advances immediately to the DONE state.

A `WAIT` operation will cause the context to sleep until the `lock_flag` is set.

7.3.5 The bus interface

The cache's interface to the bus is responsible for initiating and responding to the transactions of §7.2.4. The cache asserts the MRq signal whenever it has an outstanding request in the READY state. When a transaction is initiated as the bus master, the request's state is advanced to SENT.

Whenever a slave determines that a transaction on the bus matches one of its requests and might satisfy it, then it also moves the state to SENT; this occurs in step 4 of read-shared and writeback transactions. Read-exclusive or upgrade transactions cannot satisfy pending requests, but if the current transaction will need a memory response then they can be advanced to SENT anyway until the memory response comes back: this saves a transaction which would have generated the same response.

A memory response (MS/ME/UA) matching a request in the SENT state will either satisfy the request, or else require it to be retried: the state moves to either DONE or READY as appropriate.

Pending requests must complete promptly when they are satisfied; the pipelined protocol means that a line fill completed in cycle 8 of one transaction could be invalidated in cycle 4 of the next-but-one transaction, i.e. the same physical cycle. Pending writes are therefore merged with the fetched line as it is written into the cache, and reads are buffered into the outstanding request table at the same time.

7.4 The second-level cache

The L2 cache is rather simpler than the L1 caches, since it has no associated processor or outstanding requests of its own. It is also allowed to be slower and larger, since the timing is much less strict and it can be clocked at the bus rather than core rate.

The cache checks its tags and data in exactly the same way as the L1 caches for each transaction. The only difference is that a read-exclusive or upgrade request from an L1 cache will not invalidate the line. The L2 cache can accept ownership from L1 caches via a WB transaction, and has the option of giving ownership away when an O line is fetched from it by a SH from an L1 cache. Currently, ownership is retained. Ownership must also be given away when exclusivity is granted to an L1 cache.

Exclusivity in the L2 cache also allows a line to be in any of the L1 caches belonging to it. Thus a read-shared can supply the data in an exclusive state, and a read-exclusive or upgrade request does not need to be propagated exter-

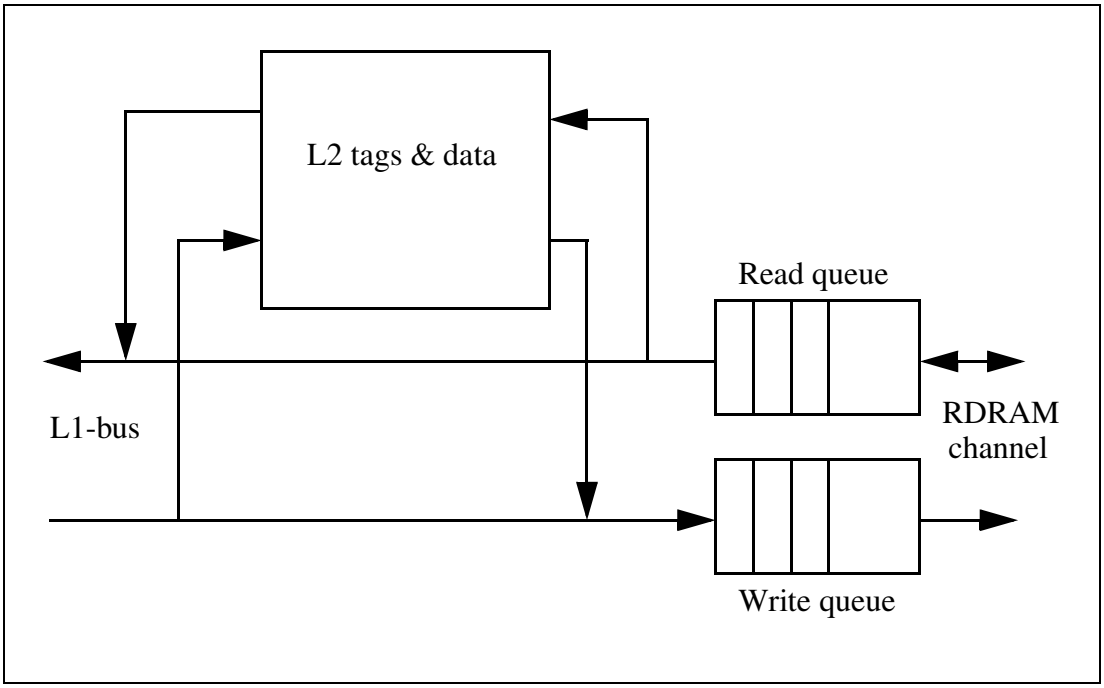


Figure 23: L2/memory interface

nally, if the L2 line is marked exclusive. In the simulated system, there are no other coherent agents at the L2 level, so every line in the L2 cache is treated as exclusive.

7.4.1 L2 cache misses

If a request misses in the L2 cache and no other L1 can satisfy it then it must be dealt with by the memory interface. A read miss ordinarily allocates a line in the L2 when the memory's data are returned; however, if this would cause an evicted line to be written back to memory and there is no free entry in the memory write queue (§7.5.3) to take it, then the new line is not retained. A write miss does not allocate a line.

The L2/memory interface has priority for the MRq/MGnt arbitration, so that a memory response will gain the L1-bus ahead of other requests.

7.4.2 Extension to a hierarchy of buses

In a hierarchy of buses like Figure 19, the L2 cache should insulate its L1-bus from snoop requests on the L2-bus as much as possible, as well as satisfying requests originating from its L1s. Maintaining inclusion would allow all L2-bus snooping queries to proceed without querying the L1 caches, with an L1-bus transaction only needed if the data must be supplied, i.e. if an L1 has mod-

ified data. An 'exclusive-invalid' state can be added to the L2 to indicate that one of its L1s has the correct data [CS98][BW88].

For the system presented, there are two problems here. Firstly, it must be possible to decide when snoop requests should be passed through to the L1-bus (the 'exclusive-invalid' state), as distinct from a normal exclusive state, when the L2 can supply the data itself. Conservatively, every hit to an E line could be passed through (but an owned or shared line must be the correct value). Alternatively, the normal protocol allows lines to be supplied exclusively to an L1 cache as the result of a read-shared request: abandoning this would require a read-exclusive or upgrade before the first write, and the L2 could then make the transition to the exclusive-invalid state. A transaction similar to a read-shared on the L1-bus would then allow the L2 to recapture the data (asserting the Shared signal), take ownership if necessary, and return to the normal E or M state.

Secondly, inclusion is not guaranteed between the L1 and L2 caches. Thus, in the absence of any further information, an L2-bus snoop which misses in the L2 would have to be propagated to the L1-bus. Whether this is a problem depends on the proportion of L1-bus bandwidth which is occupied; if the L2-bus has a lower frequency than the on-chip L1-bus, it may not matter. One solution is to reintroduce inclusion, at the cost of potential inefficiency in L1 cache use; another is to duplicate the entire collection of L1 tags at the L2 [Die99] (cf. CRAC, §4.2.1). One other possibility would be extending the L2 cache with extra tag and state bits, effectively increasing its associativity but without any more data. This could then hold the state of lines which are known to be present in the L1s and where the L2 does not hold valid data (e.g. in the 'exclusive-invalid' state described above), or even, if an 'absent' state is added, function as an 'anti-cache' to record that certain lines are known *not* to be present in the L1s[1]. This idea is not explored further.

The interaction of exclusivity and ownership in a hierarchy has already been discussed in §7.1.

1. As distinct from the usual 'invalid' state, which conveys no information at all.

7.5 The memory interface

The memory interface handles cache line reads which miss in the L2, and also evictions from the L2 which need a write back to memory. The simulator models Direct Rambus DRAMs (*RDRAM*) and channels, but many of the details have been abstracted away. Refreshing of dynamic memory is not modelled.

7.5.1 RDRAM channels and devices

Each channel has a command (address) bus and a data bus, connected to one or several RDRAM devices. Operations are pipelined so that a read command is followed by the data a certain number of cycles later; in between, other commands can be issued.

The RDRAMs themselves are internally modelled on the 128-Mbit Rambus parts [Ram99]. Each device is internally divided into 32 banks; each bank has 512 rows of 64 *dualocts*, the Rambus term for the smallest addressable unit of 16 bytes (Figure 24). A Jamaica cache line therefore consists of two dualocts, but for the purposes of simulation one transfer is counted when in reality two successive commands would be addressed to the same row.

7.5.2 RDRAM timing

These timings are summarised in Table 7 in terms of RDRAM clock cycles. The clock rate is half the data rate from Table 1 as data transfers use both clock edges.

Successive read or successive write command packets can follow each other with no gap, i.e. one every T_{CMD} cycles. However, certain adjacent banks cannot be active at the same time, as they share sense amplifiers; these will be referred to as *colliding* banks. A bank must be precharged before it can be used, and this sets a lower bound of T_{RC} between commands addressed to the same bank or adjacent colliding banks (Figure 25a). It is also necessary to insert a gap of $T_{R\_W\_BUBBLE}$ between a read and a write operation, equal to the round-trip delay experienced by the data on the channel, so that the read and write data do not overlap. A gap of $T_{W\_R\_BUBBLE}$ is needed between a write and a read to the same device, due to internal pipelining in the RDRAM (Figure 25b).

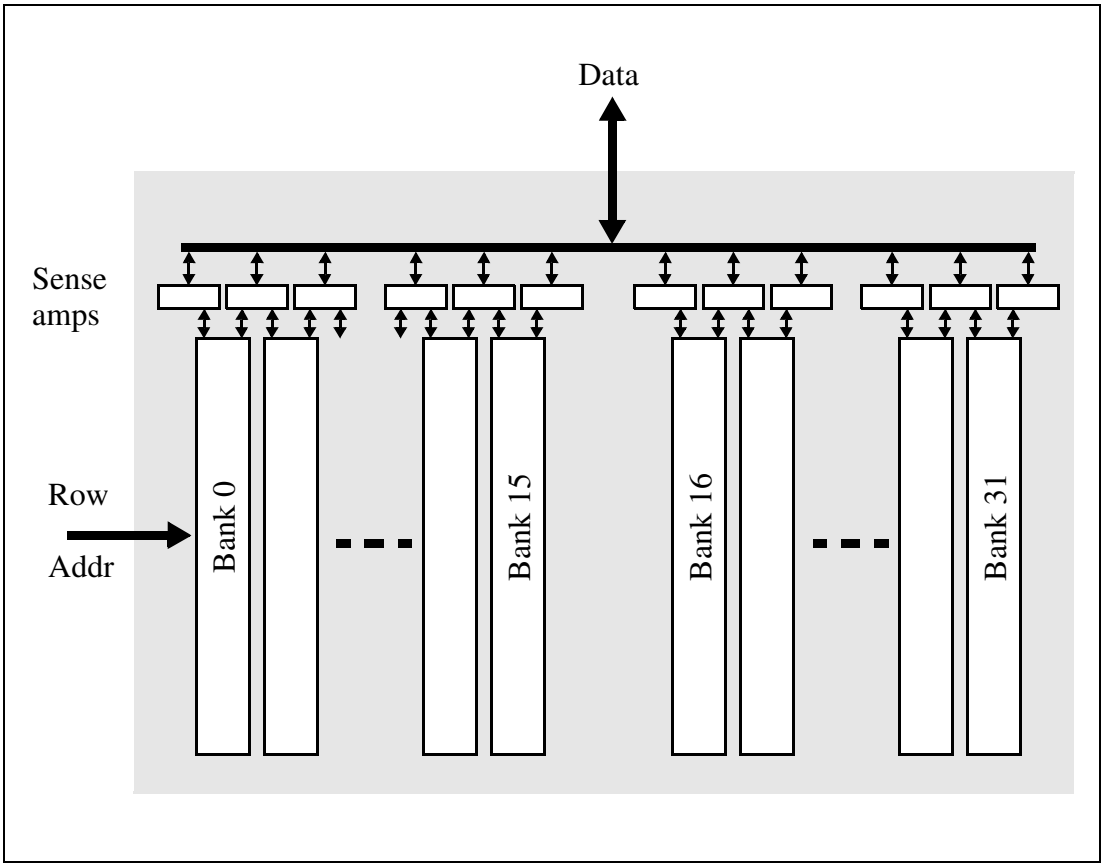
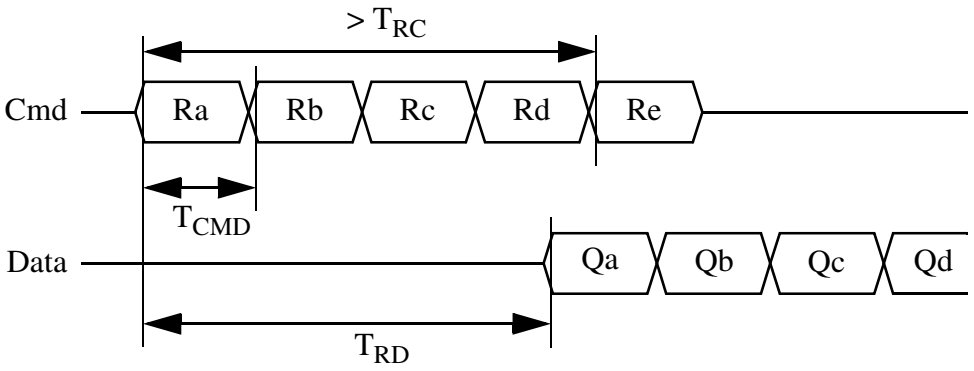


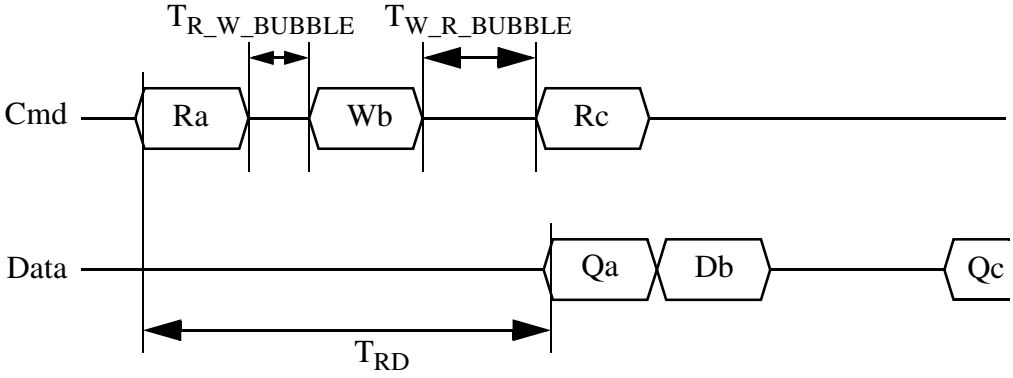
Figure 24: Internal RDRAM structure

Table 7: RDRAM timing parameters

Symbol	Meaning	RDRAM clock cycles	= CPU cycles	
			Current	Future
T_{CMD}	Transmit one command	8	20	64
T_{RC}	Interval between successive commands to the same or adjacent banks (row cycle time)	28	70	224
$T_{R\_W\_BUBBLE}$	Minimum gap between read and write commands	4	10	32
$T_{W\_R\_BUBBLE}$	Minimum gap between write and read commands to the same device	8	20	64
T_{RD}	Read latency (command to data returned)	28	70	224
T_{WR}	Write latency (for simulation purposes)	8	20	64



(a) Pipelined reads. $R_{a,b,c,d}$ must be to different banks. R_e may be to the same bank as R_a.



(b) Read-write-read: R_a, W_b, and R_c must be to different banks unless the T_{RC} criterion is met. The write-read bubble is needed only if W_b and R_c address the same device.

Figure 25: RDRAM timing

A read command will complete (i.e. return the data) T_{RD} cycles later; for simulation purposes a write completes as soon as the command is sent (T_{WR}), even though the data would not physically be sent until later.

7.5.3 Memory requests

The memory interface deals with cache line fill requests and writebacks. Each channel has a queue of outstanding requests (Figure 23) which are satisfied in first-come first-served order: reordering is not supported. This is similar to the Rambus reference memory controller [Ram00]. The logical queue is physically split into a read queue and a write queue, because the write queue must hold the data along with the cache line address; this partitioning places separate limits on the numbers of outstanding read and write operations per channel. When the memory interface accepts a request the simulator calculates

immediately when the operation will complete (based on the type of transaction and the immediate history of the particular channel, device and bank), and marks the queue entry with the completion time. If a buffer fills up, the MAccept line is not asserted and the L1 caches will retry transactions to that address.

The buffers allow merging of read and write requests if a memory transaction has not yet started, and a buffered write is allowed to satisfy a read.

Cache line addresses need mapping to channels, devices and banks, and there are conflicting requirements for avoiding collisions. A 'randomising' hash function is used, which should have similar behaviour on most programs, for example on arrays of differing strides. On the other hand, for any given stride a different mapping would result in fewer collisions.

A detailed investigation into mappings has not been performed, but the current simulator uses a function which is designed to avoid collisions when power-of-two strides are used. This is:

$$\begin{aligned}
\text{Channel} &= [a \oplus (a \gg c) \oplus (a \gg 2c) \oplus \ldots]_{3,\,2+c} \\
n &= d + b \\
a' &= [a \oplus (a \gg n) \oplus (a \gg (2n-1)) \oplus (a \gg (3n-2))] \gg (3+c) \\
\text{Bank} &= \text{Rev}(a'_{d,\,d+b-1}) \\
\text{Device} &= a'_{0,\,d-1}
\end{aligned}$$

Where a is the word address of the base of the cache line, b is the number of bank bits required (5, for these RDRAMS), c is the number of channel bits, d is the number of device bits (i.e. 2, if there are 4 devices on each channel), and Rev is a bit-reversal function.

The exclusive-or and shift operations are simply to bring some bits down from the high-order end of the address. The bank/device calculation maps across devices, then non-colliding banks and finally colliding banks.

Assuming that power-of-two strides are being referenced, this mapping will rotate successive addresses around all of the channels; banks and devices will also be rotated provided the stride is no more than 2^{3n+c+3} bytes, i.e. 256kB for the current parameters ($c = d = 0$), and 1MB for the future ones ($d = 0, c = 2$).

7.6 Conclusions

This chapter has presented the Jamaica memory system, which is simulated in detail. Each CPU on the chip has private write-back L1 data and instruction caches, connected by a pipelined, split-transaction bus. The bus protocol incorporates the best features from several existing protocols, in particular that ownership may be transferred between caches. In addition, a single transaction may satisfy many processors waiting for the same line.

Another novel feature is the `WAIT` instruction, which extends the usual load-locked/store-conditional mechanism to allow low-overhead spinlocks in a multithreaded system.

An on-chip L2 cache is also included; unlike in most systems, L1-L2 inclusion is not maintained. The memory interface models modern, high-performance pipelined DRAMs (based on the Rambus parts).

CHAPTER

8. System software

A set of software tools complements the hardware simulator presented in the last few chapters. The system as a whole must be evaluated by running benchmark programs and applications on the simulator, but realistic programs written in a high-level language (in this case, Java) require suitable compilers and run-time libraries. The effectiveness of the thread manipulation features is also determined by how well high-level programs can make use of them, so the runtime system (RTS) must provide a suitable abstraction.

Figure 26 shows how the components fit together. At the highest level, Java source code may be processed by the Javar source-to-source restructuring compiler [BG97]; this parallelises loops based on annotations provided by the programmer. The modified code, and any other Java source, is compiled to Java bytecode [LY96] using the Java compiler 'javac', from the standard Sun Java Development Kit.

The resulting Java class files are translated to assembler instructions by 'jtrans', which is intended to model a just-in-time (JIT) Java compiler although it actually produces static code. The Java runtime libraries (native methods, virtual machine functions) are written in C, compiled using a version of the Princeton LCC compiler modified by Ian Watson; the Jamaica back-end is based on that of the SPARC.

Finally, all the assembler code is assembled by 'as', and the object files are then linked by 'ld' to produce an executable binary for the simulator. Both of these tools have been written for the project, and follow the ELF

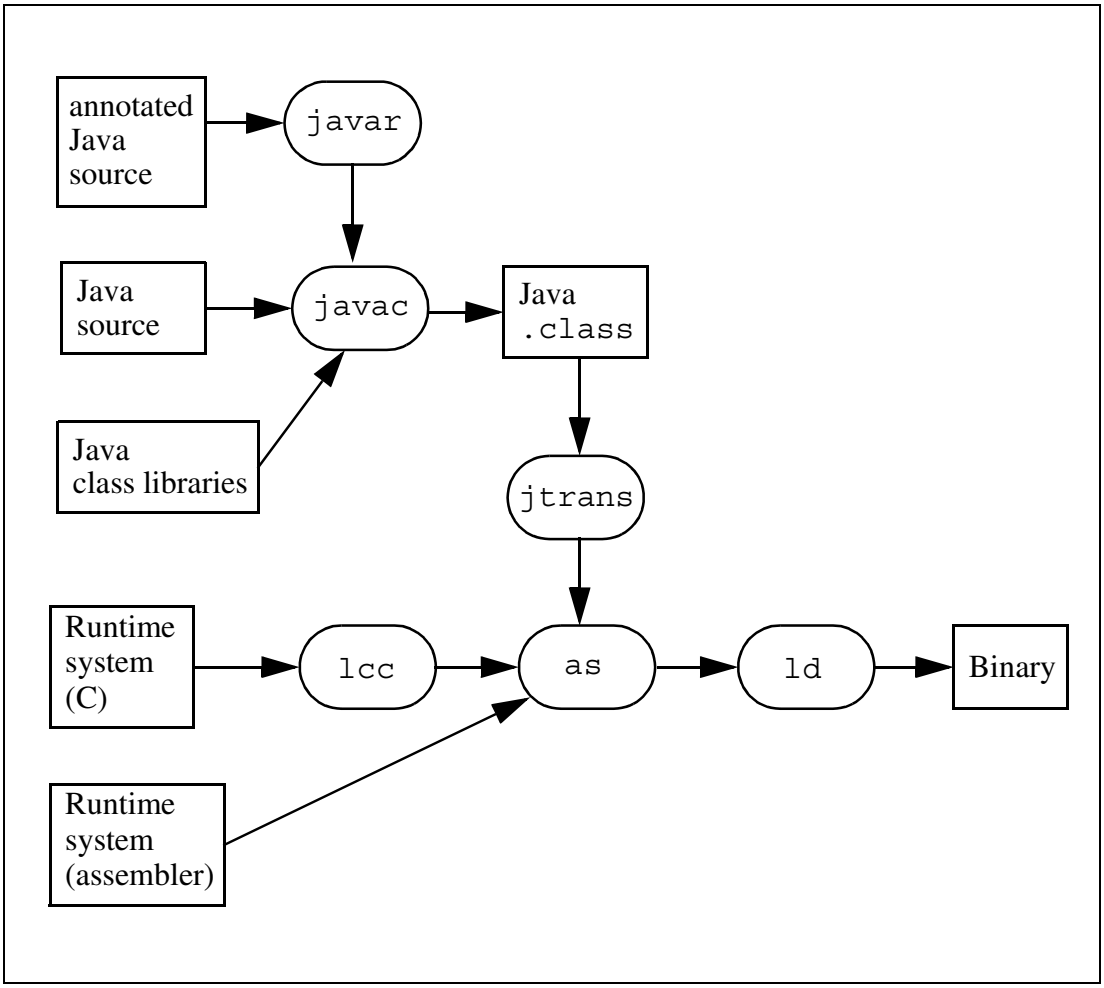


Figure 26: System software

specification[1] [TIS93] for the object and executable files. Dynamic linking is not currently supported.

8.1 Javar

The original Javar compiler [BG97] parallelises loops using guided self-scheduling. A collection of worker threads is created; the workers extract bundles of loop iterations from a task pool. For a simple loop, the pool contains only the lower and upper bounds of the loop range, along with the stride; if there are n worker threads, then $1/n$ of the remaining iterations are taken by adjusting the bounds.

Figure 27 illustrates a simple loop in a method, along with its annotation to indicate how many threads should be generated. Javar processes this to generate the code in Figure 28. The method now creates its 32 specialised worker

1. With adaptations for the particular relocation types necessary for Jamaica code.

```
class Simple {
  void method() {
    /*par threads=32 */
    for (int i=0; i<1000; i++) {
      /* body */
    }
  }
}
```

Figure 27: Simple class

threads (`parlooppack.LoopWorker` is a subclass of `Thread`), and passes this array along with the bounds to the `parloop()` function: this initialises the work pool, and starts the threads running. The `run()` method of each worker extracts from the pool, and calls back to the original class with the bounds and stride.

Ian Watson has modified Javar to parallelise loops using recursive subdivision of the bounds, so that the lightweight thread mechanism can be used; this is described in §8.6.

8.2 Jtrans

Ahmed El-Mahdy has developed the jtrans compiler. It takes Java class files and produces Jamaica assembler code. Many of the Java bytecodes correspond directly to Jamaica instructions - for example, the integer addition and load-constant bytecodes can be translated to `add` and `lda` respectively. Some others are more complex, and calls to the runtime system are planted (e.g. for creating a new object, or entering a '`synchronized`' method). The major work of jtrans is in performing register allocation, transforming the stack-based bytecodes into the three-register form. The register allocation algorithm is based on that of the Cacao JIT compiler [Kra98].

Currently, little optimisation is performed as jtrans is designed for speed; it does not perform method inlining, for example. The use of this and more advanced run-time optimisations (e.g. as performed by Sun's Hotspot system) may reduce the performance advantage of the register windows.

```
// This file has been generated by JAVAR

class Simple {
  void method() {
    // PARALLEL LOOP
    {
      LoopWorker_Simple_method_0[] worker_0 = (new
LoopWorker_Simple_method_0[32]);
      for (int i_0 = 0; i_0 < 32; i_0++)
        worker_0[i_0] =
          (new LoopWorker_Simple_method_0(this));
      parlooppack.LoopWorker.parloop(0, 1000,
worker_0, 0, 2);
    }
  }

  final void run_Simple_method_0(int l_0, int h_0,
      int s_0, parlooppack.RandomSync[] sync_0) {
    for (int i = l_0; i < h_0; i += s_0)
      { }
  }
}

class LoopWorker_Simple_method_0
extends parlooppack.LoopWorker {
  Simple target;

  LoopWorker_Simple_method_0(Simple target) {
    this.target = target;
  }

  public void run() {
    while (pool.nextWork(this))
      target.run_Simple_method_0(low, high, stride,
sync);
  }
}
```

Figure 28: Javar-parallelised Simple

8.2.1 Java object layout

Jtrans uses a direct pointer representation of objects, rather than using indirection through a handle. This saves a memory load every time an object is referenced, but may complicate garbage collection[1]. The header of each object

1. The current runtime system does not perform garbage collection, but this is being worked on.

contains its size and a pointer to its virtual method table (vtbl), and also owner and count fields for monitors using the object (§8.4.4). The instance data follow immediately afterwards.

8.3 The runtime system

The RTS has two related responsibilities: it (assisted by jtrans) generates the abstraction of the Java virtual machine (VM), and it provides implementations of native methods. The VM functions includes system initialisation, object creation, memory management, synchronisation, threading and error handling; native methods allow underlying implementation-dependent features to be made visible at the Java language level, for example file and terminal input/output. Ian Watson and Ahmed El-Mahdy have written the standard memory/object handling and native methods.

Of most interest here, though, is the thread management. Java threads are relatively heavy entities, and can be suspended, acquire locks, and wait on an object until notified. To begin with, the implementation of standard Java threads is presented; then the extensions for lightweight dynamically-created threads are introduced.

8.3.1 Spinlocks

Many of the RTS functions rely on spinlocks internally. The `acquire_lock` code is illustrated in Figure 29; the address of the lock is the only argument, in `%i0`. It uses the instructions `LDL_L` and `STL_C` to ensure atomicity; the `WAIT` instruction allows the spinlock to consume little processor time whilst it is spinning, as the thread will sleep until the cache line containing the lock is next written. There are two other observations: firstly, the test (`LDL_L/BNE`) does not invalidate the lock in any other caches; the invalidation is only needed at the attempt to acquire (`STL_C`). Secondly, in the case of no

```
acquire_lock:
L1:
    LDL_L %i1, 0(%i0)
    BNE %i1, L2
    ADD %g0, 1, %i1
    STL_C %i1, 0(%i0)
    BEQ %i1, L3
    RET
L2:
    WAIT
L3:
    BR L1
```

Figure 29: acquire_lock()

contention, the conditional branches fall through (and are correctly predicted). This sequence can be compared directly to that for the Alpha (Figure 15), on which it is based.

8.4 Java threads

A thread in Java has a dual personality, as an active agent *in* the virtual machine, and as an object (instance of `java.lang.Thread`) manipulated *by* the VM. As an object, a `Thread` has fields which can be read and set (e.g. `name` and `priority`); although all of these fields are designated private, so that methods outside the `Thread` class cannot use them, many of the management functions are handled at the Java level without requiring native methods. Internally, an executing thread needs space to save its program counter and registers, a stack, and a window spill area. The internal RTS data structure is linked from the `Thread` object using the `PrivateInfo` field intended for this purpose; this field is never interpreted at the Java level. The only other instance fields used by the RTS are `priority` and `threadQ`.

8.4.1 Starting a thread

A `Thread` instance is created in the normal way (using the `new` operator), and behaves like any other object until its `start()` method is called. The native implementation of this method performs the following tasks:

1. Allocate the PrivateInfo structure, stack and window spill area.
2. Create a single entry in the window spill area, zero apart from register 6 which is set to the new stack pointer. The area's `spill_area_count` and `spill_area_flags` are both 1.
3. Set the program counter in the PrivateInfo structure to point to the function `javaThreadEntry`.
4. Link the thread onto the run queue, at the appropriate priority level (§8.4.3).

The thread is now available for scheduling. The single spill area entry is created so that the normal context-switching mechanism (§8.4.2) can be used when the thread is removed from the run queue; the context switch routine

calls the thread entry function by *returning* (`ret`) to it. Register 6 is set so that the stack pointer is correct when the function starts executing.

The `javaThreadEntry` routine is needed because when a thread dies (returns from its `run()` method) it must call `Object.notifyAll()` on itself, to awaken any other threads which are waiting for it to terminate (because the `Thread.join()` function internally uses `Object.wait()`). It also locates the `run()` method, although this could equally well be done when the thread is started.

8.4.2 Context switching

Although the Jamaica processor can contain many threads at once, the RTS must be able to cope with more threads than this hardware limit. There must therefore be a means of switching the virtual machine's threads in and out of the available hardware contexts. For example, a thread must switch itself out when putting itself on a wait queue (§8.4.4). Before it does so, it must find a successor thread to switch in. If there are no threads waiting on the run queue, then a special idle thread is used (§8.4.3).

The context switch routine `thread_ctxswitch_release()` takes four arguments: a PrivateInfo structure to save the thread's state into, the PrivateInfo structure from which to restore, the threadId of the new thread (i.e. its Java `Thread` pointer), and the address of a word into which zero is written once the switch is complete. The latter is used to release a lock protecting the queue onto which the outgoing thread placed itself (if any); this release must be done by the new thread, because as soon as the lock is released the old thread could in theory be rescheduled on another processor.

The context switch proceeds by writing all the global and extra registers into the PrivateInfo structure, along with the return PC (from `%i7`) and stack pointer (from `%i6`). The stack pointer must be preserved because it is the only register which is callee-saved on a function call. The entire window stack is emptied using the `evict` instruction in a loop, until no more can be spilled. Then the new spill area's values are loaded into the Out registers, and the internal (PALcode) routine `ctxswitch` is called. This atomically writes the spill area values and threadId into the context structure.

The old context is now completely replaced; if the supplied address was non-null then the zero value is written to release a lock somewhere. The new thread reads its global and extra registers from its PrivateInfo, restores its return PC to `%i7` and stack pointer to `%i6`, and then returns. Since the entire window stack was just evicted, this return will cause a window fill from the new thread's spill area.

When a thread is switched back in, it therefore appears as if the call to `thread_ctxswitch_release()` which switched it out has just returned.

8.4.3 Scheduling and Idle threads

The previous section described the context switching mechanism, but there is also the question of policy: how does a thread decide when to switch out, and which successor does it choose?

The Java virtual machine specification [LY96] does not require pre-emption or fairness in scheduling, only that higher-priority threads be run in preference to lower-priority ones. In the Jamaica RTS, a Java thread only switches itself out when it suspends on a queue (`Object.wait()`) and when it terminates.

The run queue is maintained as an array of lists of `Threads`, each list being linked through its `threadQ` field; there is one list per priority level. The `resched()` function scans the run queue from high to low priority, looking for a potential replacement. If it finds one, it acquires the run queue lock, and removes the head of that list (assuming that the list is still non-empty).

If the run queue is empty then an 'idle thread' is used instead. A number of these are created at initialisation time, and a list of them is kept on a separate idle queue; if none is available then a new idle thread is created. An idle thread does nothing but scan the run queue, `WAIT`ing on the run queue lock between each scan; if it finds a replacement then it links itself back onto the idle queue and switches in the user thread. These idle threads ensure that a real thread is not held up indefinitely awaiting a replacement.

8.4.4 Synchronized, wait and notify

The Java language provides two ways for threads to synchronise with each other. The first is through the use of *monitors*, sections of code which can only

be entered if the thread holds a lock on a particular object. The `synchronized` keyword can apply to a particular block of code or an entire method; either way, javac and jtrans eventually reduce it to the `builtin_monitorenter()` and `builtin_monitorexit()` functions in the RTS.

Entering a monitor on an object X is done by comparing the current threadId with the owner field in X's header. If they're equal, the thread already has the monitor, so just increase X's count field. If the owner was null, atomically set it to the threadId and then write 1 into count. Otherwise, another thread owns the monitor, and the entering thread waits[1] on the owner field before retrying the sequence.

A monitor is exited by decreasing the count field; if it reaches zero, null is also written into the owner field.

The `Object.wait()` and `notify()` methods allow threads to sleep and be woken explicitly, rather than as a side-effect of entering a particular code block. The implementation here is based on that of the Kaffe JIT compiler [W+]. A hash value is calculated from the address of the object, and indexes into a hash table of wait-queue lists. Each entry in a wait-queue list identifies an object and the first and last threads waiting on it; other threads are linked through their `threadQ` fields. `Object.wait()` searches for an entry for the object, creates one if none exists, and adds the current thread at the tail. `Object.notify()` moves the head of the list to the run queue; `notifyAll()` removes the entire list.

Monitors are thus implemented with spinlocks, although the overhead is reduced using the hardware `WAIT` instruction; no other thread is scheduled by software whilst the entering thread waits. The hardware multithreading may hide some of this cost, but monitors should generally be used for protecting small regions of code, like shared variable updates. The `Object.wait()` and `notify()` operations are more expensive, but allow another thread to be scheduled.

1. i.e. the Jamaica hardware `WAIT` instruction, not to be confused with the Java `Object.wait()`.

8.5 Lightweight threads

The RTS described so far works with the usual Java two-step thread creation procedure (i.e. create the `Thread` object, then call its `start()` method). The Jamaica hardware allows a more dynamic model for managing threads; this section describes how the architectural features are made accessible to Java programs.

Java threads are relatively heavy entities; they can be suspended, acquire locks and wait on objects, and they need a stack and private RTS data area. Their creation and management are therefore expensive, and do not fit well with the fast hardware mechanism presented in §5.2. One possibility would be to change the Java model, and introduce another kind of thread with limited capabilities; these threads may be invisible to the normal Java operations (`wait()`, `notify()`, `currentThread()`), for example by simply running to completion. However, this would introduce far-reaching changes into the Java specification and RTS.

Instead, the Jamaica RTS makes the dynamic threads look as much like standard Java threads as possible. If the higher-level features are not used then the hardware mechanism proceeds with no overhead; otherwise, the RTS takes over to give the extra flexibility.

8.5.1 Lightweight thread creation

The RTS, at initialisation time, pre-creates some threads (complete with `Thread` objects, stacks and internal data) and places them onto contexts in the processor. This is like the idle threads of §8.4.3, but instead of scanning the run queue they put their threadId equal to their `Thread` object pointer, set their 'release token' flag, and then terminate. The acquirer of the token then has a `Thread` object 'ready to go', and may start it using a `THB` or `THJ`.

This only changes the Java model in three ways. Firstly, getting the thread requires a special method, and not the usual `new`/constructor combination. Secondly, the threads are pre-created as instances of `java.lang.Thread` itself, so they must be given a `Runnable` target rather than getting their `run()` method through subclassing. Thirdly, once they have terminated they must

not be manipulated any more, since they may be reused by a different section of the program.

During their execution, these lightweight threads are indistinguishable from normal Java threads. However, if they never `wait()` on an object, they will never need the services of the RTS and their entire life cycle, from token allocation to termination and re-release of the token, proceeds without any RTS intervention.

8.5.2 The LightThread class

The instructions to allocate tokens and start threads remotely are encapsulated by two native methods, which are placed in the `parlooppack` package for use by the modified Javar (see §8.6).

To fit with Javar's generated code, and work around the three differences described above, `parlooppack` contains a `LightThread` class. An instance of this class represents a particular invocation of a dynamic thread, and is valid even when the thread itself has gone on to other work. It is specified in Figure 30; its companion Mysys[1] is in Figure 31.

```
class LightThread implements Runnable {
    boolean finished;

    public native void light_start();
    public native void light_join();
    private final void light_export() {
        this.run();
        finished = true;
    }
    public void run() { }
}
```

Figure 30: class LightThread

```
class Mysys {
     public static native boolean free_proc();
}
```

Figure 31: class Mysys

1. There is no reason why this method could not be integrated into LightThread; it exists separately for 'historical reasons'.

The `Mysys.free_proc()` method corresponds directly to the `TRQ` instruction; in the current RTS, several `TRQ`s are executed successively, and the method returns the logical-OR of the results. This increases the probability of picking up a token if one is available.

The `LightThread.light_start()` method equates to `Thread.start()`, except that a token must have been allocated beforehand. It causes the remote thread to start executing at the `light_export()` method, after setting up the stack pointer correctly. This extra wrapper is necessary to set the `finished` flag when `run()` returns, and is the equivalent of `javaThreadEntry` in §8.4.1. The `light_join()` method uses the hardware `WAIT` instruction in a loop until the `finished` flag is set. The only argument passed by `light_start()` to `light_export()` (and then on to `run()`) is the implicit `this` pointer.

The assembler implementations of these native functions are in Figure 32; `light_join()` is very similar to `acquire_lock()`. The method names have been mangled by jtrans to distinguish their class and argument/return types.

8.5.3 The medium RTS

An alternative runtime system (referred to as using 'medium' threads) has also been implemented. The major difference is in the `free_proc()` method; instead of using the token-passing mechanism, the idle threads are kept on a queue in shared memory, and `free_proc()` tests whether the queue is empty. If it finds an idle thread, then it acquires a lock on the queue and removes the thread. To minimise changes to the rest of the RTS, `free_proc()` then artificially creates a token, and everything proceeds as before.

This allows the token-passing mechanism to be compared against a software-only scheme, whilst still allowing the fast thread argument passing on the bus.

8.5.4 The run queue

In the 'heavy' Java-threads-only RTS each of the idle threads scans the run queue. The 'light' and 'medium' systems replace these with threads which im-

```
light_start___V_parlooppack_LightThread:
  stl %g0, 16(%i0)! finished = 0
  mov %i0, %o0! pass object on as arg.
  thb light_export0! call export
  ret

light_export0:
  ! entry point for running remotely
  ! set up stack
  ! (on entry we have caller's SP in %i6)
  rcr -1, %i6! currentThread
  ldl %i6, 28(%i6)! ... ->PrivateInfo
  ldl %i6, 96(%i6)! ... ->init_stack_ptr
  br light_export___V_parlooppack_LightThread

free_proc___Z_parlooppack_Mysys:
  trq %i0
  trq %i1
  trq %i2
  trq %i3
  trq %i4
  trq %i5
  bis %i0, %i1, %i0
  bis %i0, %i2, %i0
  bis %i0, %i3, %i0
  bis %i0, %i4, %i0
  bis %i0, %i5, %i0
  ret
```

Figure 32: Implementation of native methods

mediately terminate and release a token. This means that there are no longer any threads examining the run queue; a thread woken as a result of a `notify()` would never execute.

To remedy this, the first extra thread started in these two RTSs is not a token-releasing idle thread but a scheduling thread. If it finds a thread on the run queue then it removes it and repeatedly tries to acquire a token by calling `free_proc()` (either version). When it finds a token, it starts the remote thread executing at a 'sacrifice' routine. The remote thread links itself back onto the idle queue and then performs a software context switch to the newly runnable one.

In the absence of any threads on the run queue this scheduling thread consumes no processor time, but it does occupy a context which would otherwise be available.

8.6 Modified Javar

Ian Watson has modified the Javar compiler to use the lightweight threads. It parallelises the Simple class from Figure 27 to produce the code in Figure 33; notice, however, that the `parlooppack.LoopWorker` class is not the same as that for normal Javar.

There is no longer a central pool of work; instead, the worker thread takes its bounds directly as an argument, and the computation proceeds by recursive subdivision. The argument to the '`/*par threads=... */`' directive is taken as a control on the minimum grain size, i.e. the smallest number of iterations which may be subdivided further. Assuming that subdivision occurs, the `run_Simple_method_0()` function divides its range in two, and determines whether it can create a new thread to handle half of its work. If so, the new thread is created and its bounds passed to it by creating a new worker object; otherwise, the first recursive call is performed locally. The second half of the range is always done locally; when it completes, the worker thread's completion is awaited using `light_join()`.

Eventually the minimum size is reached, and the loop is actually run.

The specialised `LoopWorker`-derived class here serves two purposes. When the thread is created it identifies the method (`run_Simple_method_0()`) and holds the arguments to pass to it, so that effectively they are passed through memory rather than using the argument-transferring feature of the `THB/THJ` instruction. (The single pointer to the `LoopWorker_Simple_method_0` object is passed in hardware instead). Also, it identifies the particular unit of work given to the allocated thread, so that it can be joined on – the `Thread` pointer is not enough, since it can be reused once this work is done.

If both of these needs could be removed then the LoopWorker and consequent object allocation would be unnecessary: this is the single greatest overhead when creating new dynamic threads. The argument-passing and method-identifying problem is the harder: what is needed is a version of `light_start()` which can be called with a 'pointer to a method' and an arbitrary number of arguments, instead of creating the new `LoopWorker` object. This could in theory be done with the Java reflection capabilities

```
// This file has been generated by IW's modified JAVAR
class Simple {
  void method() {
    // PARALLEL LOOP
    {
      run_Simple_method_0(0, 1000, 1);
    }
  }

  final void run_Simple_method_0(int l_0, int h_0, int
s_0) {
    if ((h_0 - l_0) >= 32) {
      int mid = (l_0 + ((h_0 - l_0) / 2));
      LoopWorker_Simple_method_0 worker_0 = null;
      if (Mysys.free_proc())
        worker_0 =
         (new LoopWorker_Simple_method_0(this, mid,
h_0, s_0));
      else
        run_Simple_method_0(mid, h_0, s_0);
      run_Simple_method_0(l_0, mid, s_0);
      if (worker_0 != null)
        worker_0.light_join();
    }
    else
      for (int i = l_0; i < h_0; i += s_0)
        { }
  }
}

class LoopWorker_Simple_method_0
extends parlooppack.LoopWorker {
  Simple target;

  LoopWorker_Simple_method_0(Simple target, int low,
int high, int stride) {
    this.low = low;
    this.high = high;
    this.stride = stride;
    this.target = target;
    light_start();
  }

  public void run() {
    target.run_Simple_method_0(low, high, stride);
  }
}
```

Figure 33: Modified Javar on Simple

(`java.lang.reflect`), but this involves so much wrapping[1] of the arguments involved that it multiplies the number of objects created. A more direct solution is for Javar to create a specialised native method, complete with assembler code, for each call. The thread joining problem could be solved if each thread kept an invocation counter, which it increments whenever one of its tasks completes; joining then becomes testing for the counter's being greater than it was when the work was forked.

8.6.1 Load-based inlining and lazy task creation

The Mul-T parallel Scheme system [KHM89] implemented *load-based inlining* (LBI), similar to the technique used by the Jamaica RTS. The expression `(future` *X*`)` corresponds to a fork; it immediately returns a *future*, an empty value indicating a computation in progress, whilst the evaluation of *X* (any other expression) proceeds in parallel. When *X* terminates, its value replaces the future. Any (strict) operation needing the value of the future will block until it is available, an implicit join. LBI was an optimisation: if the load of the system (the length of the processor's local task queue) is greater than a certain parameter then don't create a future for *X*, but instead 'inline' it into the serial computation. The Jamaica RTS is similar, except that the load criterion is token availability rather than a run queue length, and since future values cannot be handled the forking function must deal with the joining itself.

Later work replaced LBI with *lazy task creation* [MKH91], based on work-stealing. Instead of testing the load, the default is to inline. However, enough information is maintained, in the stack and a lazy task queue, that an idle processor can 'un-inline' the call by taking the continuation whilst the original processor is still evaluating *X*. They identified four problems with LBI which may be solved by lazy creation, and they are considered here as the same criticisms may apply to the Jamaica RTS. They are programmer involvement, irrevocability, deadlock, and 'too many tasks'.

The first, programmer involvement, arises because LBI is not the best solution everywhere. They report that some programs run faster with eager creation, where every fork must take place. Also, when LBI is used, the run queue

1. The target method is represented as a `Method` instance, and all its arguments must be objects, i.e. `int` types are converted to instances of `Integer`, etc.

length parameter must be chosen. The former may apply to Jamaica, although even allowing eager creation would require significant changes to the RTS. The latter does not apply, since a token either is or is not available, and the decision is not just based on a local queue length.

The second, irrevocability, is a valid criticism, and is inherent in the programming model. The two-step forking procedure, i.e. `free_proc()` followed by `light_start()`, gives the parent thread a chance to divide its workload as it sees fit. However, with the Javar-produced code, the recursive call happens anyway, and the subdivision is always the same: in this case, it might be possible to implement lazy task creation with some clever stack manipulation if some kind of futures were also added.

Deadlock is only a problem with LBI as an optimisation of eager creation: in the Jamaica RTS, the default must be not to fork, and the corresponding problem is ensuring that parallelisation is safe. The Scheme language, being functional, does not suffer this.

Too many tasks, caused by the need to keep at least one task on the local run queue, is also not a problem in Jamaica, because the token mechanism guarantees that a context is always available. The problem instead would be too few tasks, but this is hidden to some extent by the multithreading.

8.7 Conclusions

This chapter has described the runtime system and compilers for the Jamaica system. There are two routes for producing parallelised code. The original Javar generates Java threads which manipulate a software pool of work. Alternatively, a modified Javar can use the lightweight mechanisms provided by the Jamaica processor, with either hardware token-passing or a software queue to locate idle contexts.

Whichever version of the RTS is used, a full range of Java thread operations is supported, including monitors, `Object.wait()` and `notify()`.

There are still some inefficiencies in the lightweight threading routines, in particular the need to create a new object for each thread invocation; a possible

solution involving specialised native methods was outlined, but has not been implemented.

Finally, the RTS's style of fork-join parallelism was compared with the futures of the Mul-T parallel Scheme implementation, and in particular the load-based inlining optimisation. Most of the criticisms of LBI, solved by more complex lazy task creation, either do not apply to the Jamaica RTS, or are difficult to avoid because they interact with the Java programming model.

III. Results

CHAPTER
9. Assembler benchmarks

To evaluate Jamaica's features it is necessary to run benchmark programs. Undoubtedly, the only way to judge performance on real applications is to run real applications. However, real programs have many complex interactions, and isolating and explaining the performance of individual machine features is difficult.

This chapter therefore presents several simple benchmark programs, designed to test particular aspects of the system. Each program is short, and hand-coded in assembler; the aim is to target a particular component so that it becomes the performance bottleneck, and in this way discover bounds on the machine's capabilities.

Since every other program depends on it to some extent, the memory system is considered first.

9.1 The memory system

The limiting component for many programs, particularly when dealing with large data sets or streaming data (e.g. video or audio), will be the memory system. Hiding latency and enabling memory pipelining were two of the major motivations for introducing multithreading. There are several questions to be answered:

1. The pipelined DRAMs have some fairly complex interactions between successive requests. What proportion of the theoretical peak bandwidth is realistically achievable?
2. Can multithreading hide the latency and make use of this bandwidth?
3. How many threads are required?
4. Where does the shared bus limit scalability?
5. How do bus protocol options improve performance?
   a. Allowing read-shared requests to return exclusive data.
   b. Allowing piggyback reads on shared data.
   c. The hardware WAIT instruction.

The simplest memory-intensive program is the well-known Stream benchmark [Str], which performs long-vector arithmetic on double-precision (64 bit) floating point values. Here, 32 bit integers are used, with three separate variants: a vector copy (**c** = **a**), an addition (**c** = **a** + **b**), and an addition where one source is also the destination (**c** += **a**).

In these experiments, the vectors are 1M words long (4MB each), and the L2 cache is reduced to 64kB; at this size it should have little effect on the results. The arrays are allocated consecutively in main memory, so the n^{th} element of each array will map to the same line in the L1 cache; this will increase cache interference.

The following numbers are tabulated:

- P, the number of processors on the chip.
- T, the number of threads per processor.
- SpUp, speedup relative to the 1 processor, 1 thread case.
- CBW, cache bandwidth, that is 8MB (copy) or 12MB (add) divided by the execution time.
- MBW, memory bandwidth achieved through the RDRAM channels; recall that the peak figures are 1526 MB/s (C) and 19,087 MB/s (F).
- Bus%, the utilisation of the on-chip bus.

The next five are a breakdown of the RDRAM channel usage (aggregated over all channels, if more than one). Due to rounding, they may not sum exactly to 100%.

- Act%, time active (i.e. memory bandwidth as a fraction of peak).

- None%, idle because the request queue was empty.
- RWB%, read-write bubble.
- WRB%, write-read bubble.
- Bank%, bank conflict.

A normal Stream result would measure and report the CBW figure. Here it should be treated cautiously; for large numbers of processors the combined L1 data cache size is too large compared to the vector length (32 processors with 16kB each is 512kB). The speedup figure is not very meaningful for the same reason.

None of these programs uses the token distribution mechanism to start the threads running; a (reliable) IRQ is used instead, so that all the threads begin at the same point.

9.1.1 Vector copy

The main loop of this program is in Figure 34. Register `%i0` is initialised to hold a count of the number of words copied by each thread, i.e. 2^{20} divided by the number of threads; `%i1` and `%i2` point into the source and destination vectors.

```
L1: ! copy loop
  ldl %i3, 0(%i1)
  add %i1, 4, %i1
  stl %i3, 0(%i2)
  add %i2, 4, %i2
  sub %i0, 1, %i0
  bne %i0, L1
```

Figure 34: Vector copy code

The program performs one load and one store per iteration, for total CPU-cache traffic of 8MB. However, each cache line (32 bytes) copied will generate one shared read, one exclusive read and then a writeback, so the memory traffic will be more like 12MB (less any data which are not written back from the L1 caches at the end), and the MBW is expected to be about 1.5 times the CBW. Since the L1 caches are only 4-way associative, interference may occur when more than two threads are resident in the same processor.

Table 8 and Table 9 in Appendix A.1 have the results for up to 32 total threads, with a maximum of 16 on any one processor. Memory and bus utilisation, and the cache bandwidth, are plotted in Figures 35 to 41 against the total number of threads, i.e. PT.

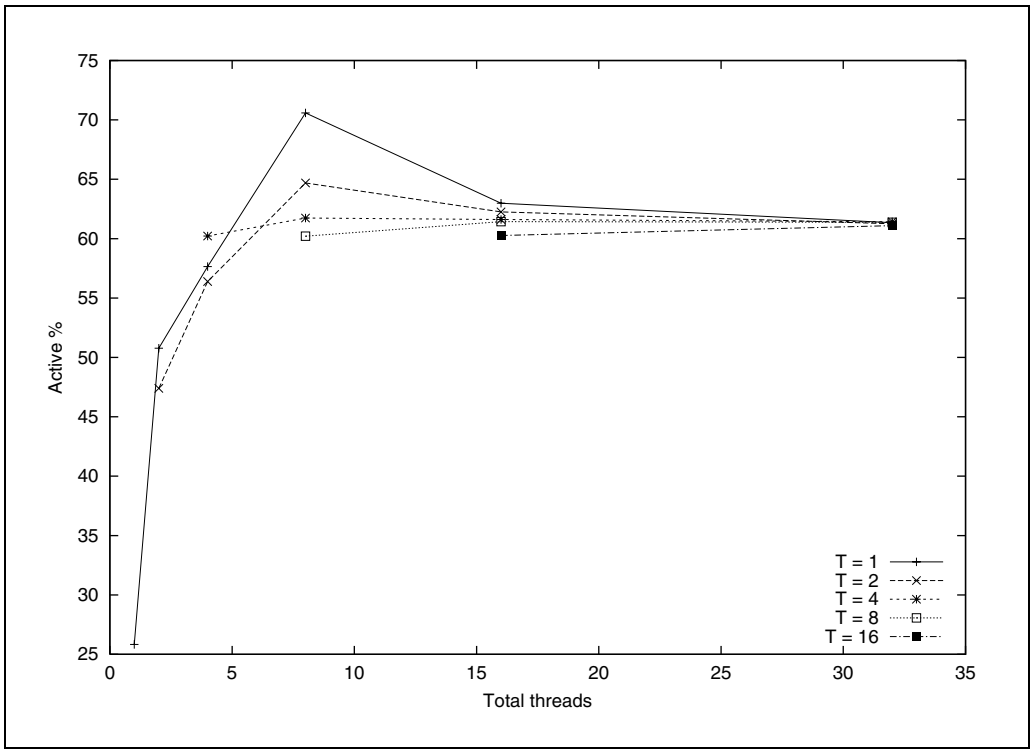


Figure 35: Vector copy, memory bandwidth (Act%), current

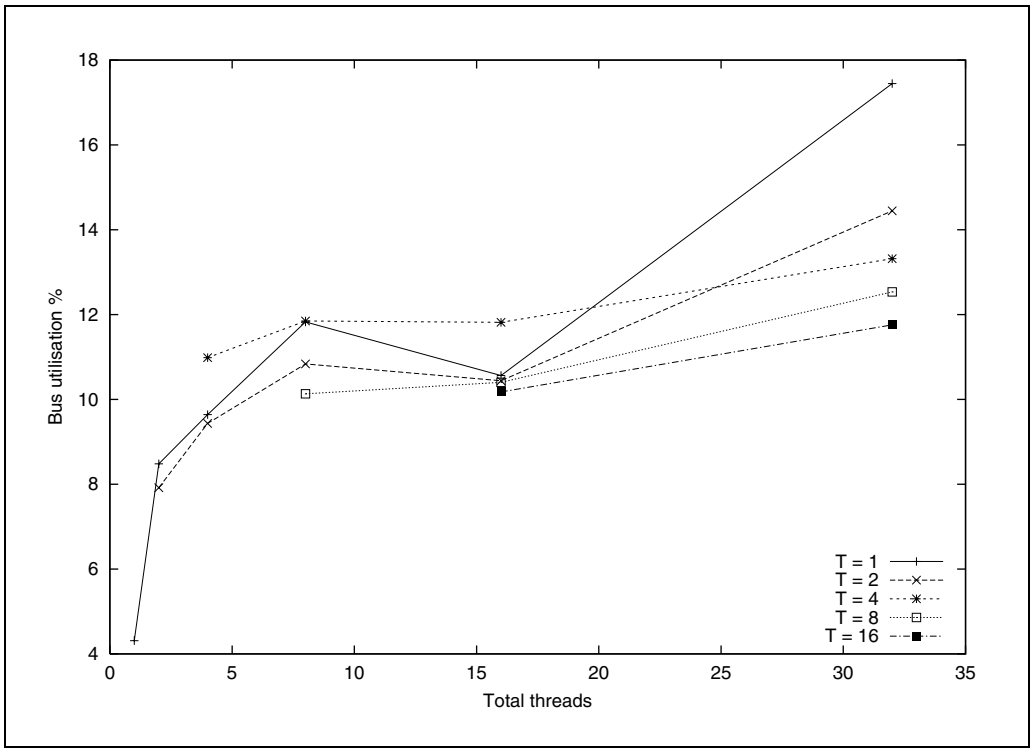


Figure 36: Vector copy, bus utilisation (Bus%), current

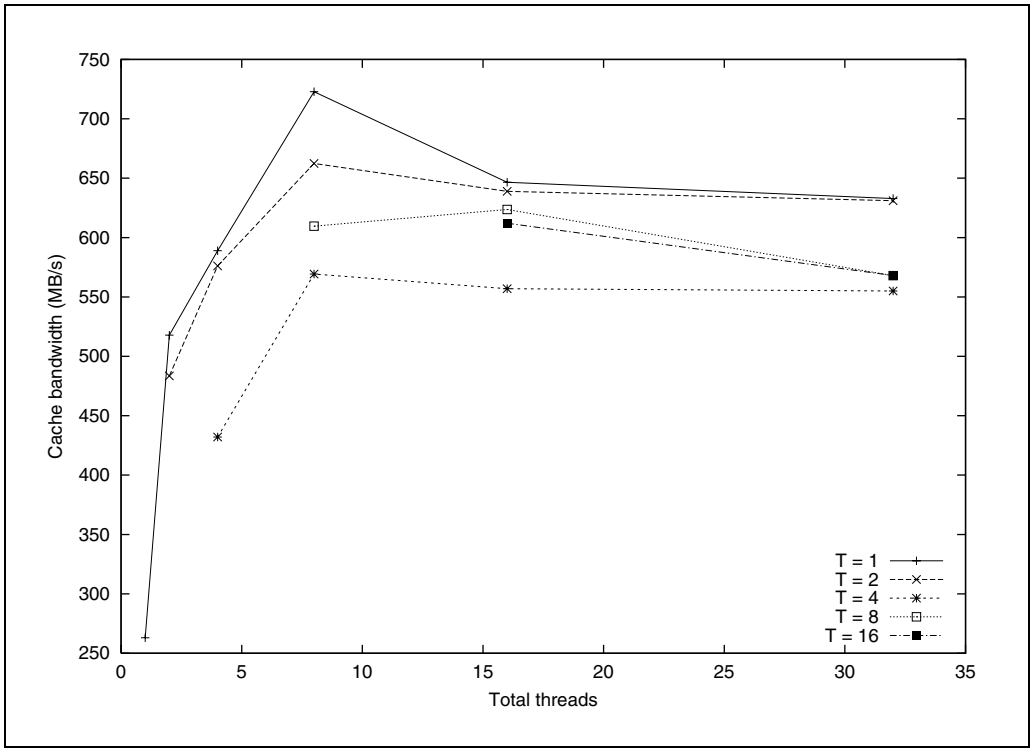


Figure 37: Vector copy, cache bandwidth (CBW), current

Taking the 'current' results first, a single thread sustains a memory bandwidth of 394 MB/s, or 25.8% of peak. Adding a second thread on the same processor increases this to 723 MB/s (47.4%); using a second processor instead gives 774 MB/s (50.8%). In each case, the MBW figure is 1.49 to 1.50 times the CBW.

Going up to four threads the MBW increases again, achieving 860 MB/s (56.4%, at P=2, T=2) to 918 MB/s (60.2%, at P=1, T=4). However, the CBW actually falls in the latter case, to 1/2.13 of the MBW (Figure 38). This shows the effect of interference in the L1 cache; the memory system may be sustaining good bandwidth, but the processor is not using it efficiently. The best CBW is attained with four processors and one thread each. Figure 38 shows two distinct behaviours: one where CBW is linearly related to MBW, and one where the memory is saturated, MBW is almost constant, and cache interference reduces CBW.

Eight processors with one thread each produces the best overall MBW: 1078 MB/s (70.6% of peak), which is 1.49 times the CBW. For all the results with at least eight threads in total the memory channel is fully saturated (the None column is close to zero). The achieved bandwidth then depends on the occurrence of bubbles and bank conflicts. Neglecting banks for a moment, if the channel

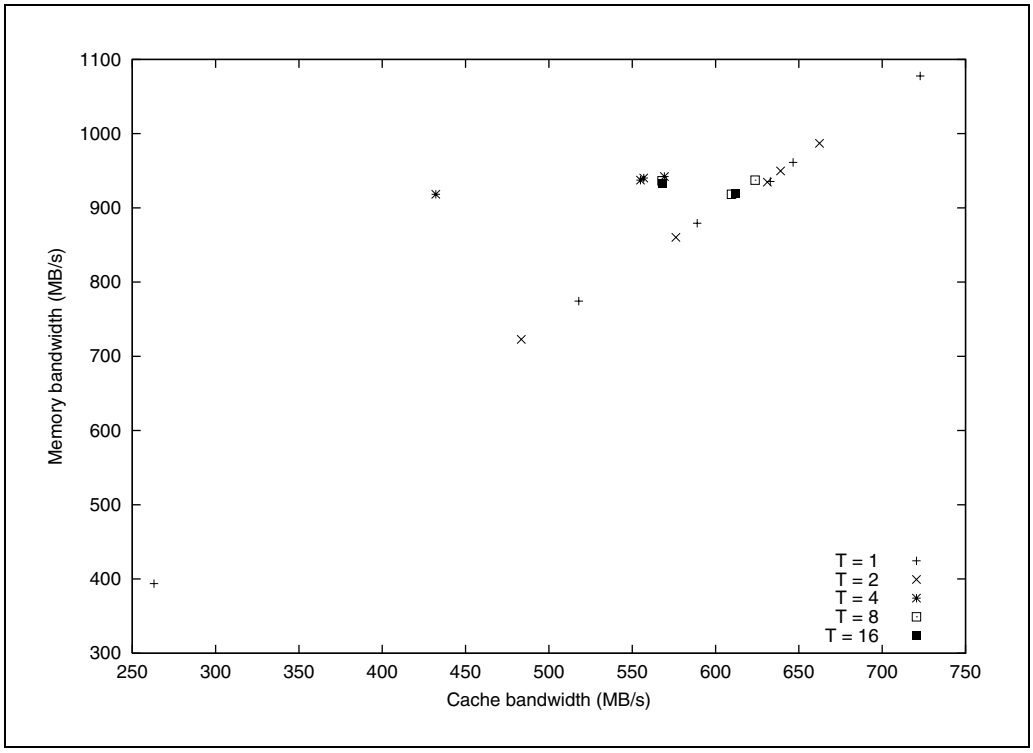


Figure 38: Vector copy, MBW against CBW, current

commands follow the same pattern as an individual processor, i.e. read-read-write, then each 24 cycles of data will have a 4-cycle read-write bubble and an 8-cycle write-read bubble, for an active time of 24/36 = 67%, with 11% and 22% wasted as bubbles respectively.

A single RDRAM device has 32 banks: assuming a uniform random distribution of requests across the banks, the frequency of collisions can be estimated. Each data transfer takes three banks out of service for 20 (i.e. $T_{RC} - T_{CMD}$) RDRAM clock cycles, apart from the edge banks where only two are affected. That is, the expected number of banks rendered unusable is $3 \cdot 28/32 + 2 \cdot 4/32 = 2.875$, and the expected delay for the immediately following command is $20 \cdot 2.875/32 = 1.8$ RDRAM cycles. Ignoring second-order effects, the previous two commands could also collide with this one but with smaller penalties of 12 and 4 cycles respectively, for a total expected delay of 3.23 cycles. Bank conflicts would therefore take up $3.23/(8 + 3.23) = 28.8\,\%$ of the total, and the maximum bandwidth is only 71.2%, i.e. 1086 MB/s (C) or 13,580 MB/s (F).

For this program, with the read-read-write pattern, the bank conflicts may coincide with bubbles; in the case of a simultaneous bubble and bank conflict, it is counted as the former. For each 8 cycles of data, 1.33 cycles of read-write

bubbles, 2.67 of write-read bubbles and 2.28 cycles of bank conflicts are expected, i.e. 56.0%, 9.3%, 18.7% and 15.9% respectively. This equates to 855 MB/s (C) or 10,691 MB/s (F).

In practice, the vector copy program does slightly better than this, since the bank distribution is not completely randomised, and any read or write clustering will reduce the number of bubbles. With at least eight threads, the memory bandwidth remains in the range 918 MB/s (60.2%) to 987 MB/s (64.7%), apart from the single (P=8, T=1) result which suffered an unusually small number of conflicts.

For the cache bandwidth CBW, the best results are obtained with only one thread per processor, where this is enough to saturate the channel (P=8, P=16). With four processors two threads each are best, although this could be due either to one thread's not being enough or to eight total threads' being a sweet-spot for bank conflicts. All the saturated results lie in the range 555 to 723 MB/s.

The 'future' results track much more closely with the total number of threads, independent of their arrangement on processors. The memory channels never become completely saturated; instead they asymptotically approach saturation, and utilisation (Act%) levels out at just over 60%, as for current parameters. Bank conflicts and bubbles occur as before.

Cache interference appears to be much less of a problem, with the MBW/CBW ratio remaining between 1.48 (P=32, T=1) and 1.57 (P=2, T=16). Indeed, achieved bandwidth (both CBW and MBW) improves monotonically with both processors and threads per processor. This suggests that multithreading offers a more reliable performance with the future parameters than with the current ones, in the sense that adding more threads is unlikely to slow execution. The latency of memory operations, which is being successfully hidden with 32 total threads, is very large: several thousand cycles. This increased latency is caused by limited memory bandwidth: there are large queues of requests for each of the memory channels.

Bus utilisation is low for the current parameters (less than 13%), but becoming high for the future machine (up to 63.5%). In each case, increasing the throughput on this benchmark would require more RDRAM bandwidth; the

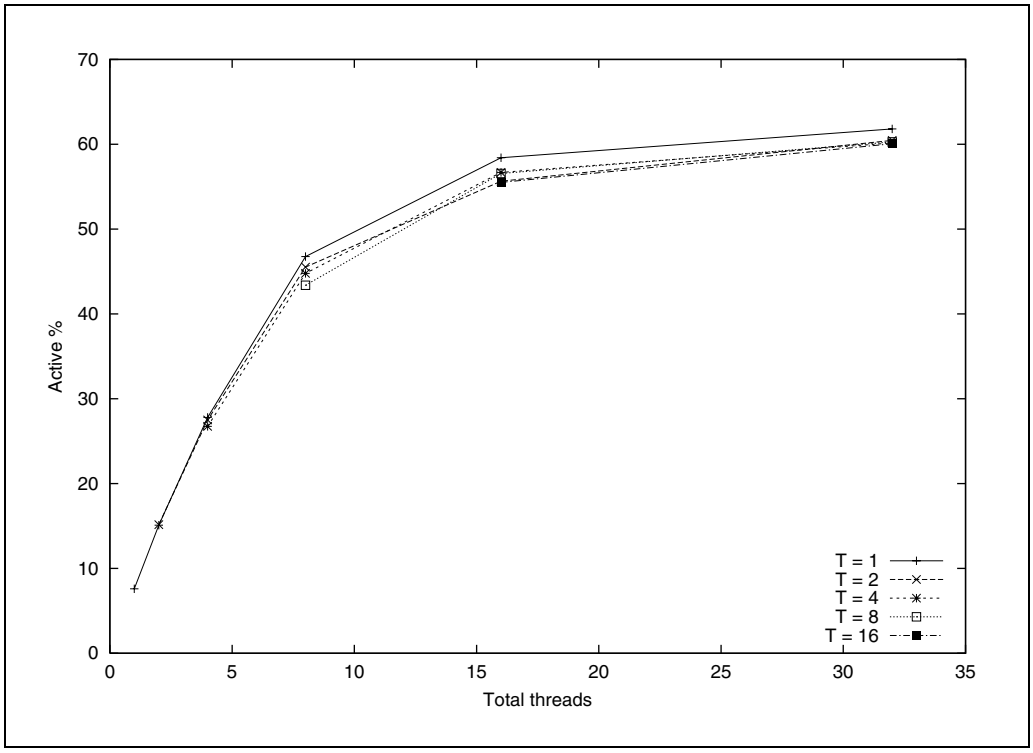


Figure 39: Vector copy, Act%, future

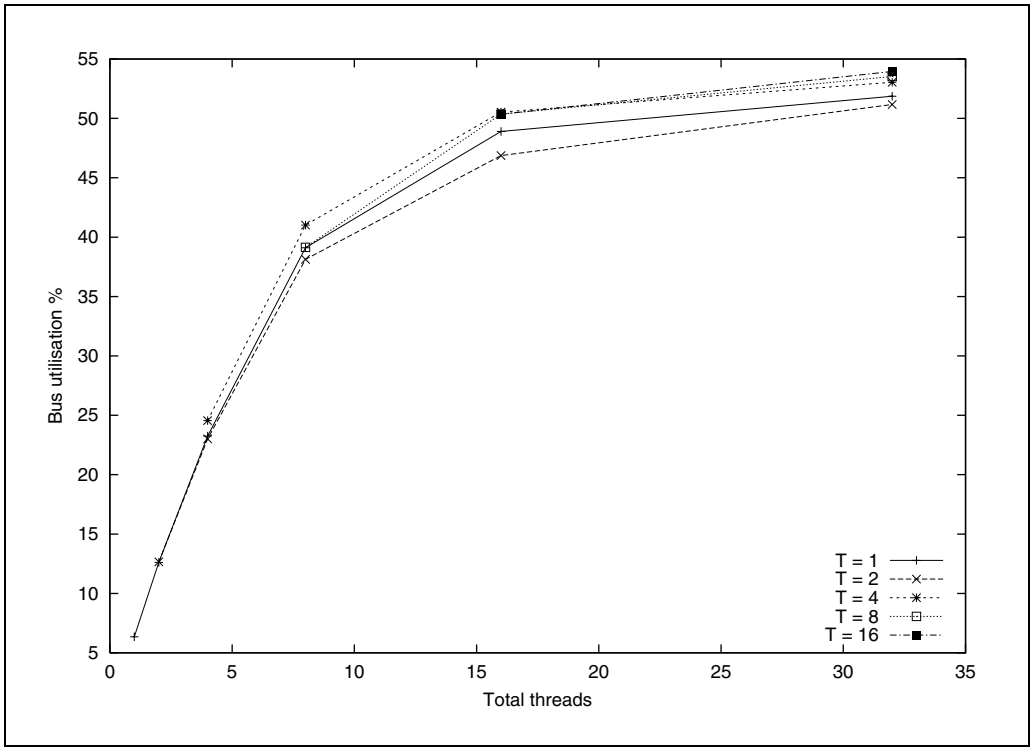


Figure 40: Vector copy, bus utilisation, future

future machine would also need faster on-chip communication, say by doubling the cache line and bus width, using two independent buses, or moving away from a shared bus entirely.

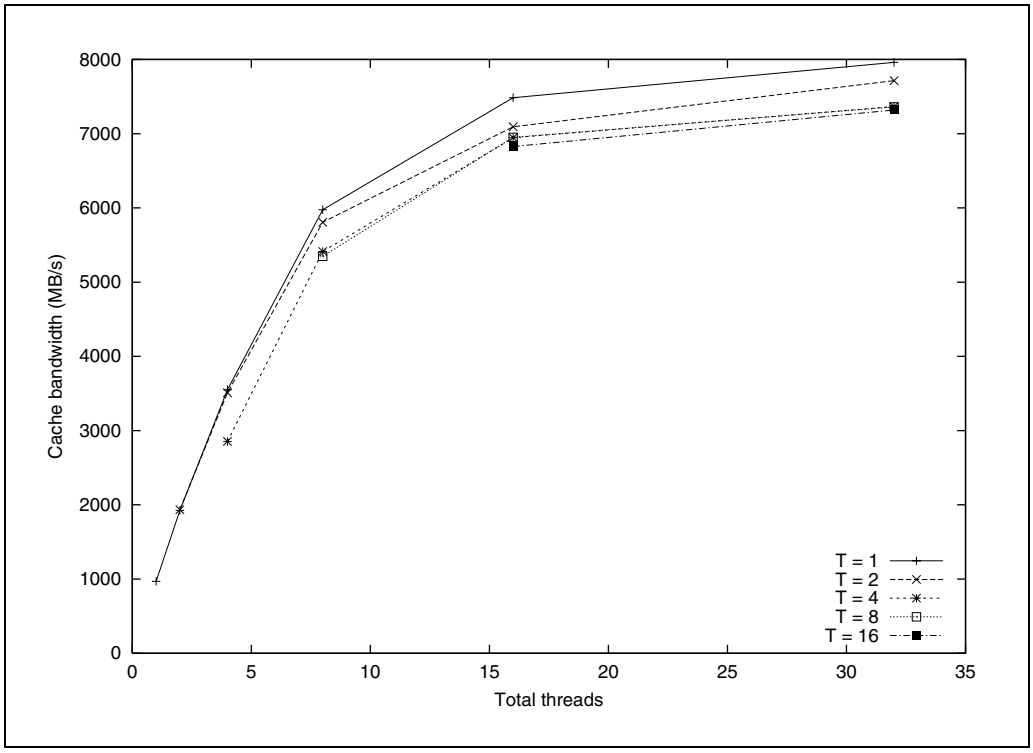


Figure 41: Vector copy, cache bandwidth, future

9.1.2 Vector addition

The addition loop is similar to the copy loop, and is shown in Figure 42. Another pointer is the only extra feature. In this case, the CPU-cache traffic will total 12MB (two reads and one write per iteration), and the memory will handle almost 16MB (two shared reads, one exclusive read, and one writeback).

```
L1: ! copy loop
  ldl %i4, 0(%i1)
  add %i1, 4, %i1
  ldl %i5, 0(%i2)
  add %i2, 4, %i2
  add %i5, %i4, %i5
  stl %i5, 0(%i3)
  add %i3, 4, %i3
  sub %i0, 1, %i0
  bne %i0, L1
```

Figure 42: Vector add code

Cache interference will be worse in this program, since three sets will be needed in the L1 cache by each thread.

For the current parameters, the memory is saturated with one thread per processor and at least 8 processors. With two threads per processor CBW falls dramatically; the memory bandwidth also falls, due to a large (about 30%) bank conflict time. Both of these effects are due to interference in the L1 caches: each thread manages only one or two additions before one of its cache lines is replaced. The frequent re-fetching of the same lines causes the bank conflicts. Interestingly, performance recovers as more contexts are added.

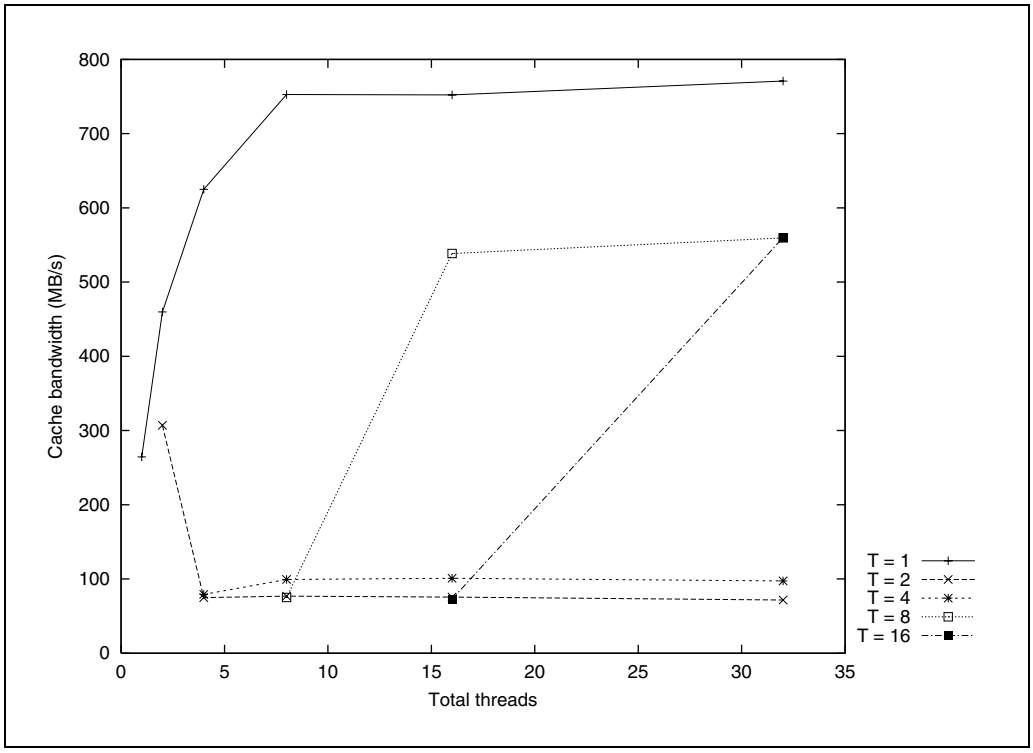


Figure 43: Vector add, CBW, current

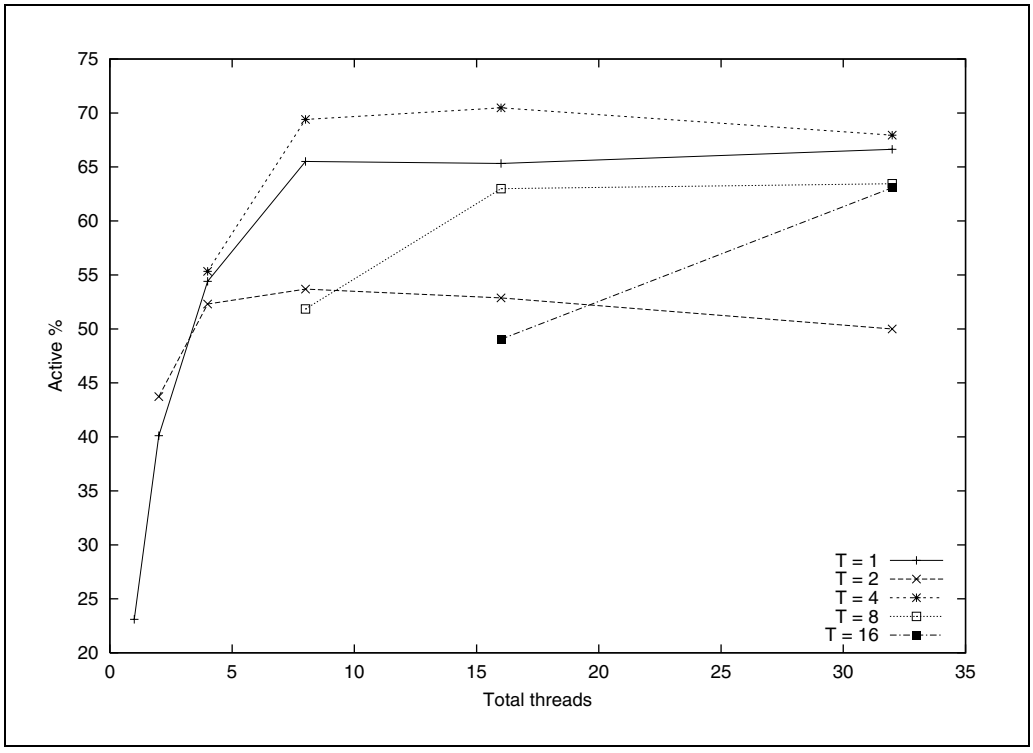


Figure 44: Vector add, Act%, current

The future parameters again produce more predictable results: adding more processors or threads per processor again improves MBW. CBW also increases, except when going from (P=16, T=1) to (P=16, T=2).

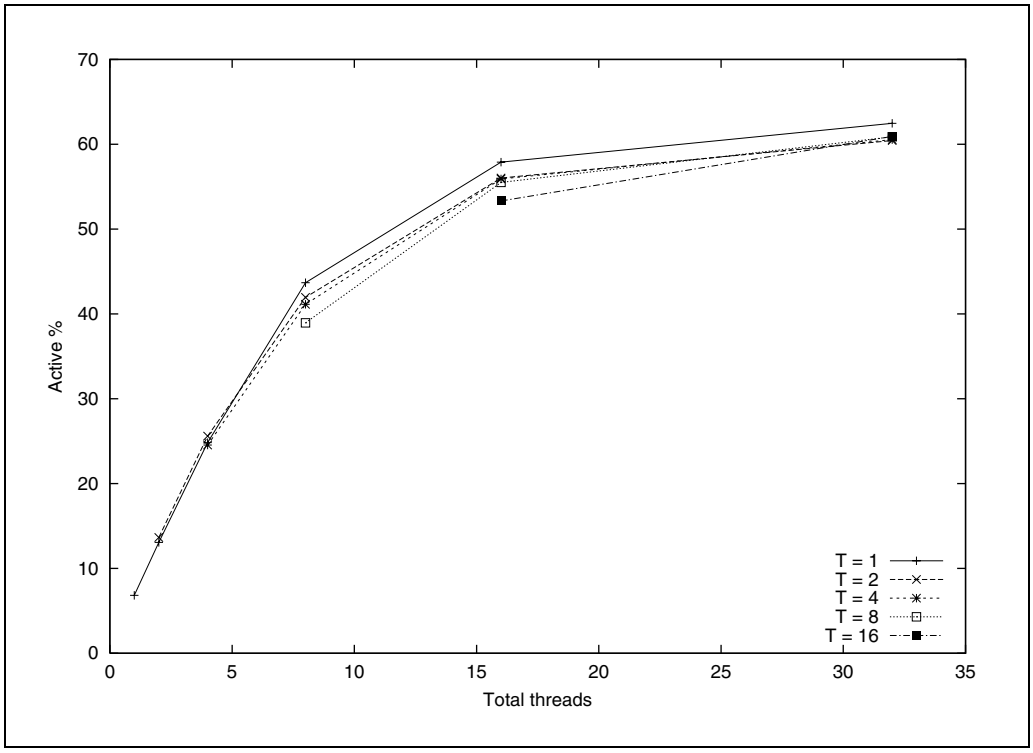


Figure 45: Vector add, Act%, future

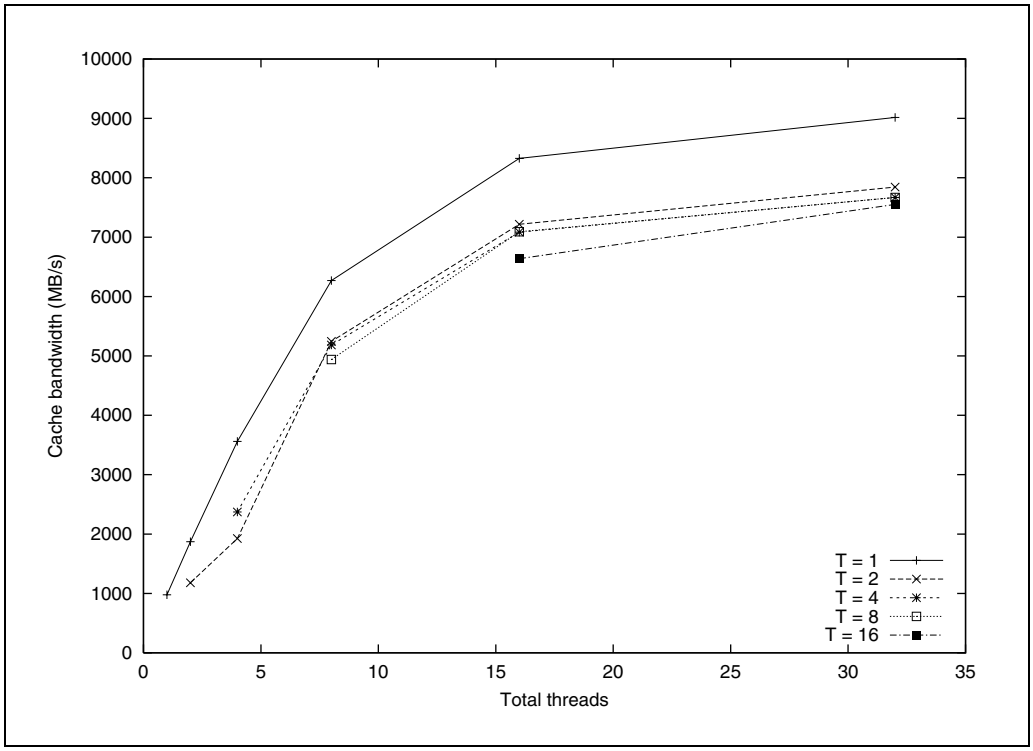


Figure 46: Vector add, CBW, future

9.1.3 Vector c+=a

This loop (Figure 47) is again like the copy loop, except that it loads from the destination vector before writing to it. This means that a shared read will be requested on each cache miss, and an upgrade will be needed if the line is not returned exclusively. For this experiment, two versions of the simulator are used, one in which shared reads do not return exclusive lines.

```
L1: ! copy loop
  ldl %i4, 0(%i1)
  add %i1, 4, %i1
  ldl %i5, 0(%i2)
  add %i2, 4, %i2
  add %i5, %i4, %i5
  stl %i5, -4(%i2)
  sub %i0, 1, %i0
  bne %i0, L1
```

Figure 47: Vector c+=a code

The `%i2` pointer is increased *before* its second use simply to fill the load-delay slot before the addition. CPU and memory traffic will both be about 12MB for this program.

If shared reads give exclusive lines, the MBW and bus utilisation results are within a few percent of those for the copy program, so the graphs are not reproduced. More interesting is the comparison with the modified simulator: disallowing this protocol feature slows the program by about 6% for one or two threads, and results in a 14% increase in bus utilisation (that is, about a 20% increase in the number of transactions, as would be expected). Using 'current' parameters, the more highly parallelised versions behave irregularly: some slow down more (P=1, T=16 is 7% slower), and some even speed up (P=8, T=1 is 3% faster). The speedup is due to a reduction in bank conflicts (e.g. P=16, T=1 is down 17%) or bubbles (e.g. P=8, T=2 down 6%).

The 'future' parameters again act more predictably: every trial runs more slowly, but the performance improves steadily with the number of threads, indicating that the extra delays are being successfully hidden by multithreading/multiprocessing. For example, with 32 threads the CBW is no more than 1% worse. However, the 20% increase in bus traffic still occurs, and in other programs could result in a bottleneck.

9.2 Lock contention

The vector programs in the previous section had several threads working completely independently. In more general applications, threads will synchronise and communicate through shared memory, and the efficiency of this affects the grain size which may be used effectively.

The Bounce program (Figure 48) suffers severe contention on a lock (whose address is in `%i3`). The first section of code shown is basically the `acquire_lock` routine discussed earlier (§8.3.1). At the start of the program each thread gets a serial number (1 up to PT), which is held in `%i1`; all the threads complete to write their number into the lock location.

Once the lock is acquired, another memory location in a different cache line is incremented[1] (`%i2`), and then the lock is released. The loop is executed 1024 times, with each thread doing 1/PT of this total.

```
L1: ! loop
  ldl_l %i4, 0(%i3)
  bne %i4, L2
  mov %i1, %i5
  stl_c %i5, 0(%i3)
  beq %i5, L3

  ldl %i5, 0(%i2)
  add %i5, 1, %i5
  stl %i5, 0(%i2)

  stl %g0, 0(%i3)

  sub %i0, 1, %i0
  bne %i0, L1
[wait for threads to
finish, and exit]
L2: wait
L3: br L1
```

Figure 48: Bounce code

The results are given in Table 16 (current) and Table 17 (future). The execution time in cycles is graphed in Figures 49 and 50, against the total number of threads: smaller is better.

With only one processor, the execution time remains fairly constant at 16.5–17k (C) or 18.5k (F) cycles, since all the lock operations happen within a single cache. As soon as a second processor is added the time almost doubles, and more processors generally produce better results than more threads per processor for any given total. Bus utilisation also jumps, reaching a maximum of 32% (current, at P=8, T=2) or 36% (future, P=32, T=1).

The 'current' run with two threads on each of two processors is anomalous, in that it is not much slower than the single-processor results. It happens that

1. In reality, if an atomic increment were needed then an `LDL_L/ADD/STL_C` sequence would be better than this separate lock.

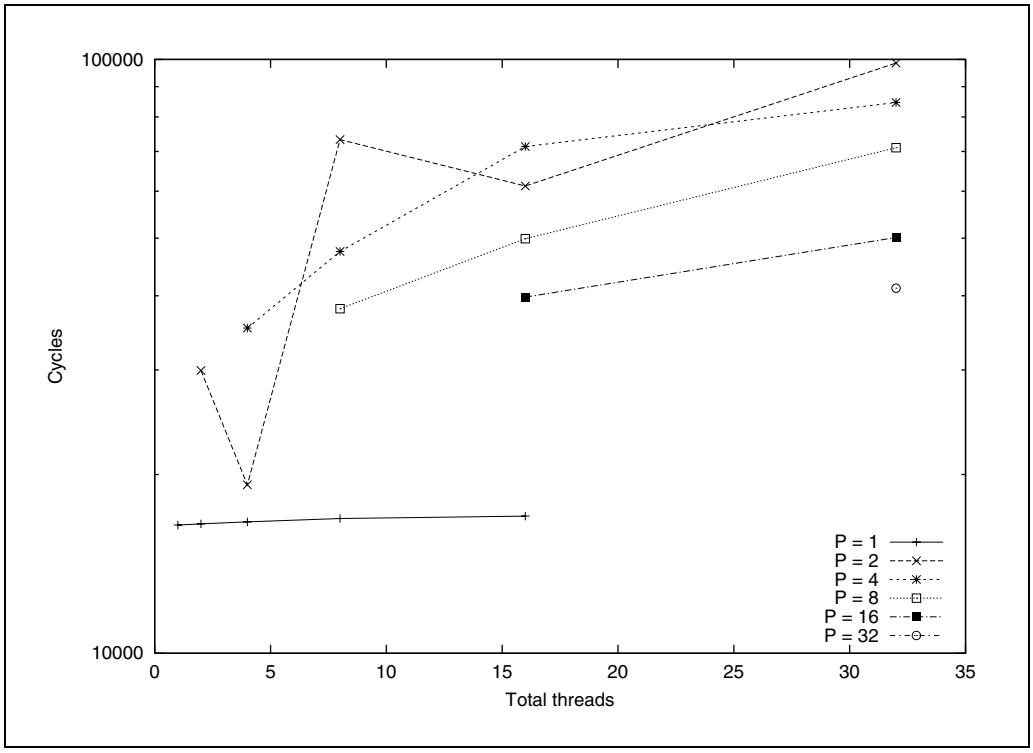


Figure 49: Bounce, execution time, current

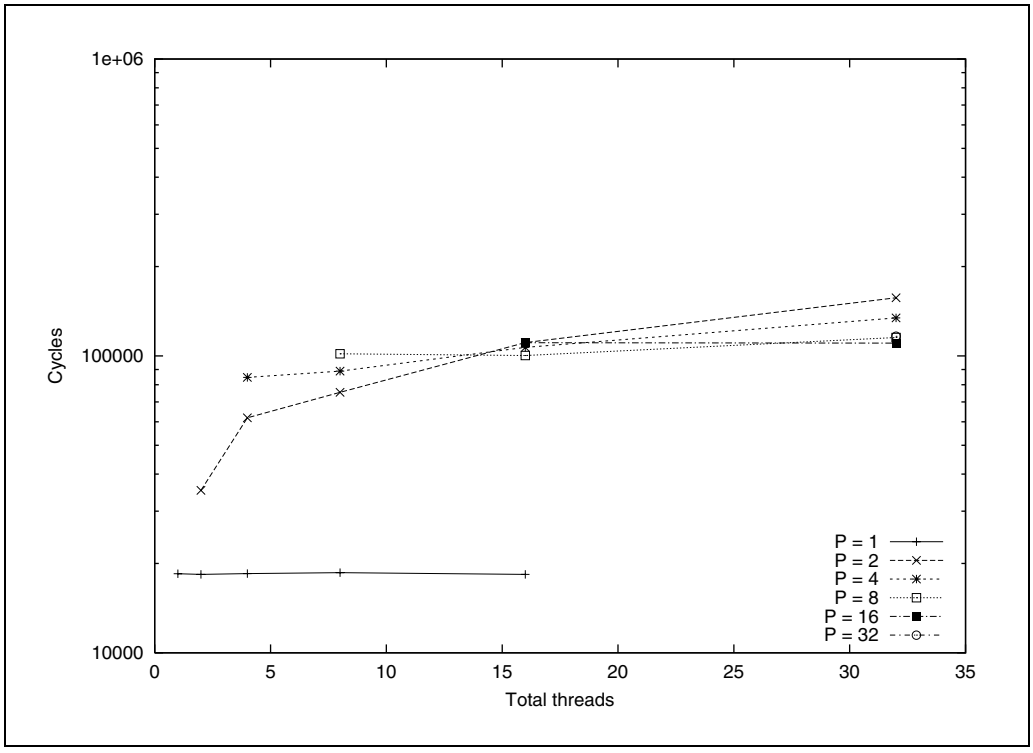


Figure 50: Bounce, execution time, future

the timing is such that the threads on processor 1 acquire the lock alternately, with the `STL_C` (acquire) of one thread and the `STL` (release) of the other occurring as a result of the same upgrade transaction. After 11.4k cycles both of these threads have finished, and then the two threads on processor 0 complete without any further bus traffic. Figure 51 illustrates this, showing the number

of unfinished threads against time. Most other configurations show evidence of similar behaviour, where some group of threads finishes at almost the same time after sharing the lock among themselves to the exclusion of other threads, e.g. eight processor with four threads each (Figure 52).

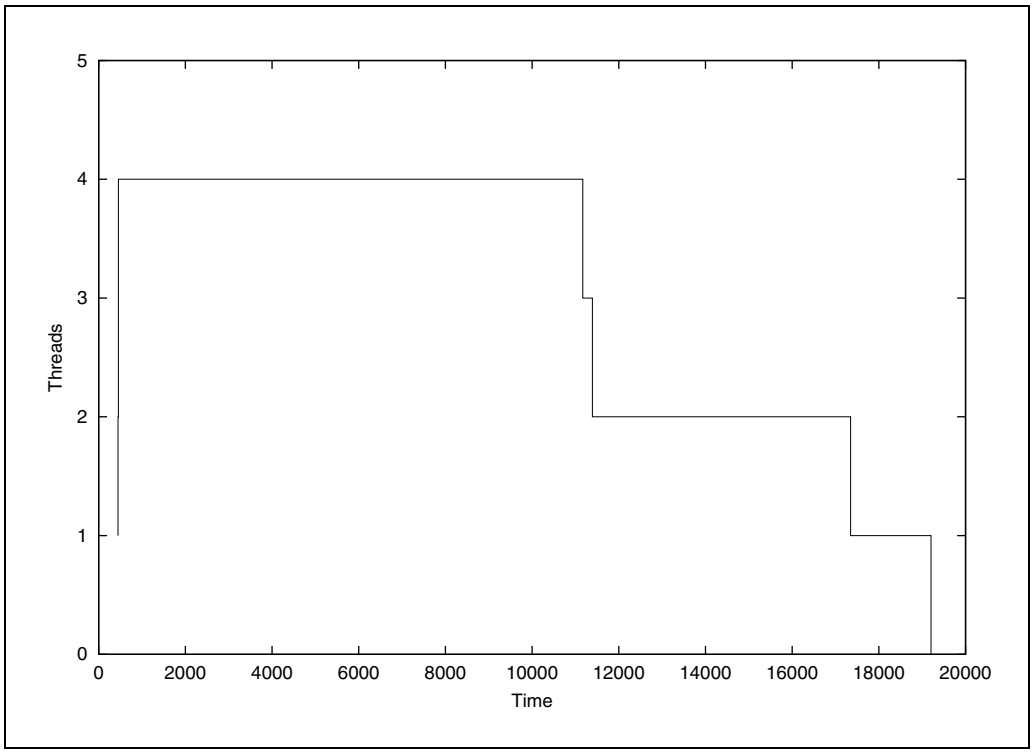


Figure 51: Bounce, current (P=2, T=2) execution profile

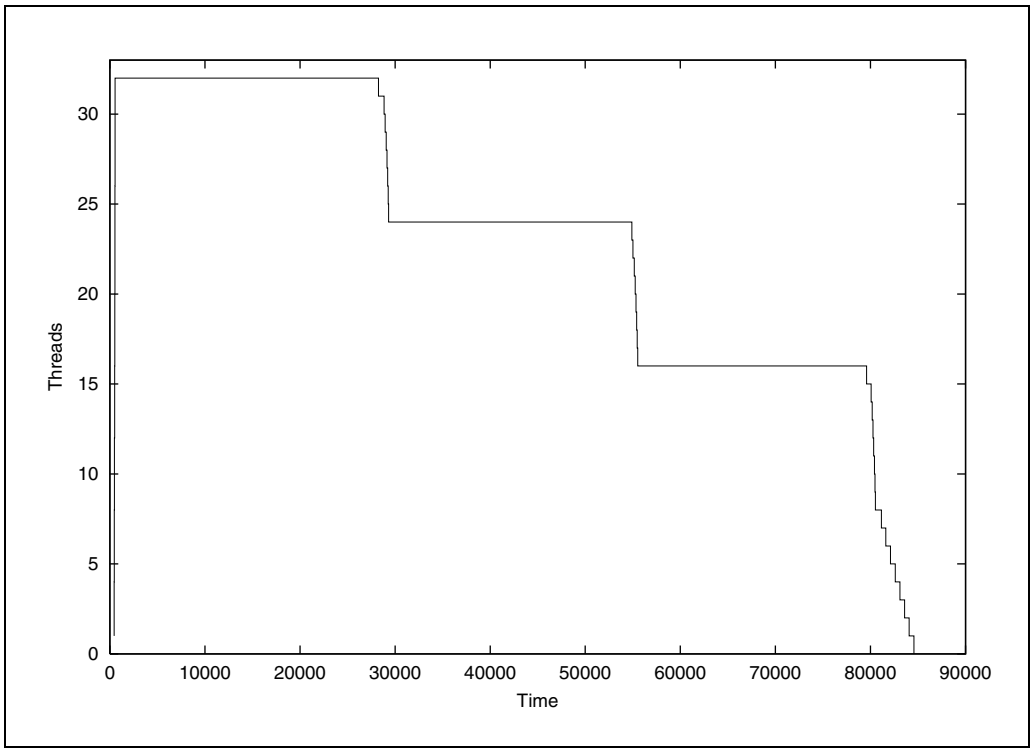


Figure 52: Bounce, current (P=4, T=8) execution profile

The 'future' results show even more unfairness, e.g. Figure 53, where a thread often releases the lock and then immediately reacquires it itself. This is a consequence of the increased shared bus transaction time.

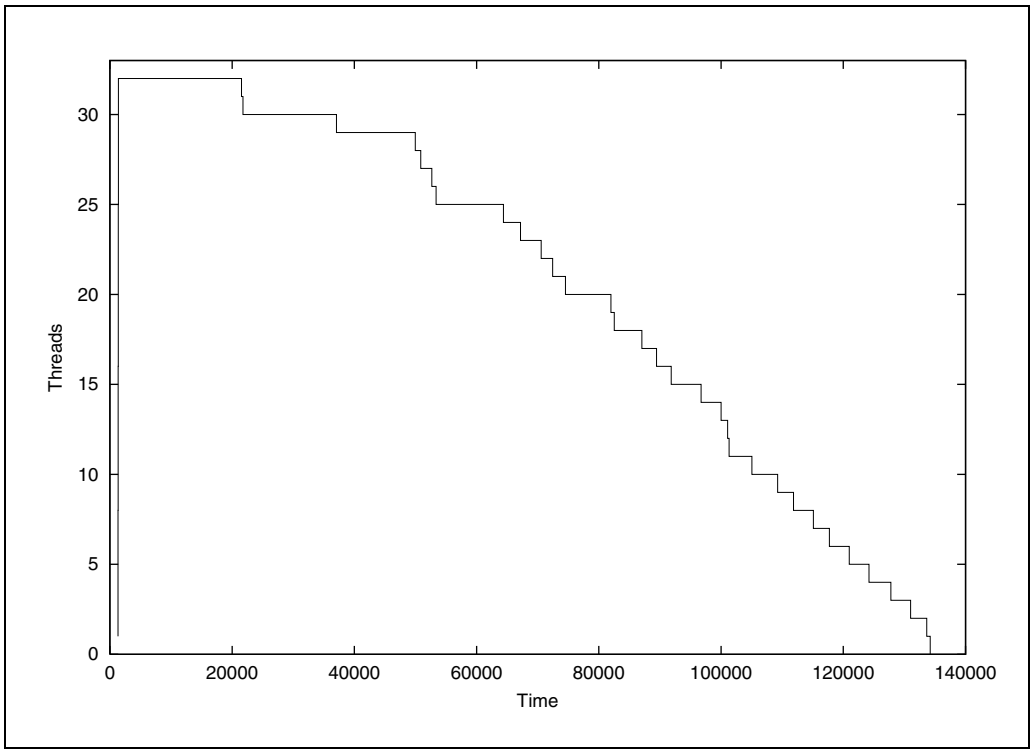


Figure 53: Bounce, future (P=4, T=8) execution profile

The complete acquire-add-release sequence takes, with 32 total threads, an average of 72 cycles (C), or 128 cycles (F). This effectively puts a lower bound on the grain size of any parallel section which needs synchronisation; this time is limited fundamentally by the latency of the bus, and since the purpose is serialisation this latency cannot be hidden by multithreading if all the threads are competing for the same lock.

It should be noted that each of these times is less than that of a single cache miss. If better synchronisation performance is needed, a completely different mechanism would be required; e.g. a separate hardware lock manager which does not work through shared memory ([TLEL99]).

To illustrate the performance advantage of the `WAIT` instruction in a multithreaded system the same program was run with the `WAIT` at label `L2` removed. The results are in Table 18 and Table 19; Figure 54 shows the execution time for the current parameters (the 'future' graph is similar).

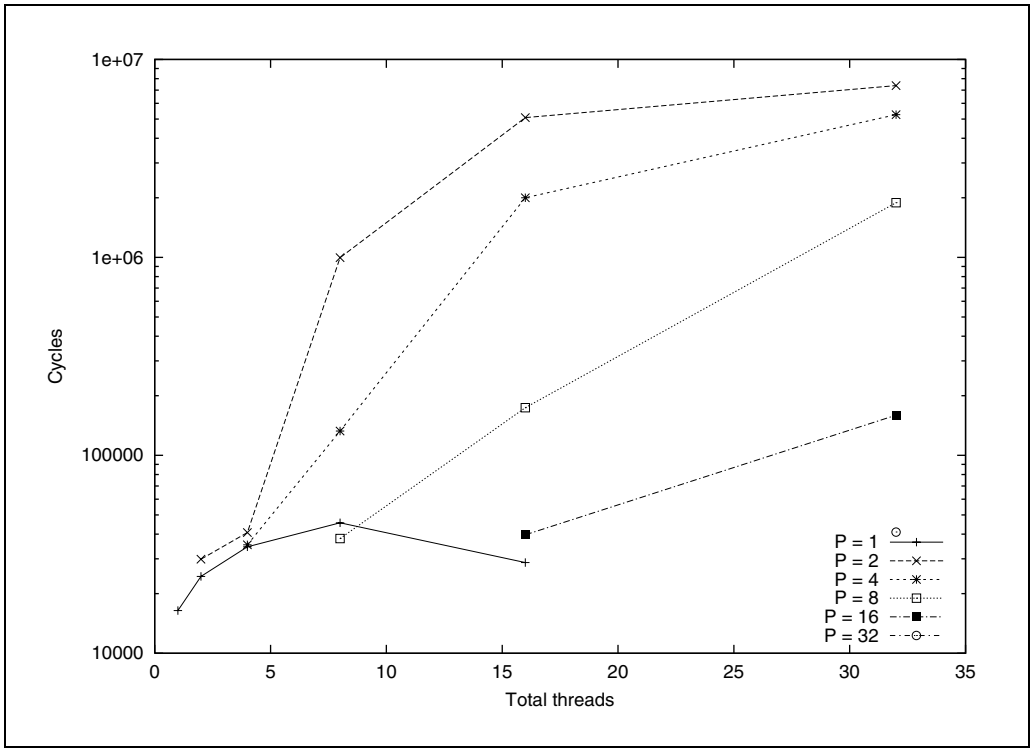


Figure 54: Bounce (no wait), current, execution time

With only one thread per processor, the no-wait program is very slightly faster (around 0.3%); with more than one, the performance is much worse (up to 83 times slower, at P=2, T=8), although with T=2 the slowdown is a more moderate 1.5 to 2.2. The `WAIT` instruction is therefore a big advantage.

9.3 Token distribution

The nfib function is defined by the recursive equations

$$\mathrm{nfib}(n) = \mathrm{nfib}(n-1) + \mathrm{nfib}(n-2) + 1 \qquad (n > 1)$$
$$\mathrm{nfib}(0) = \mathrm{nfib}(1) = 1$$

It is similar to the Fibonacci function, except that adding 1 means that in a direct recursive evaluation the returned value equals the number of function calls performed. Of course, this is not the best way to evaluate this function, but the doubly-recursive structure is similar to the loops parallelised by the modified Javar. The recursion tree is also unbalanced, which is a harder test of the scheduling system.

The assembler code for this program is longer than the previous ones; a C-like pseudocode description is in Figure 55. The program calculates nfib(21) = 35421.

```
int
nfib(int n) {
  if (n < 2) return 1;
  if ((n-2) >= 3 //grain size
      && TRQ) {
    int r1, r2 = 0;
    THB(nfib_child,
      n-2,&r2);
    r1 = nfib(n-1);
    WAIT(r2 != 0);
    return r1 + r2 + 1;
  }
  return nfib(n-2)
    + nfib(n-1) + 1;
}

void
nfib_child(int n, int *r) {
  set up stack;
  *r = nfib(n);
}
```

Figure 55: nfib pseudocode

First the base-case is tested. After that, a comparison is made against a grain size parameter, so that a thread is not forked if it will perform too little work. If the argument is large enough, and there is a remote thread available (`TRQ`), the new thread is started at the `nfib_child` function. This wrapper takes an address into which to place its return value, and also sets up the new thread's stack (using a memory address calculated from the result of `gettid`, a PALcode routine which returns a unique number identifying the context executing the thread). The other recursive evaluation is done locally, then the remote thread's result is awaited.

By altering the grain size (GS) parameter, 3 in the above code, the behaviour of the system under differing 'shapes' of task can be compared. With the value 3, all invocations down to nfib(5) will attempt to fork, a total of 4180 possibilities (i.e. $\text{fib}(n - GS) - 1$, where fib is the normal Fibonacci function). The smallest unit of work is evaluating nfib(3), about 40 instructions. Pushing GS up to 9 gives 232 possible forks, and nfib(9) is about 1000 instructions. Apart from the results transmitted through shared memory, this program will suffer no data cache misses.

The results are listed in Table 20 and Table 21, for two versions of the simulator. The first has the normal token mechanism; the second uses a token 'oracle,' in which a TIU can remove a token from anywhere on the ring, not only its own position. This simulates an (unimplementable) fully-connected system, where knowledge of a context's availability is propagated instantaneous-

ly. The (in)efficiency of the token mechanism can thus be assessed. To remove the effects of register spilling and filling for now, each processor has 80 windows (640 registers) for these experiments. A more realistic number is considered in §9.4.

In the tables, SpUp is the speedup relative to a serial program with all the thread-handling code (grain size test, `TRQ`, etc.) removed. The number of threads forked (Thr) and their creation rate (i.e. execution time divided by this number) are also shown.

With 'current' parameters, the fine-grained threads have a slight advantage (Figure 56), with 4 threads/processor being optimal. For the 'future' machine (Figure 57), the coarser grain is clearly better; the fine-grained program has saturated the bus by 32 processors. Each thread forked could require up to four bus transactions (i.e. context send, shared read by caller, upgrade by callee, and a second shared read), and at this point the creation rate is close to or even below the time for four. Both configurations with one processor lose about 14% to the serial version; in this case, multithreading offers no benefit because there are no cache misses.

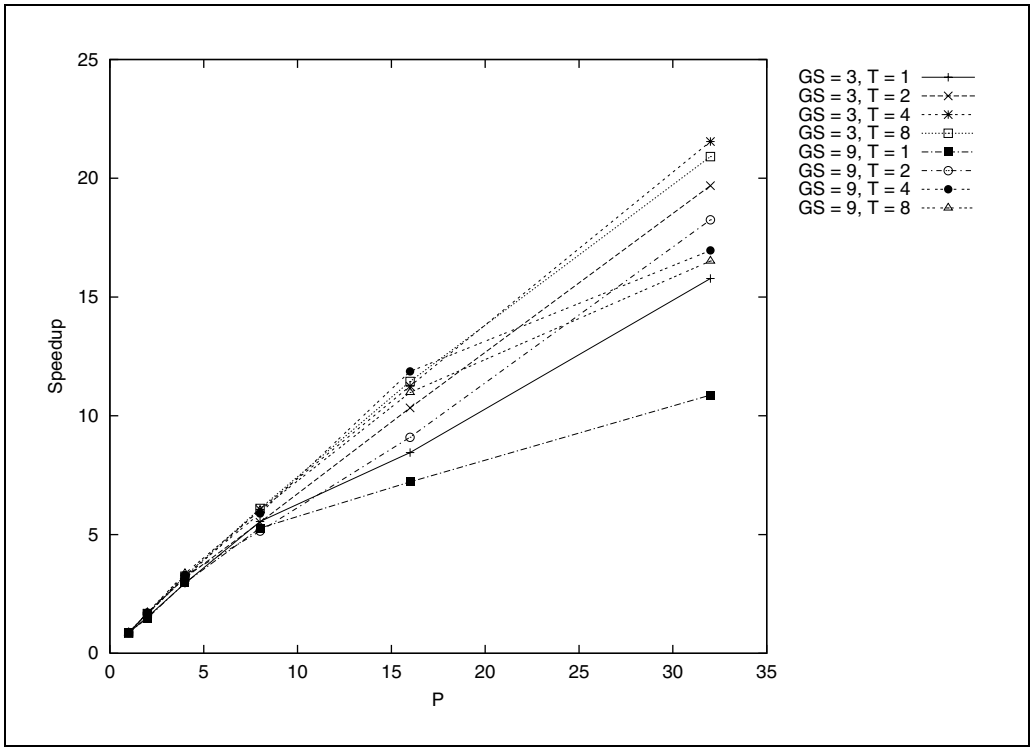


Figure 56: nfib, current, speedup

The oracle is significantly faster than the token mechanism when there is only one context per processor, but adding a second closes the gap. With at

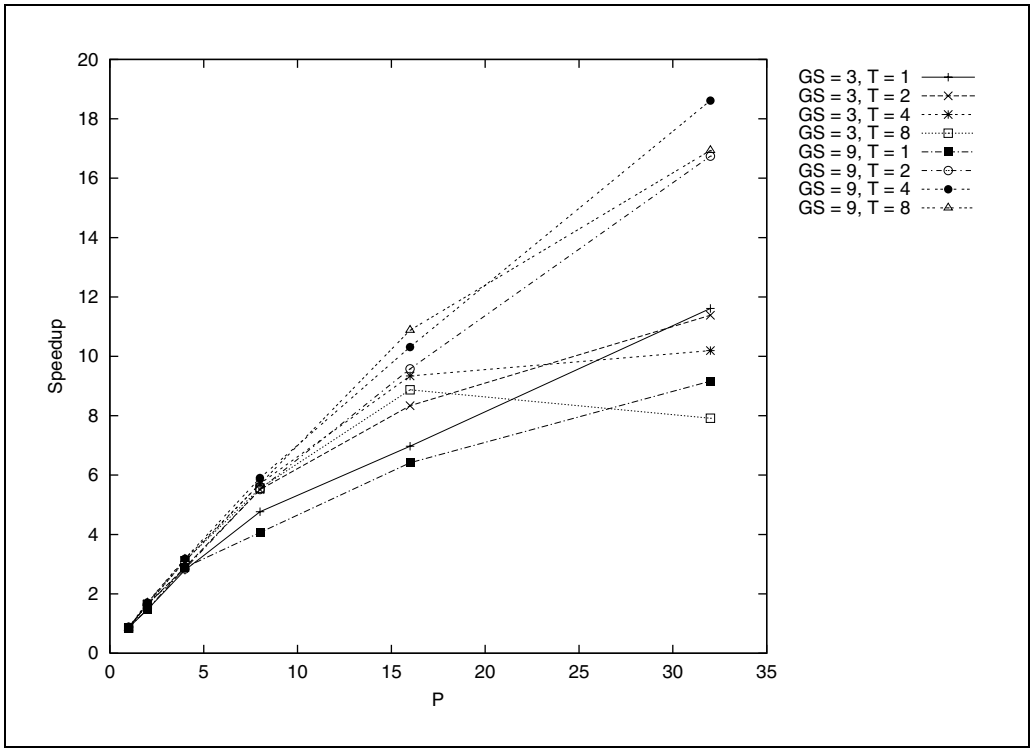


Figure 57: nfib, future, speedup

least four contexts per processor and 32 processors, the oracle is still noticeably better only for coarse-grained tasks with current parameters; with future parameters, the token mechanism is slightly better.

There are several effects which account for this behaviour. The first is that a processor does not need to have all of its contexts occupied to run at full efficiency, particularly when cache misses are scarce. Thus, it does not matter if the token mechanism fails to allocate the final context; when all the processors have, on average, at least one token in circulation then the ring will be full and every `TRQ` will succeed. The second, a consequence of this, is that it is not always beneficial to create more threads; if a processor is saturated, then forking a new small thread to it only adds overhead. The token and oracle mechanisms will act similarly in the early stages of the calculation, when the tasks being distributed are large, but as the contexts fill up the token mechanism will miss forking some smaller tasks when the oracle would have done.

The speedups achieved on nfib are encouraging: this program is hard on the token allocation and thread distribution mechanisms. For example, in the maximum speedup cases, a `TRQ` is being performed somewhere on average every 5.7 cycles (current, P=32, T=4) and 10.8 cycles (future, P=32, T=2). Similarly, a thread is created (and equally, a thread terminates) on average every 13.9 cy-

cles (C, P=32, T=8) and 31.8 cycles (F, P=32, T=8). Both of these rates would be impossible to sustain through any data structure in shared memory.

The limiting component in this program, for fine-grained threads, turns out to be the shared bus, which is passing the results back to the threads' invokers. To get better performance with such small threads would need some alternative synchronisation mechanism to pass results back. Since creating more threads does not help when the bus is saturated, a dependence could also be introduced into each TIU so that token requests will fail when bus utilisation is high. These ideas are not explored further here.

To present an idea of the 'shape' of the calculation, and the sizes of the threads created, Figures 58 to 60 show histograms of the life-cycle of a context for GS=3, P=32, T=4 with current parameters. Figure 58 shows the times elapsed between a token's creation and another thread allocating it; during this time a context is idle. The mean for the program is 318 cycles. Figure 59 gives the times between the allocation and the new thread's being marked runnable; the mean is 233 cycles. Finally, Figure 60 shows the lifetimes of threads, i.e. from being started until the token is again released; this averages 2,831 here. These correspond to a, f and w from §5.2.3. The last two graphs have long 'tails' to the right (not shown), where a small number of threads have unusually long delays, caused by context-switching during the fork operation, or long lifetimes. It should be noted that these times disregard how the thread was scheduled during these times; so, for example, during most of its lifetime a thread may have been waiting for its children, and not consuming any processor cycles.

The graphs for GS=9 are similar, but all the times are greater: the mean times for release-allocate, allocate-start and start-release are 5,645 cycles, 503 cycles and 22.3k cycles respectively. The release-allocate time thus appears to be proportional to the minimum grain size, as might be expected if the token-request interval is also proportional. The actual grain size (start-release) is much larger than the minimum because of the forks which are not taken towards the leaves of the tree.

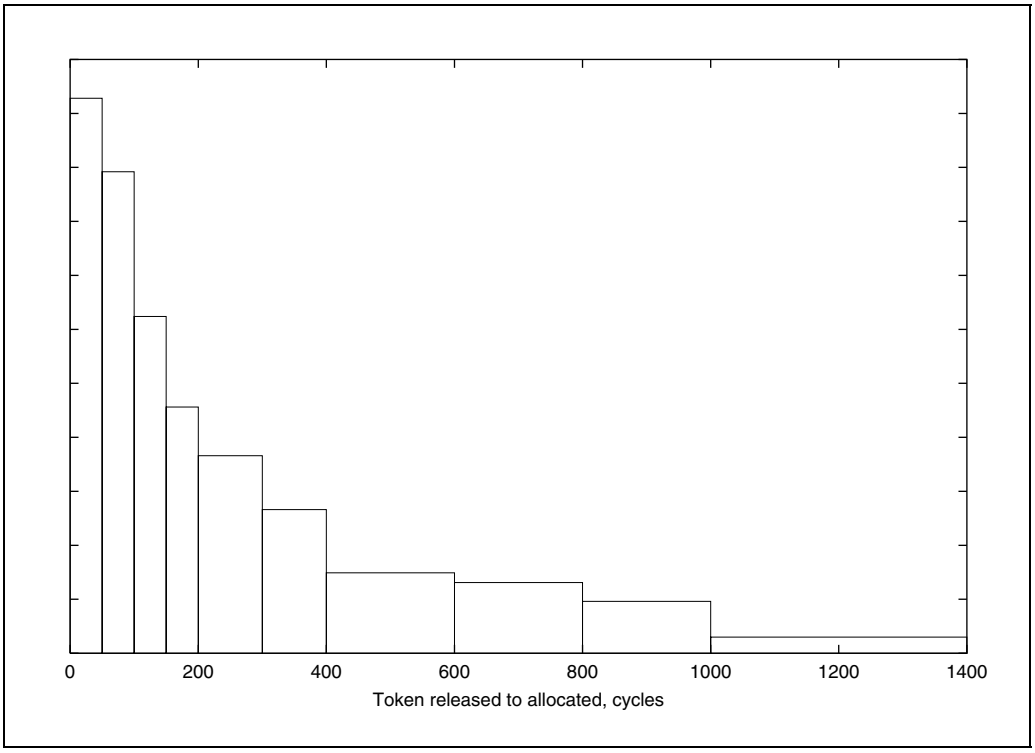


Figure 58: nfib, token released to allocated

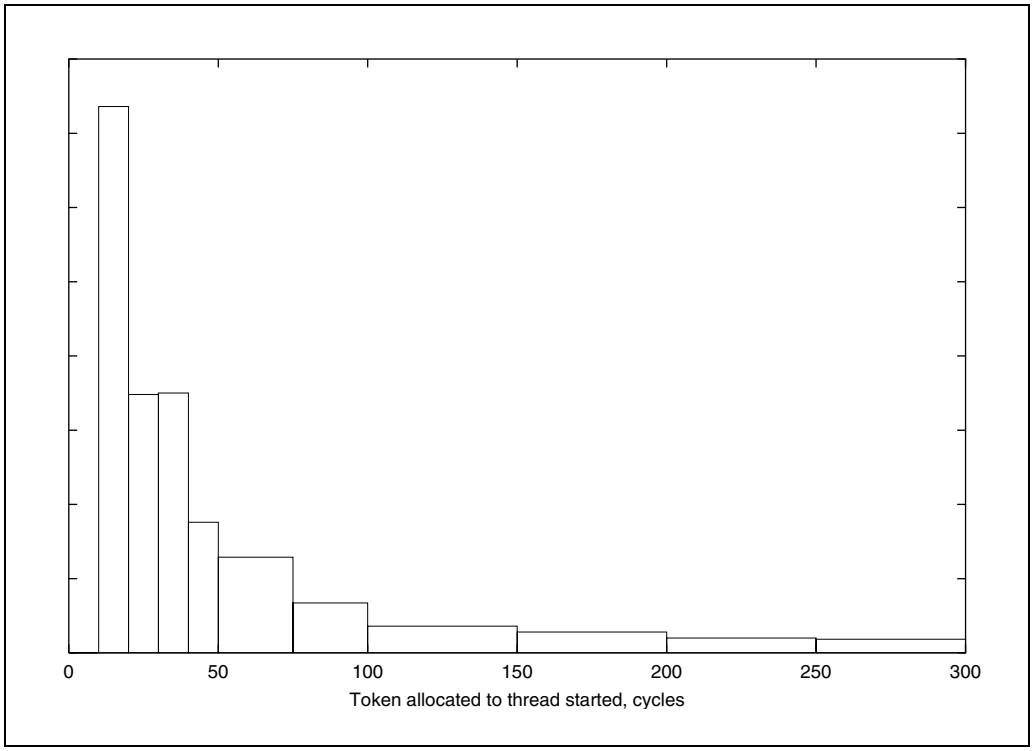


Figure 59: nfib, token allocated to thread started

9.4 Heap-allocated register windows

The nfib program, in the previous section, used 80 register windows per processor, which resulted in practically no register spills (a small number did

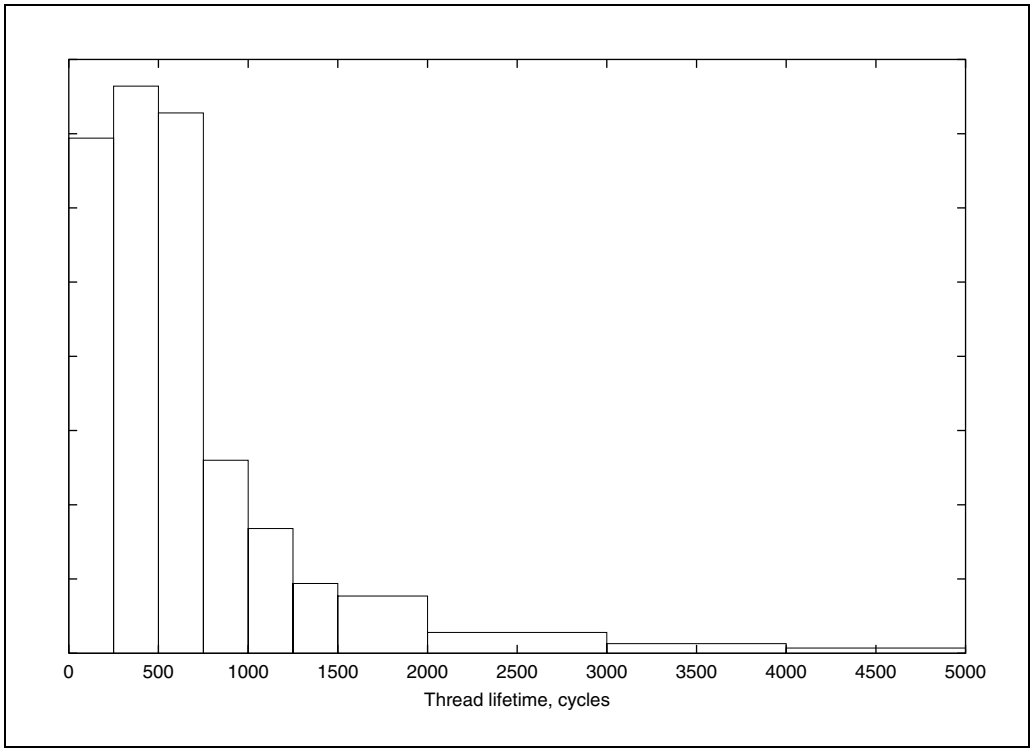


Figure 60: nfib, thread lifetime

occur in some runs with 8 threads per processor, but accounted for less than 1% of the runtime). This number is rather unrealistic, and from now on it is reduced to 24. This equates to 196 individual registers, similar to the number in a conventional superscalar processor with register renaming. Not all of these need be heap-allocated; the Global and Extra windows do not shift, so for, say, 4 contexts per processor this could be implemented as 64 static registers and 128 as a heap. However, these simulations assume that all the frames are in the heap; if a thread happens not to need its Extra window, then this makes it available for another use.

The results using only 24 windows are in Table 22 and Table 23, for two versions of the simulator. The first partitions the frames equally between the contexts, so for example with four hardware contexts each would have 6 available. For this program, which does not use the Extra window, this is equivalent to each context's having a single Global frame and a 5-entry circular buffer. The second simulator models the heap allocation described earlier. With only one context per processor these produce identical effects.

The speedups are relative to the serial program running on a single-threaded uniprocessor; this takes 516,011 cycles (current), or 516,791 (future). For comparison, the same serial program (current parameters) takes 516,539 cycles

on a 4-context uniprocessor with heap allocation (i.e. 0.1% more), and 555,215 on a 4-context uniprocessor with static partitioning (7.6% longer). In the latter two, of course, the additional contexts are not being used, they simply reduce the number of registers available to the main thread. Since nfib(21) recurses to a maximum depth of 22 the non-multithreaded case causes no spilling, and is the same as the earlier 80-window results; the speedups are therefore directly comparable with Table 20 and Table 21.

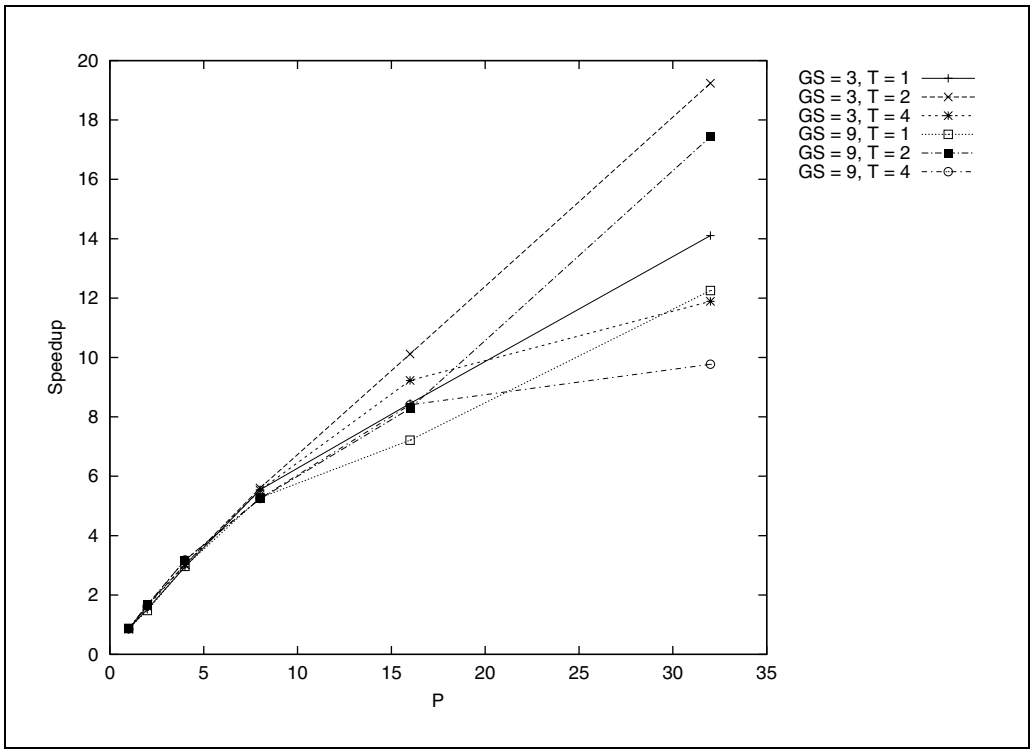


Figure 61: nfib, current, 24 windows: speedup

For the current configuration, fine-grained threads and two contexts per processor, the results are close: heap allocation has a 3.9% advantage for P=8 and 8.3% for P=32, while static allocation has 2.8% for P=16. With four contexts, the gap widens: heap allocation is 9.0%, 5.2%, 6.3%, 9.9%, 14.0% and 22.6% faster for P=1 to 32, showing a clear advantage.

The 24-window results are very close to the 80-window results for two contexts per processor (e.g. 80 windows is 2.1% faster for P=16 and 2.3% for P=32). The biggest change is with four contexts; whereas the best speedups with 80 windows were achieved using four, with only 24 the results fall away sharply: for P=16, 80 windows is 21.9% faster, and 81.2% faster for P=32. The reason is compulsory (cold-start) cache misses when the spills take place; the memory bandwidth for P=32 has jumped up to 827 MB/s from 196 MB/s. By changing

the program to calculate nfib(21) twice, the effect of a warm cache can be seen: the first nfib takes 44,164 cycles, and the second only 25,709. The latter figure is only 10% slower than the 80-window result. This cold-start effect also magnifies the difference between static and heap allocation: the second nfib is only 9.8% faster with heap allocation, rather than the 22.6% mentioned earlier.

The same pattern is repeated with the coarser-grain program and future parameters. These results suggest that, even for highly recursive programs, 24 windows per processor is enough. However, heap-allocated register windows offer only a small advantage over static partitioning.

CHAPTER 10. Java benchmarks

The previous chapter contained hand-written assembler benchmarks which examined only certain aspects of the system. Here, the programs are written in Java. This introduces the restructuring compiler Javar, the translator jtrans and the runtime system into the equation. How well does the system perform on automatically parallelised code and larger, more complex, applications?

10.1 jnfib

To compare the performance of the runtime system managed Java threads with the raw hardware ones in the previous chapter, the first Java program is simply nfib recoded. The source code (Figure 62) has been parallelised by hand; although the original Javar can automatically parallelise recursive functions like this, the modified Javar does not support this at present. Nevertheless, the code is almost exactly what would be produced.

The results are in Table 24 (current) and Table 25 (future), for the two different runtime systems: recall from §8.5 that 'light' idle threads are found using the hardware token passing, while 'medium' ones are removed from a software queue. Both RTSs then pass the arguments across the bus with `THB` in the usual way. It should also be remembered that if there is more than one thread in the system then the second is always a scheduling thread (§8.5.4); this is why the two-thread results are so close to the one-thread, and why no threads are forked.

```
import parlooppack.*;

class Nfib3 {
  public static void main(String args[]) {
    System.out.println(run_nfib(21));
  }
  public static int run_nfib(int x) {
    if (x < 2) {
      return 1;
    } else {
      LoopWorker_Nfib3 w = null;
      int t = 0;
      if (x >= (3+2)
          && Mysys.free_proc()) {
        w = new LoopWorker_Nfib3(x-2);
      } else {
        t = run_nfib(x-2);
      }
      t += run_nfib(x-1);
      if (w != null) {
        w.light_join();
        t += w.result;
      }
      return t+1;
    }
  }
}

class LoopWorker_Nfib3
    extends parlooppack.LoopWorker {
  int result;

  LoopWorker_Nfib3(int low) {
    this.low = low;
    light_start();
  }
  public void run() {
    result = Nfib3.run_nfib(low);
  }
}
```

Figure 62: jnfib source

The speedups are relative to a serial Java program, which took 571,221 cycles (current), or 583,613 (future); this is only about 10% slower than the serial nfib program in hand-written assembler, §9.3. All of the results discount the RTS initialisation time, which is about 0.6M (current) or 1.1M (future) cycles, and only start measuring with the call to `main()`.

Starting with the 'current' results and fine grain size, both the light and medium RTSs run at two-thirds of the speed of the serial version. With one thread per processor and the light RTS, the multiprocessor speedups are between 0.5 and 0.6 of what the assembler version achieved. Multithreading, as with the assembler nfib (§9.4), does not improve the results much; with 32 processors it makes it worse, and the best result measured is P=32, T=1 with a speedup of 7.12.

The medium RTS's single-threaded performance is similar up to 8 processors, and then falls away: its 32-processor performance is only 51% of that of the light RTS. With multithreading, this figure is even worse; with four threads per processor on at least 16 processors it is under 20%, and the serial program is actually faster.

For the coarse-grained program the speedups are better than for the fine-grained; this is the opposite of the assembler program, where fine-grained threads were the better. With multithreading and larger numbers of processors the light RTS is 10 to 54% (T=2) and 49 to 71% (T=4) faster than the medium. Figures 63 and 64 show the 'current' results.

The 'future' results are worse, in terms of speedup, than the 'current' ones for both RTSs, and the light system is a bigger improvement over the medium one; the light one is more than 3 times faster for at least 16 processors on the fine-grained program. The coarse results are closer, but the medium RTS is still clearly slower. Once again, coarse grained threads produce better speedups. Figure 65 and Figure 66 are graphs of these.

By examining some executions in more detail, the limiting factors can be determined. For the current simulator, the fine-grained, light RTS program with P=32, T=2 is spending only around 1% of its time spilling and filling register windows. The memory bandwidth totals 633 MB/s, 39% of peak, and 65% of the processing time is spent idle. Of this idle time, 23% is spent waiting for

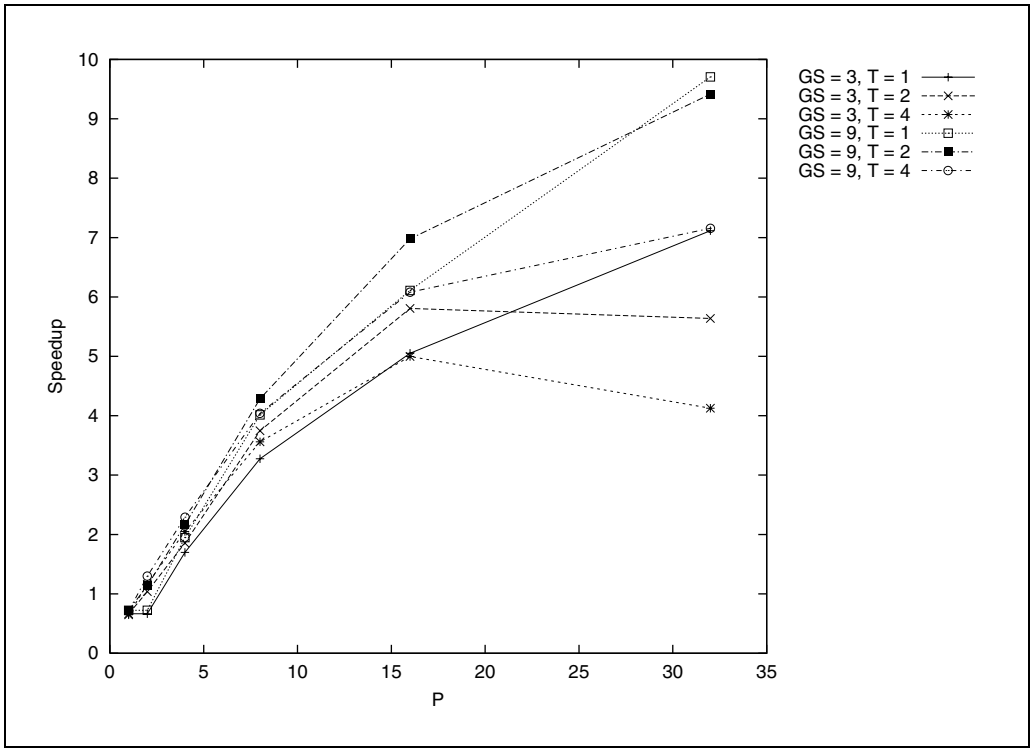


Figure 63: jnfib(21), current, light RTS

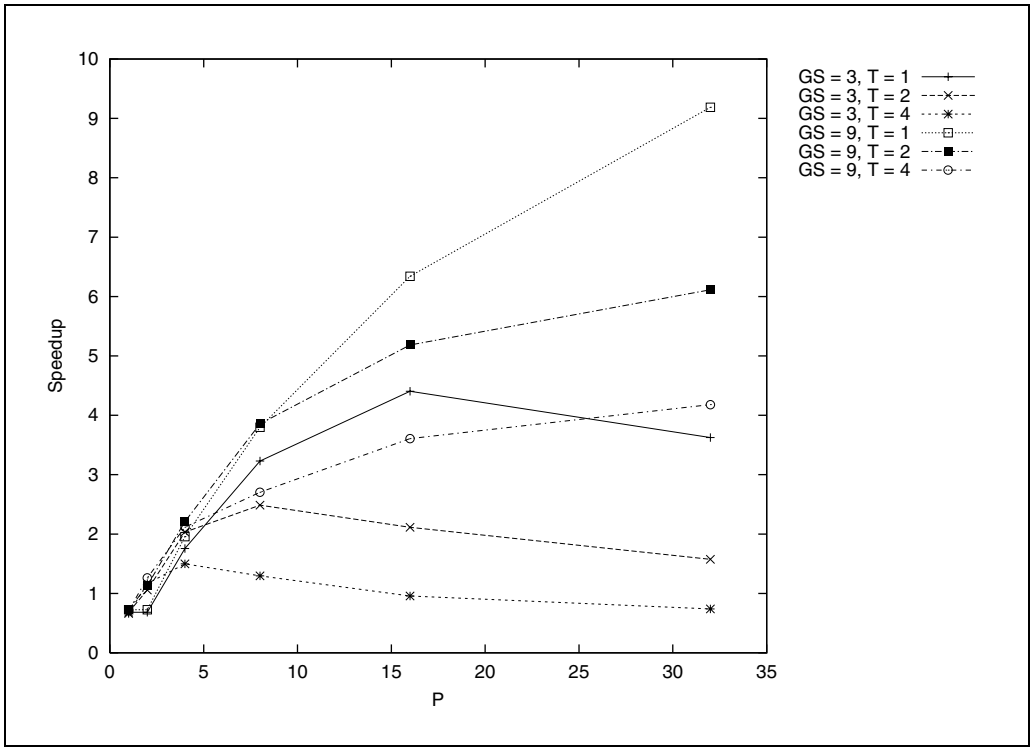


Figure 64: jnfib(21), current, medium RTS

memory operations (whether loads, stores or explicit `WAIT`s), and 42% is from a processor's having no threads resident. The medium RTS also remains 67% idle, but the number of instructions executed is much higher: 1,854,744, compared with 728,706 for the light RTS and 365,080 for the serial program.

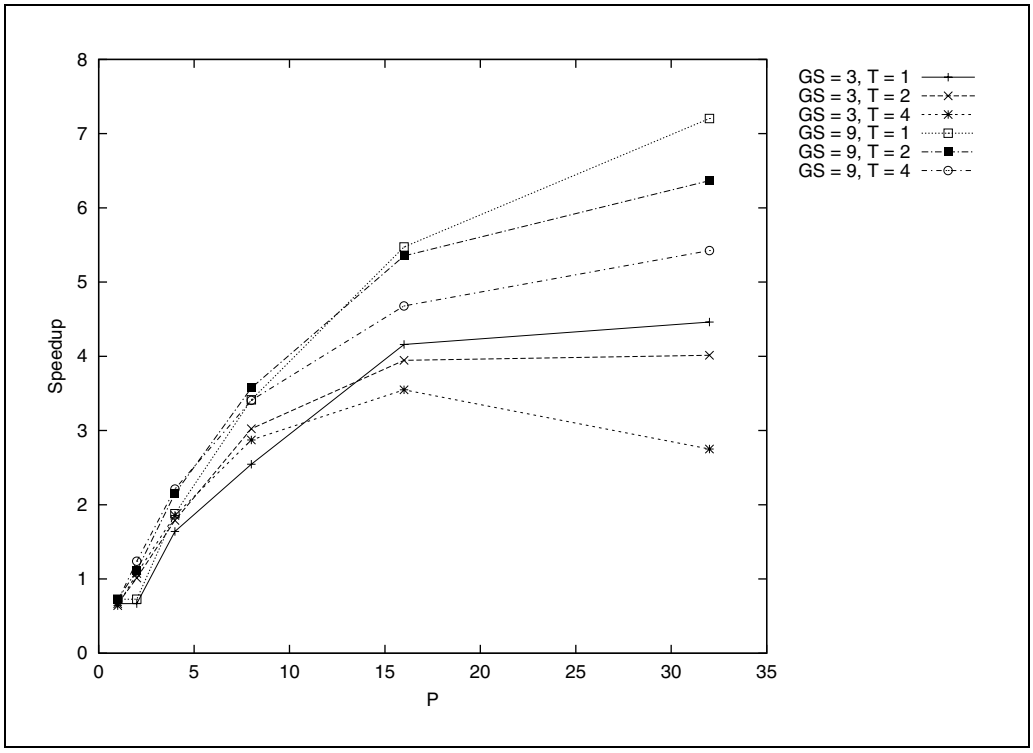


Figure 65: jnfib(21), future, light RTS

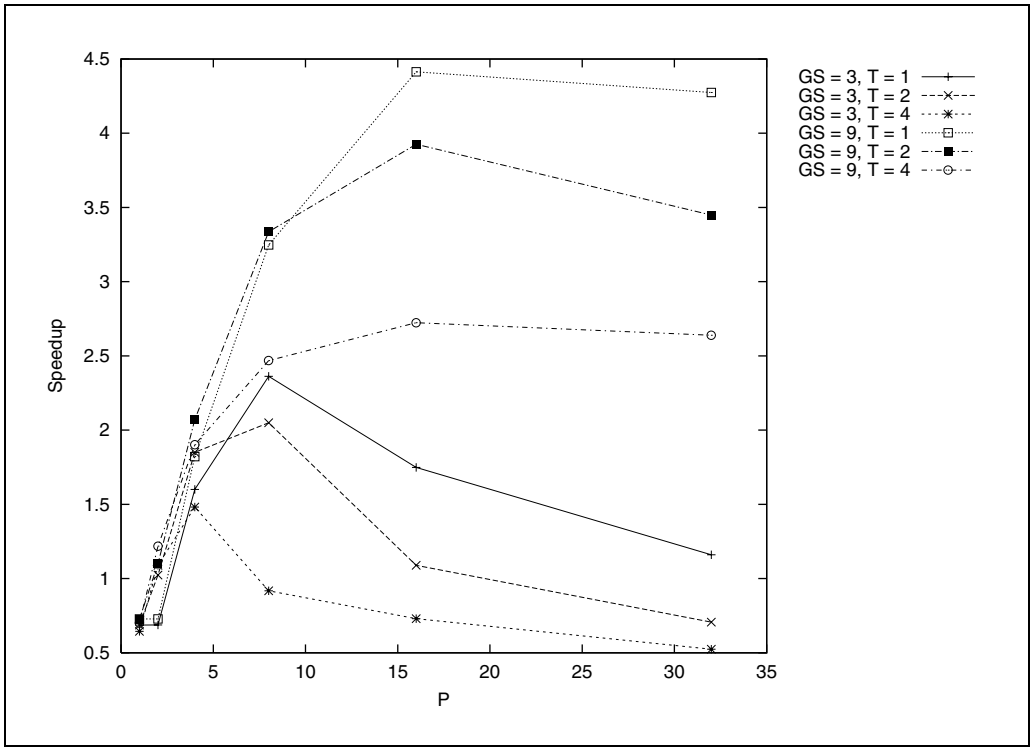


Figure 66: jnfib(21), future, medium RTS

The only memory operations in the program, excluding the small amount of window spilling, are to allocate the LoopWorker objects whilst forking threads; since there is no garbage collection, every object allocated results in cache misses. Increasing the granularity decreases the frequency of these allocations, which is supported by the GS=9 result, where the memory bandwidth

is significantly lower at 308 MB/s, even though the program executes for 40% less time (i.e. a 70% reduction in the memory traffic). However, the processors are still 55% idle. In fact, the utilisation figures for GS=3 and GS=9 are close for all of the current results (Figure 67), and do not account for the performance difference: the decreased execution time must also come from executing fewer instructions and/or less waiting for memory.

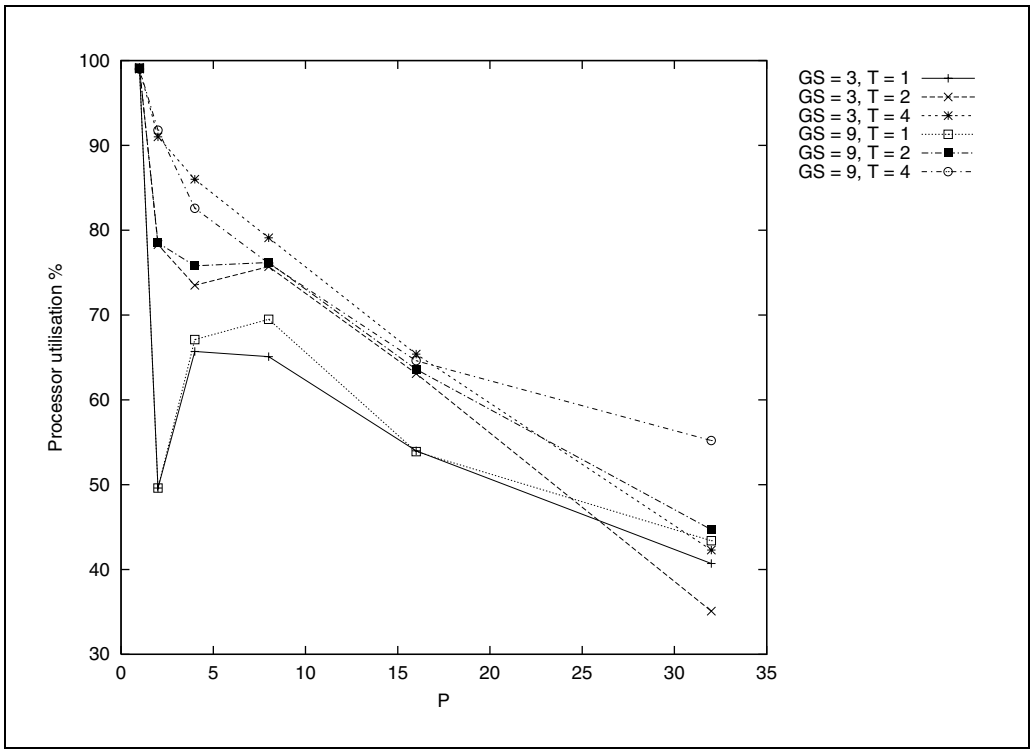


Figure 67: jnfib(21) processor utilisation, current, light RTS

To investigate this, the range nfib(21) to nfib(30) is run on the 'current' simulator, with 32 processors and 2 contexts/processor (Table 26). The speedup results are plotted in Figure 68, with the processor utilisation and instruction count (relative to the serial version) in Figures 69 and 70.

The speedup figures can be accounted for very closely by the changes in instruction count and utilisation; that is, for all the results,

$$32 \times \frac{\text{utilisation}}{\text{rel.instructions}} \approx \text{speedup} \pm 2.5\%.$$

This might be expected from the definitions; however, exceptions and stalls count towards utilisation but not instructions, so this result is not completely trivial.

Figure 69 shows that the utilisation figures are close for smaller problems, but whilst the coarse-grained program more than doubles its utilisation as the

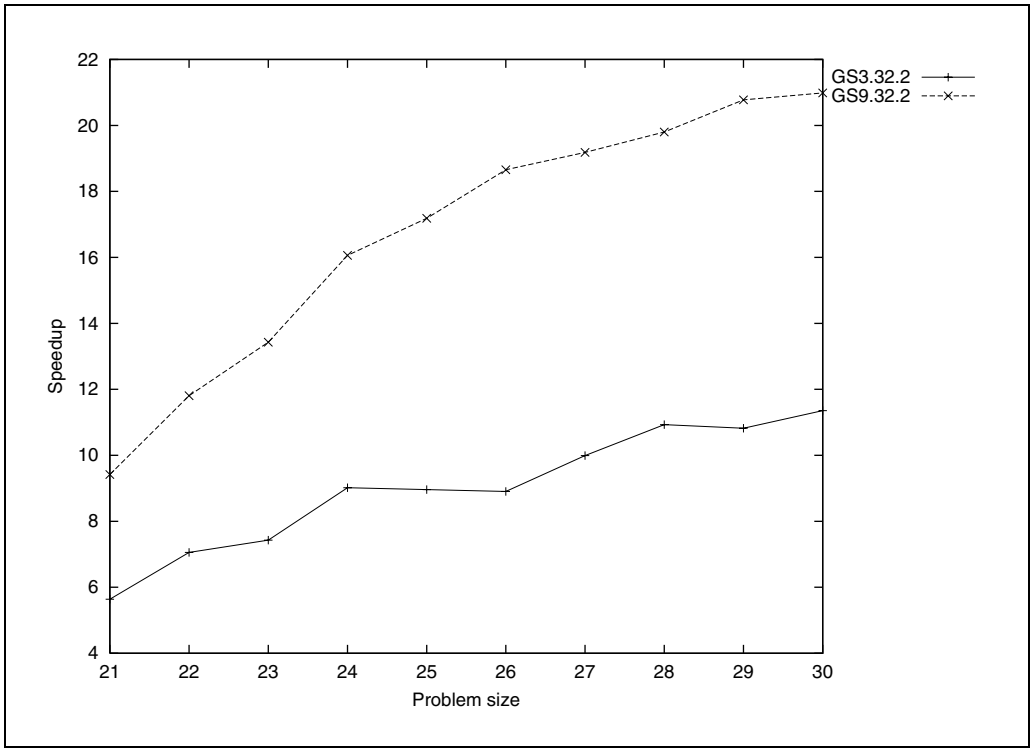


Figure 68: jnfib speedup, current, P=32, T=2

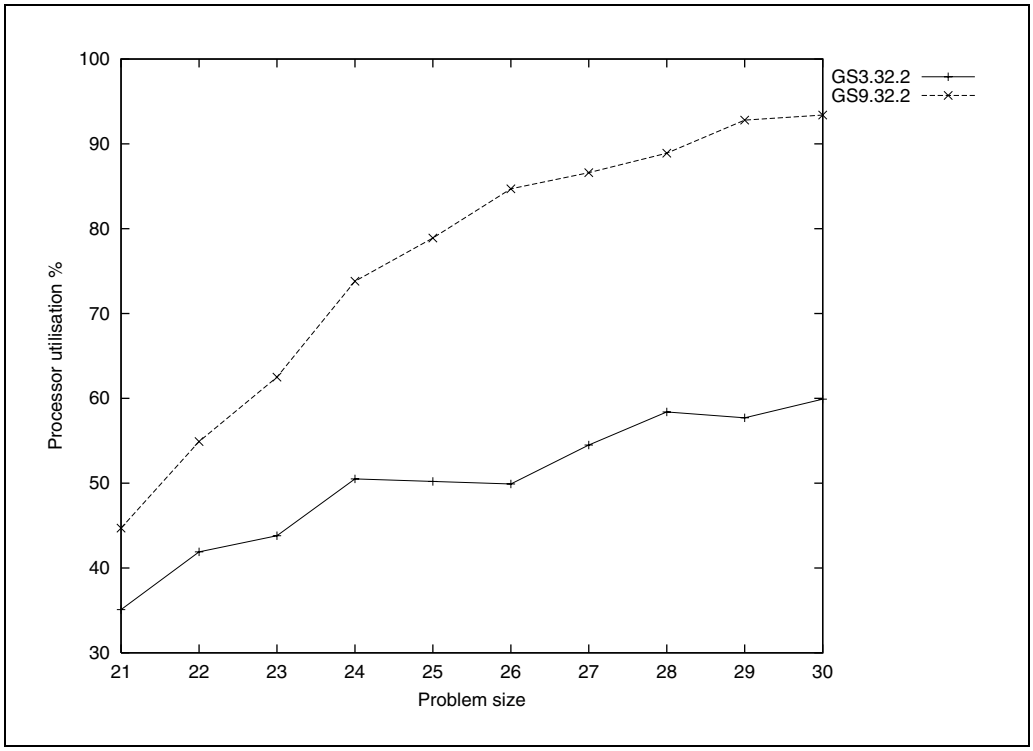


Figure 69: jnfib processor utilisation, current, P=32, T=2

problem size increases, levelling out at over 90%, the fine-grained one only approaches 60%. On the other hand, the coarse-grained program barely improves its instruction count (Figure 70), whilst the fine-grained one shows a 14% improvement.

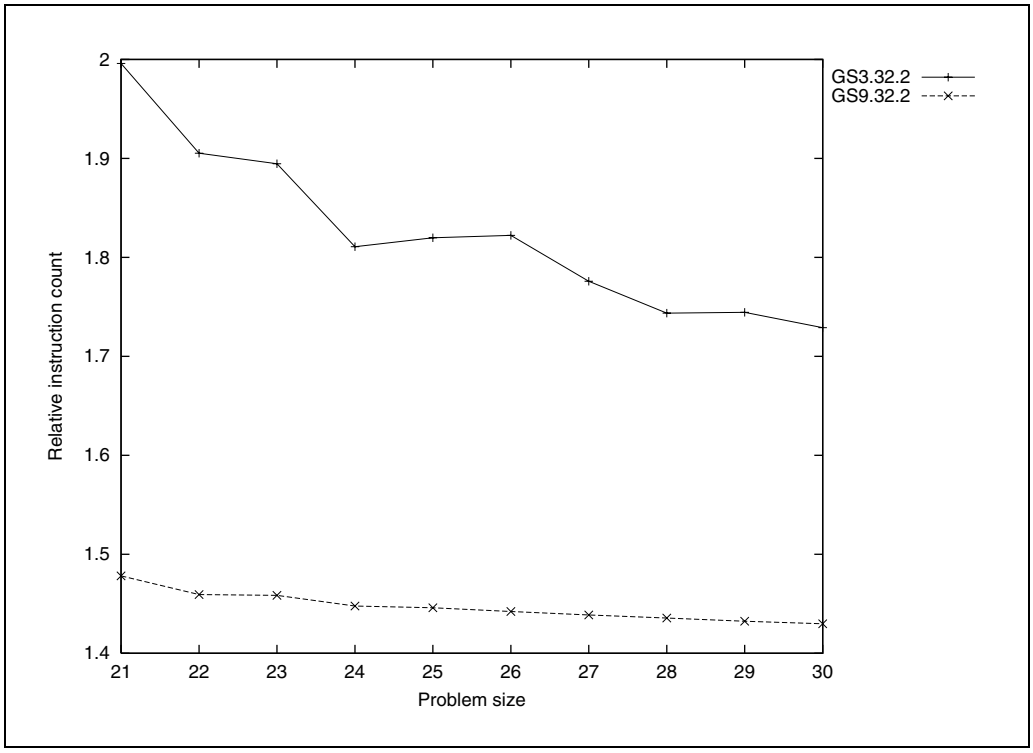


Figure 70: jnfib, relative instruction count, current, P=32, T=2

To interpret these results, a grain size parameter of 9 results in large enough parallel tasks that the costs of forking (token distribution) and joining (synchronisation), when a processor would otherwise be idle, are completely hidden by multithreading. The small change in relative instruction count reveals that the 'shape' of the computation does not change enough to affect the parallelisation overhead substantially: as the program size increases, more forks will succeed nearer the root of the recursive tree, but this appears to have little effect here. The limiting factor for nfib(21) appears to be the available parallelism, as 42% of `free_proc()` calls succeeded. For larger problems, utilisation approaches the ideal; any further performance improvements will have to come from reduced overhead, i.e. increasing the grain size still further.

In contrast, attempting to use too small a grain size means that the overheads, in particular the idle time whilst a token circulates and a fork takes place, are not hidden. The fall in relative instruction count as the workload increases means that the structure of the computation is changing; the effective grain size becomes coarser as the workload is divided at a higher level. This may also account for the utilisation improvement. Nfib(21), with only 13% of fork possibilities succeeding, was not lacking opportunities for creating threads.

There are two competing effects here, grain size and available parallelism, which must be evaluated against the cost of creating new threads. For the assembler nfib program on the current simulator, thread creation was fast enough that the fine-grained threads won. The Java RTS imposes a greater cost on thread creation, which means that the coarser-grain threads were better even though an insufficient number could be created. These jnfib results show that this cost can be hidden, so that given sufficient parallelism the system approaches full utilisation, and that this can be achieved with a minimum grain size of around 1000 instructions. The parallelism overhead, in this case, is a little over 40% (i.e. speedup approaches 70% of the ideal), and there is still plenty of spare memory bandwidth and bus capacity, so that the system should scale even further.

The hardware token distribution provides a big advantage over the 'medium' RTS, with its software queue, in this program. If the grain size were increased, one would expect this advantage to decrease; however, this also results in less available parallelism for the same sized problem.

10.2 Empty

The jnfib program above demonstrated that impressive speedups could be achieved using the Jamaica hardware mechanisms for fine-grained tasks, even given the overheads of the Java RTS. The Empty program, whose main loop is in Figure 71, is designed to explore further the trade-off between grain size and parallelism for different problem sizes.

```
/*par threads = 2 */
for (int j=0; j<L; j++) {
  for (int k=0; k<11*N;
        k++){
    /* nothing */
  }
}
```

Figure 71: Empty code

The program is parallelised automatically using the modified Javar, and the body of a single static method consists of the nested loop above. The loop bounds L and N are adjusted so that the total amount of work remains constant. Each iteration of the inner loop takes 9 cycles; the number of inner iterations is 11N, to give approximately 100N cycles as the minimum grain size when the outer loop is parallelised; L measures the potential parallelism. All

of the results here use the light RTS on the current simulator, with 32 processors and two contexts/processor.

L and N are varied in powers of 2, subject to $L \geq 2^6$, $2^7 \leq LN \leq 2^{19}$; the restriction on L is so that there is (potentially) at least one thread available per context. The results are plotted twice; Figure 72 shows the speedup against L, and Figure 73 shows it against N. In each case the solid lines are lines of constant LN (total work), and the dashed lines are of constant N or L respectively.

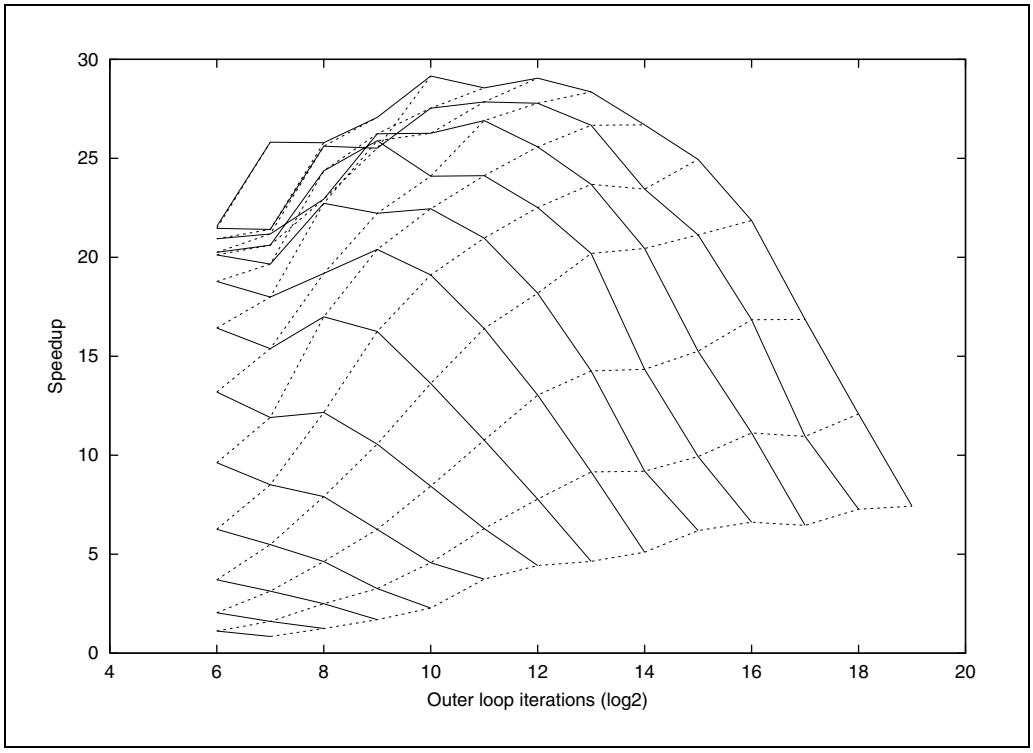


Figure 72: Empty, speedup vs. outer loop iterations, current

The speedups are relative to a single serial run *per value of LN*, thus some caution is needed when interpreting the graphs. Within each LN, a different division of work between the inner and outer loops would give a different serial execution time, so in theory every point on the graph could be a speedup relative to its own serial execution; however, this would give little information on which configuration was actually faster. Instead, each solid line is relative to the $N = \min(2^4, LN/2^6)$ serial execution, which means that each is internally consistent, i.e. for a given product (total workload), each line is proportional to execution rate. The difference in the serial executions is only noticeable for small values of N; for example, for $LN = 2^{19}$, compared with the $N = 2^4$ result, $N = 2^{13}$ is 1.6% faster, and $N = 2^0$ is 20% slower. This would improve the speedups at the right side of Figure 72 and the left of

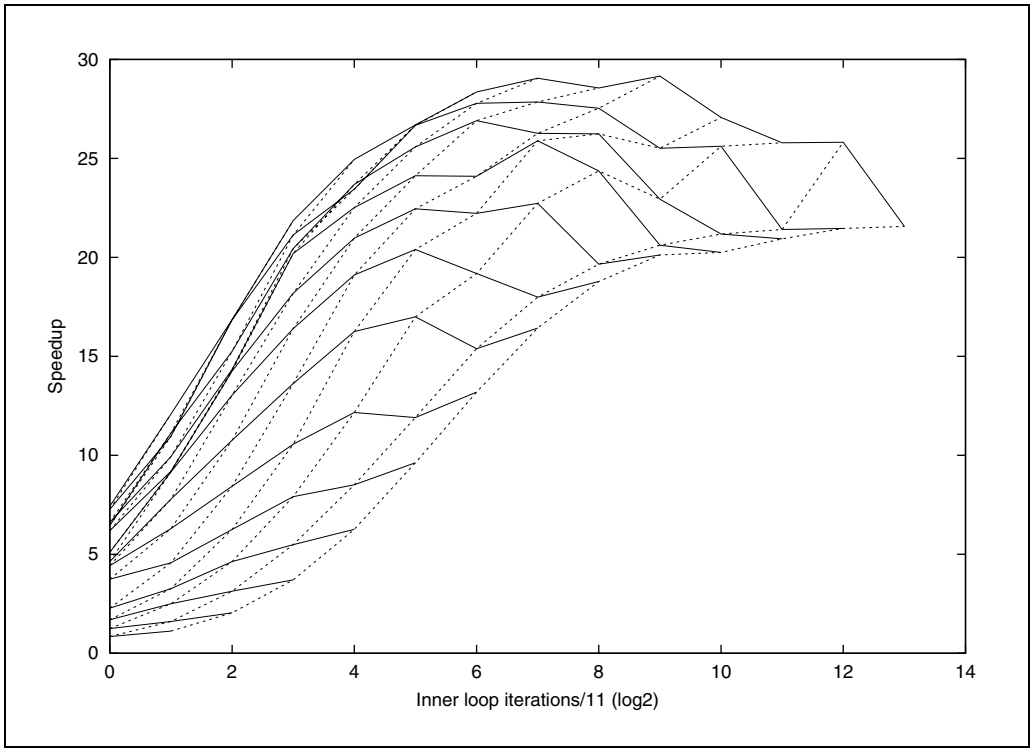


Figure 73: Empty, speedup vs. inner loop iterations, current

Figure 73. Also, comparisons between lines are valid to within a few percent provided that $LN \geq 2^{10}$, since both speedups are then relative to $N = 2^4$ serial results. Lines with $LN < 2^{10}$ have their speed effectively increased compared with the others, since their serial executions are slower.

To take Figure 72 first: a dashed line's gradient indicates sensitivity to L; sensitivity to N is given by vertical distance. It can be seen that, for very small N, an increase in L does not make a significant improvement: performance is generally poor. In this region, a program is limited by overhead more than by lack of parallelism. Performance is much improved for $N > 2^2$; once N reaches 2^7, i.e. 1,408 iterations of the inner loop, further increases in N make little difference and L is the biggest determiner. Here, parallelism matters more, and performance is determined by the load balance.

As a vivid illustration of these effects, Figure 74 shows quartiles of the number of instructions executed by the processors for $LN = 2^{19}$ (i.e. three quarters of the processors in the system executed fewer instructions than the '75%' line, etc). At the left side, where the parallelised loop is small, the load balance is poor; at least one processor is executing many more instructions than the average. As L increases, the lines draw closer together, indicating a much better load balance. For large L, with $N \leq 2^3$, all of the lines begin a steep

upward curve, as the cost of manipulating the threads becomes large. As remarked above, even the serial program would exhibit an upward trend for very small N, due to the differing loop overheads, but this is at most 20%. The low '0%' (minimum) line is due to the scheduling thread which occupies a single context on one processor without doing any work.

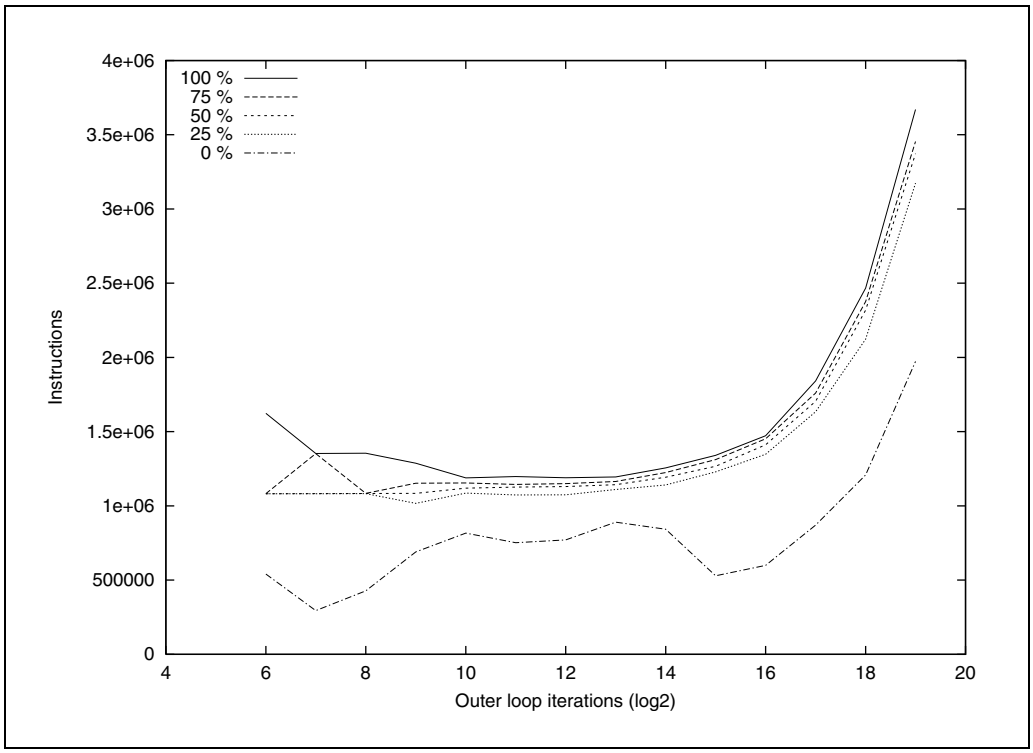


Figure 74: Empty, load balance, LN=2^{19}, current

So, given a particular amount of work (i.e. product LN), how should it be split between L and N? For $LN \leq 2^{12}$, the answer is always to decrease L (as each solid line has negative gradient), so that grain size increases. Indeed, it may be better to decrease L even below the number of contexts, on the grounds that they cannot all be used efficiently, but this region is not represented in the graphs. For $LN > 2^{12}$ there is a compromise. Peak speedup appears between $N = 2^5$ and 2^9 (Figure 73); a good rule of thumb would be to set $N = 2^7$, although any point with $N \geq 2^5$ and $L \geq 2^8$ is 'close' to optimal. These correspond to a minimum grain size of 352 iterations (i.e. about 3,168 cycles), and four times excess parallelism (potential threads/available contexts).

Figure 75 shows how good a predictor L is of the number of threads created; the best fit line ($3.53 \times L^{0.633}$) is also drawn, and all of the points except the last are within $\pm 36\%$ of this line (i.e. doubling the size of the problem increases the number of threads by a factor of 1.55). The slight convex appearance of the line

on the graph suggests that the real number actually grows faster than a power law.

N is not such a good predictor of the thread creation rate, but still has a big influence (Figure 76).

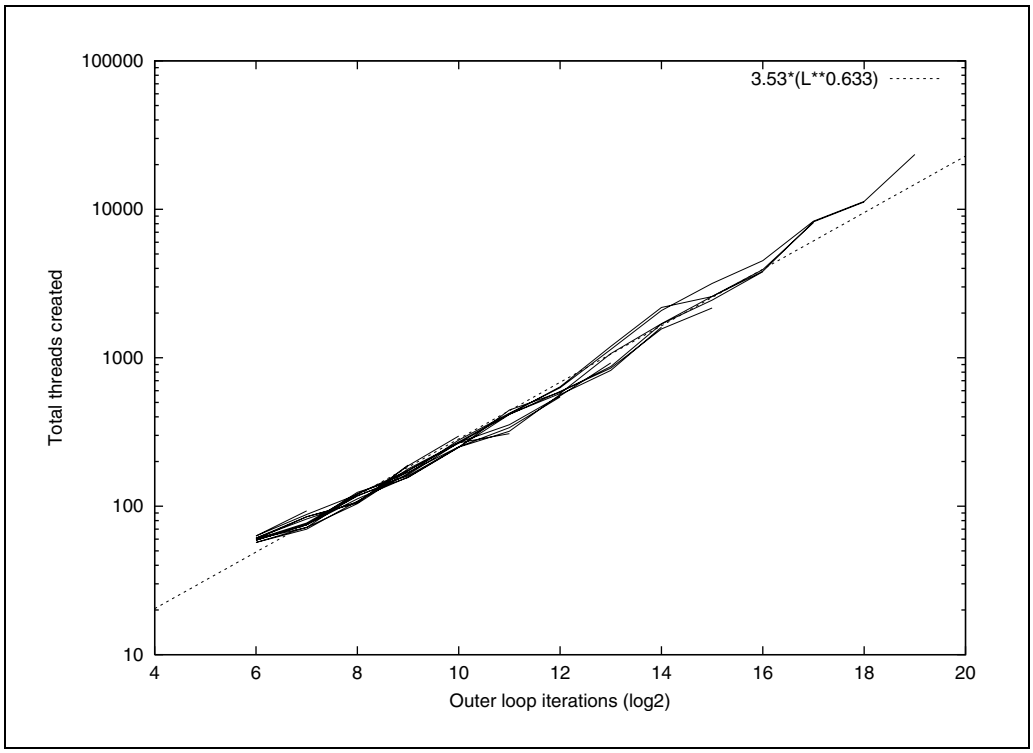


Figure 75: Empty, number of threads created, current

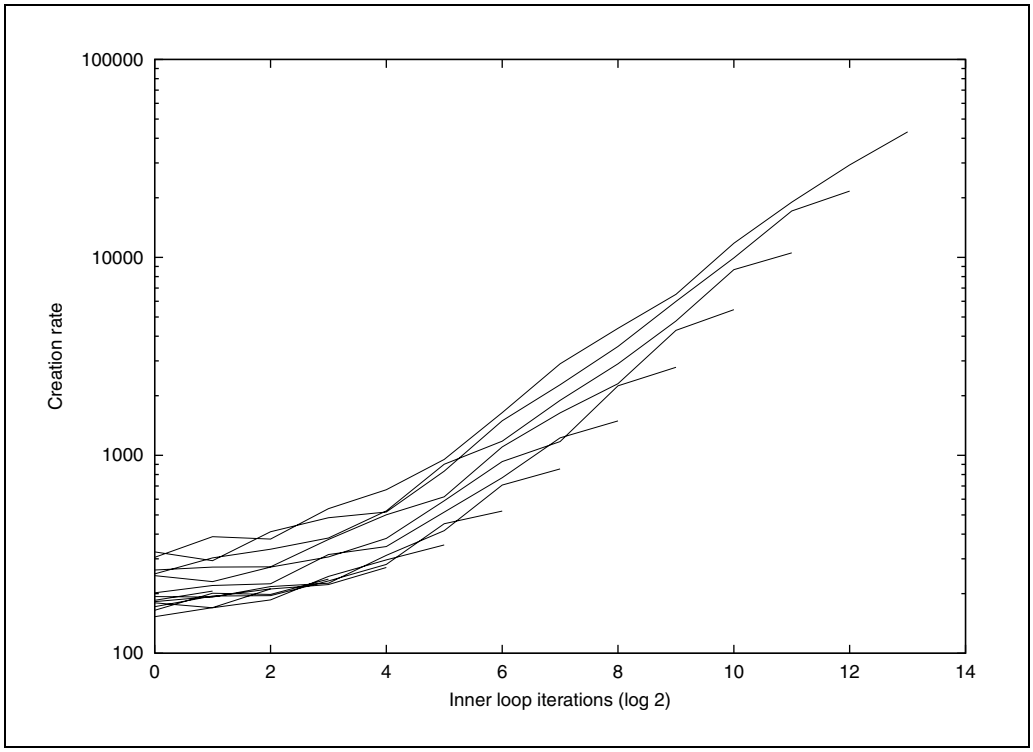


Figure 76: Empty, thread creation rate, current

The future parameters produce almost identical effects, with similar grain sizes and parallelism requirements for good results even if the speedups achieved are slightly lower. The number of threads produced has a best fit line of $2.72 \times L^{0.721}$, i.e. a factor of 1.65 for each doubling of problem size.

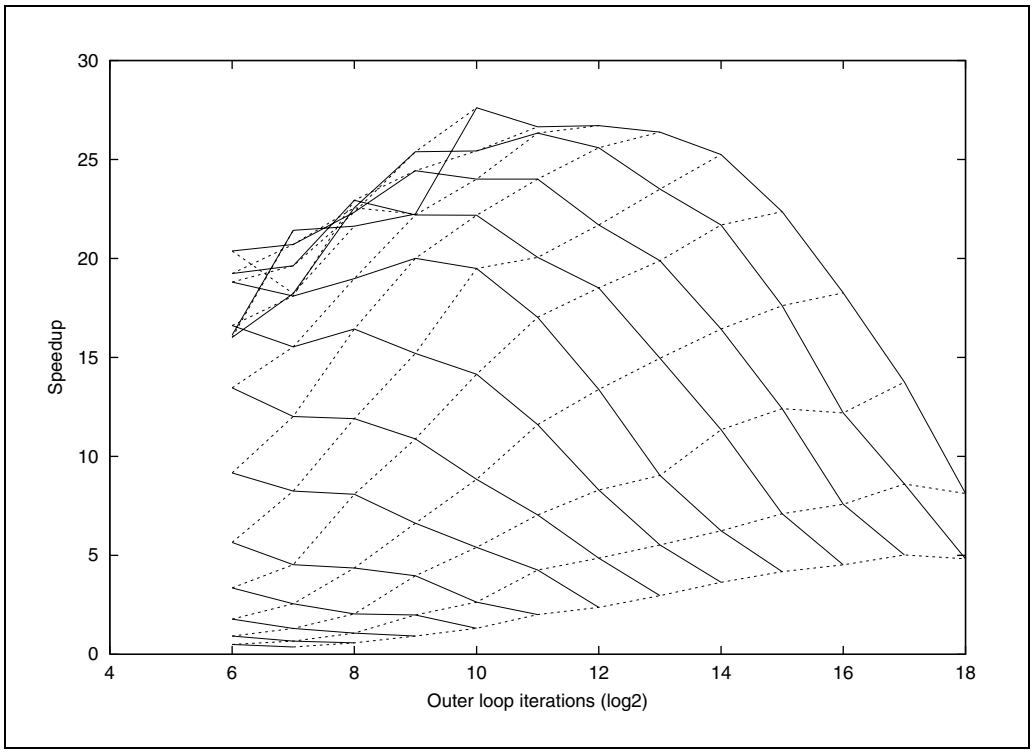


Figure 77: Empty, speedup, future

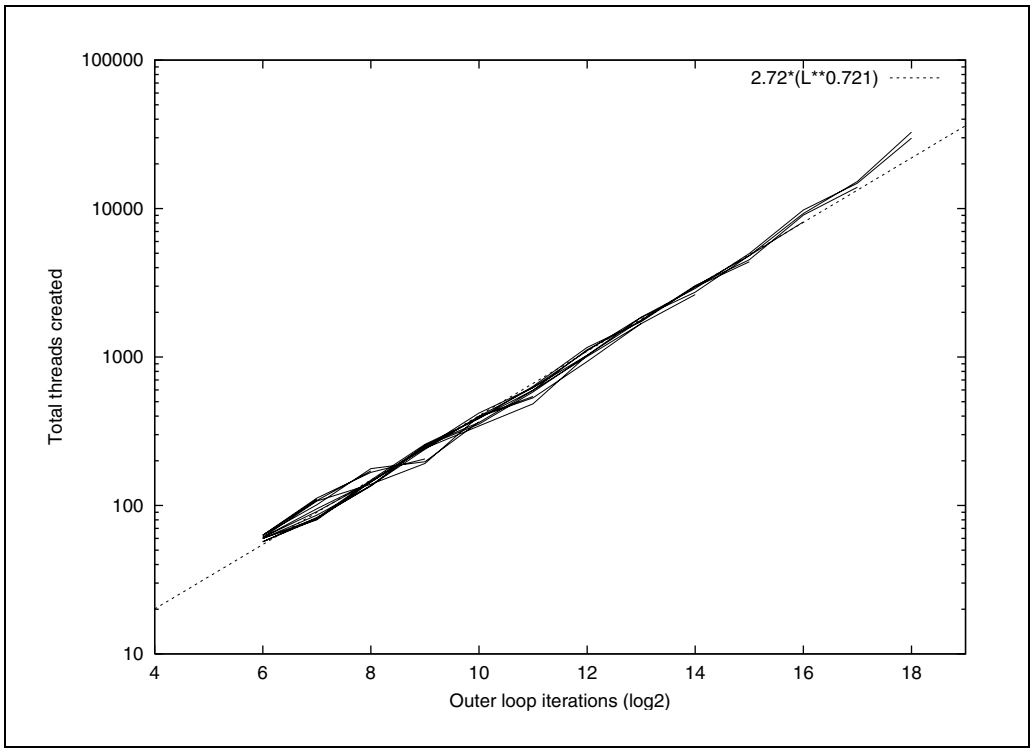


Figure 78: Empty, number of threads created, future

10.3 MPEG encoding and decoding

Ahmed El-Mahdy has translated the MPEG Simulation Group's MPEG2 encoder and decoder [MPE96] from the C originals into Java. The performance of the Jamaica system can thus be assessed on two real multimedia kernels. The descriptions of the programs are adapted from [WEMW00].

10.3.1 jmpeg2encode

The MPEG2 encoder is used to encode two 360x240-pixel video frames, 1/4 the resolution of MPEG main profile, main level. The motion estimation kernel, which accounts for about 65% of the runtime, is parallelised using both the original and modified Javar; the parallel section loops across 16x16 pixel macroblocks, giving a maximum parallelism of 345 coarse-grained units.

The serial motion estimation, with no parallelisation performed, takes 954,047,954 cycles (current) or 960,388,744 cycles (future); the small difference of under 1% in cycle count reveals that the program has very good cache behaviour, and hence almost linear speedup with clock rate. Speedups are expressed relative to these figures, and the 1% serial difference means that current and future speedups can be compared directly as an indication of absolute cycle count. The source-code granularity in the parallelised versions will be around 3M cycles, several times larger than the largest considered for the Empty program (which was $100 \cdot 2^{13} \approx 0.8\text{M}$).

The light and medium RTSs, with the modified-Javar parallelised code, are shown in Figure 79 running on a single processor. This expresses the parallelisation overhead, and is under 0.02%, as would be expected with such coarse granularity.

The multiprocessor versions are run with 32 processors and two contexts/processor, as for the Empty program. The results are shown in Figure 80. The amount of parallelism, at 345, corresponds to $L = 2^{8.4}$ for Empty; from Figure 72, the expected speedup is 20 to 25, which agrees with the measured values.

For both the current and future parameters, the heavy RTS (i.e. original Javar with Java threads) produces very good speedups, of 29.5 and 28.9 respec-

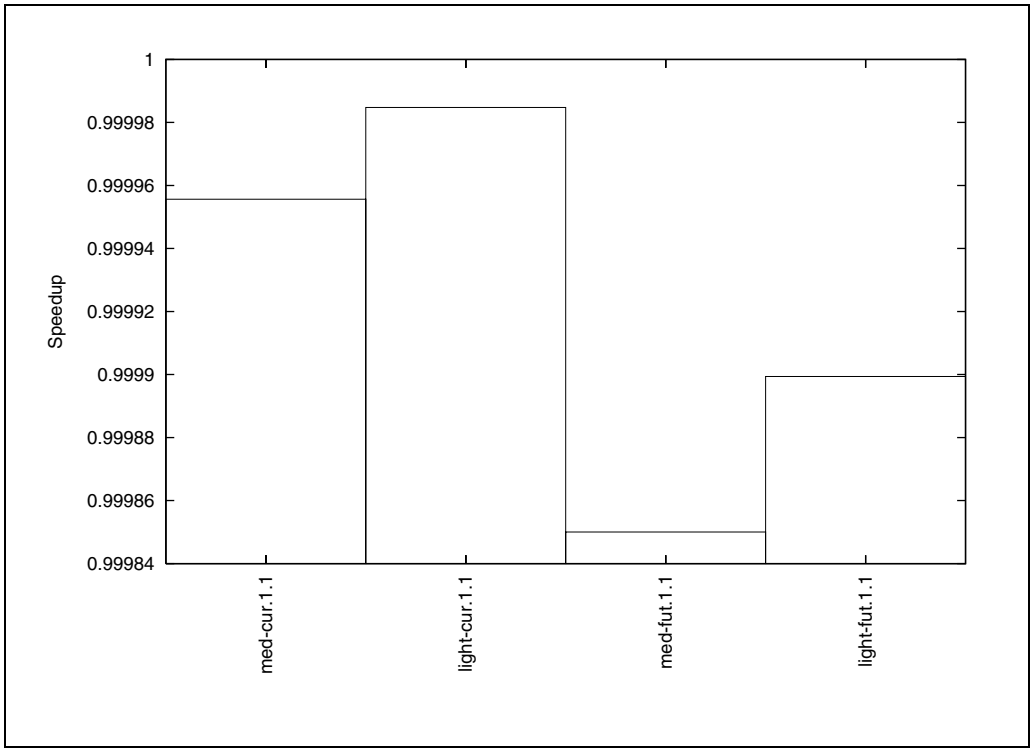


Figure 79: jmpeg2encode, uniprocessor speedup

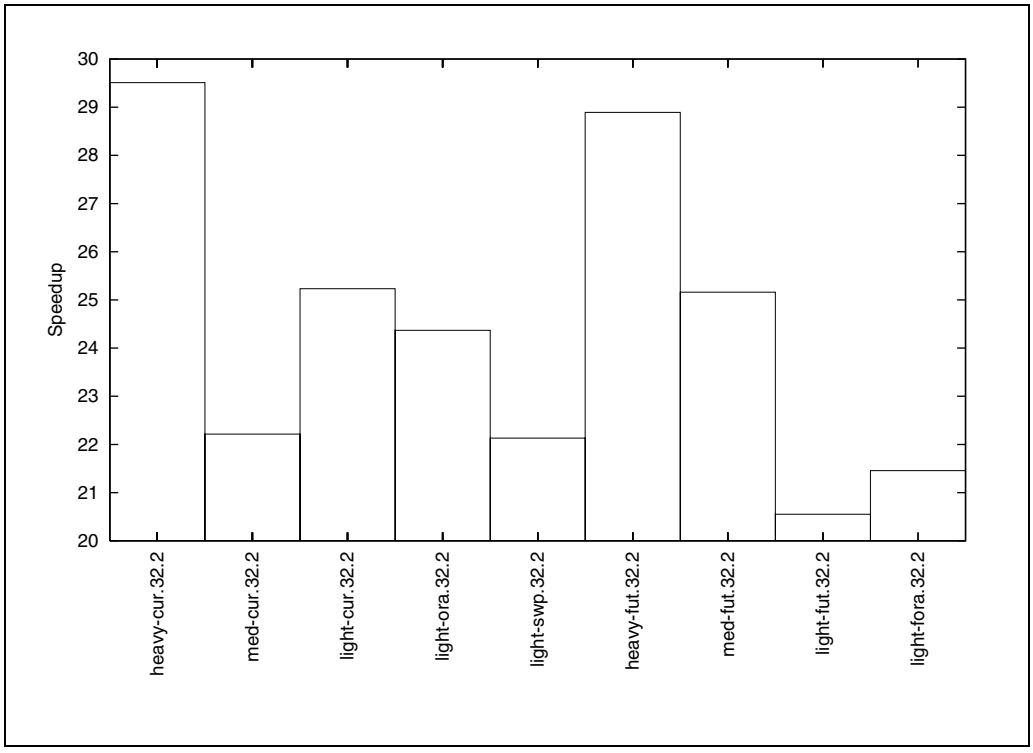


Figure 80: jmpeg2encode, multiprocessor speedup

tively. Evidently, the guided self-scheduling is producing an extremely good load balance, whilst the management costs are small compared with the 3M cycle granularity. With the current parameters, the next best results are light, light with the token finding oracle ('ora'), and then medium; light has a 14%

performance advantage over medium. The heap-allocated windows also give a 14% improvement over static partitioning ('swp'), using the light RTS.

For the future parameters, the order of the dynamically-parallelised results is reversed: the medium RTS is 22% better than the light, with the token oracle ('fora') between them.

Given the granularity, these light/medium results must be down to load imbalance rather than differences in thread overhead; the partitioning of the computation among the processors is sensitive to small changes in timing, as the tokens circulate around. Even the medium system's queue will give different results if two fork-tests happen in the opposite order. The difference between the 'med-cur' speedup of 22 and 'light-cur' with 25 is $38.2\text{M} - 43.4\text{M} = 5.2\text{M}$ cycles, less than two macroblocks. An extremely good speedup of 30 would be 31.8M cycles, another 6.4M different, also around two blocks. Given this sensitivity, it is hard to argue that either RTS has an inherent advantage.

This is illustrated in Figure 81, which shows the 0%, 25%, 75% and 100% quartiles of instruction count[1], for the encoding operation, i.e. half of the processors lie within each box, with one quarter in each tail[2]. It can be seen that the number of instruction executed by most processors for each program is in the same band, around 25 to 30M. For the current results, the peak instruction count tracks the speedup figures in reverse. The odd point is the light RTS with the future oracle: its most heavily loaded processor executes more instructions than without the oracle, yet it runs faster. In fact, the highest count is alone at 39.5M, with the next processor down at 31.4M, but this has not affected the execution time so much.

The conclusion here is that the recursive-subdivision programs have broadly similar performance for any RTS, and that the software scheduling of original Javar is a better method for such large granularities.

1. In fact the first processor also executes the rest of the program, so the largest instruction count is ignored, and the '100%' figure is for the next largest.
2. Recall that the scheduling thread occupies a context.

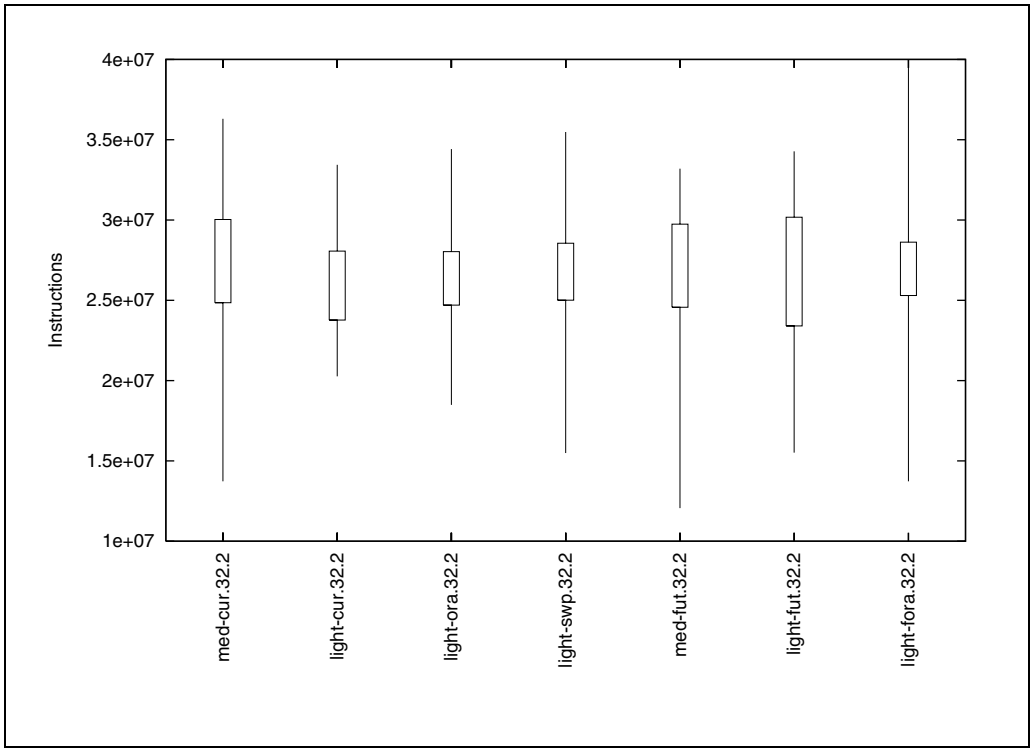


Figure 81: jmpeg2encode load balance

10.3.2 jmpeg2decode

The MPEG2 decoder is used to decode one video frame of the same size as above. The horizontal and vertical interpolation filters were parallelised, and are estimated at about 20% of the runtime. This loop has much finer granularity than the encoder's: the serial version takes only 23,735,429 cycles (current), or 24,537,135 cycles (future), and again the cache behaviour is good, with a 3.4% increase. The maximum parallelism is 420 (that is, 180 horizontal and 240 vertical), corresponding to $L = 2^{8.7}$ for Empty, and this gives a granularity of about 57k cycles (or $N \approx 2^{9.2}$).

The uniprocessor results are in Figure 82, and give a parallelisation overhead of about 26% for all of the light RTSs. This is surprisingly large: based on the jmpeg2encode results, a figure of about 1% would be expected. Examining the instruction counts in more detail, the serial and parallelised programs execute almost the same numbers of all instructions apart from memory operations: the current-light program performs 15% more loads and 16% more stores than does current-serial. The explanation must be that the recursive function calls are 'interfering' with register allocation in jtrans, so that some variables are being placed on the stack rather than in registers. If jtrans performed better optimisation, nearly all of this overhead should be removable.

The slowdown in the jnfib program, which ran at about two-thirds its serial speed, is not due to this effect.

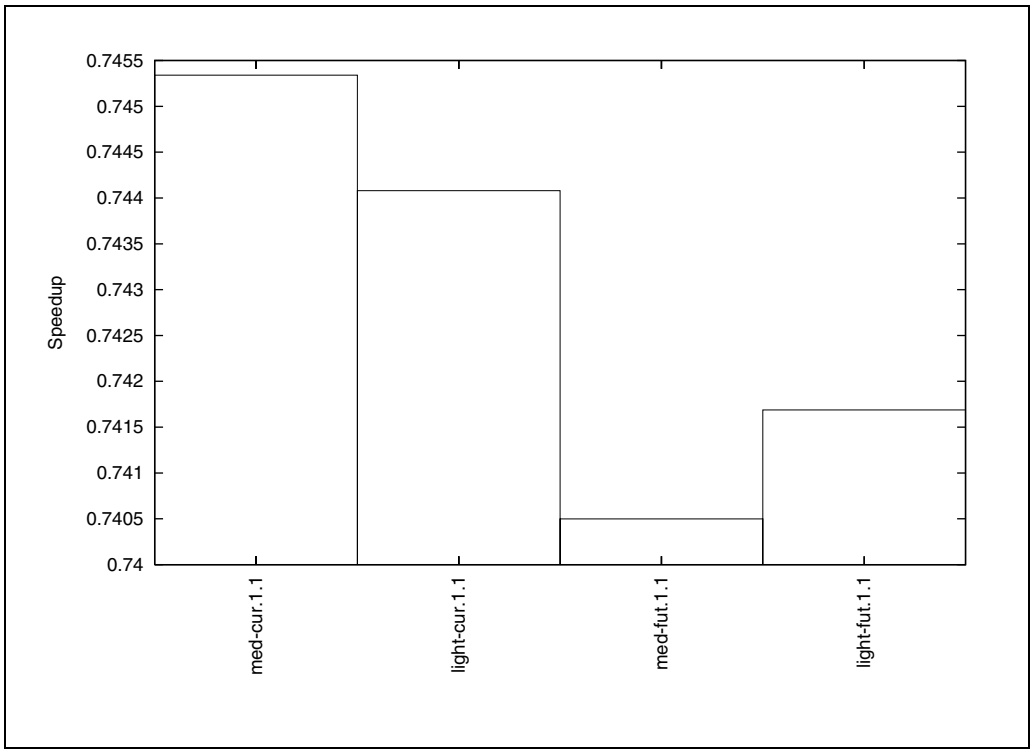


Figure 82: jmpeg2decode, uniprocessor speedup

The multiprocessor results are in Figure 83, again for 32 processors and two contexts/processor. The 'heavy' system gives poor results on the fine-grained workload, with speedups of only 7.2 and 4.1 for current and future parameters; the thread creation time is simply too great here. The other RTSs produce results similar to each other, a speedup of around 16 or 11. From Figure 72 and Figure 77, one would have expected about 25 or 22 for Empty; much of this difference is accounted for by the 26% uniprocessor overhead.

The light system is 3.7% better than medium (C), rising to 12% (F). Again, these could be explained in terms of load balance, but the graph (Figure 84) shows how all of the light systems produce similar instruction counts, with medium being noticeably worse particularly in the future. This may represent a genuine improvement from the light RTS.

As a measure of the granularity actually exploited, the current, light RTS program created 351 threads with a creation rate of one every 4,083 cycles; the future one made 378, one every 5,718 cycles.

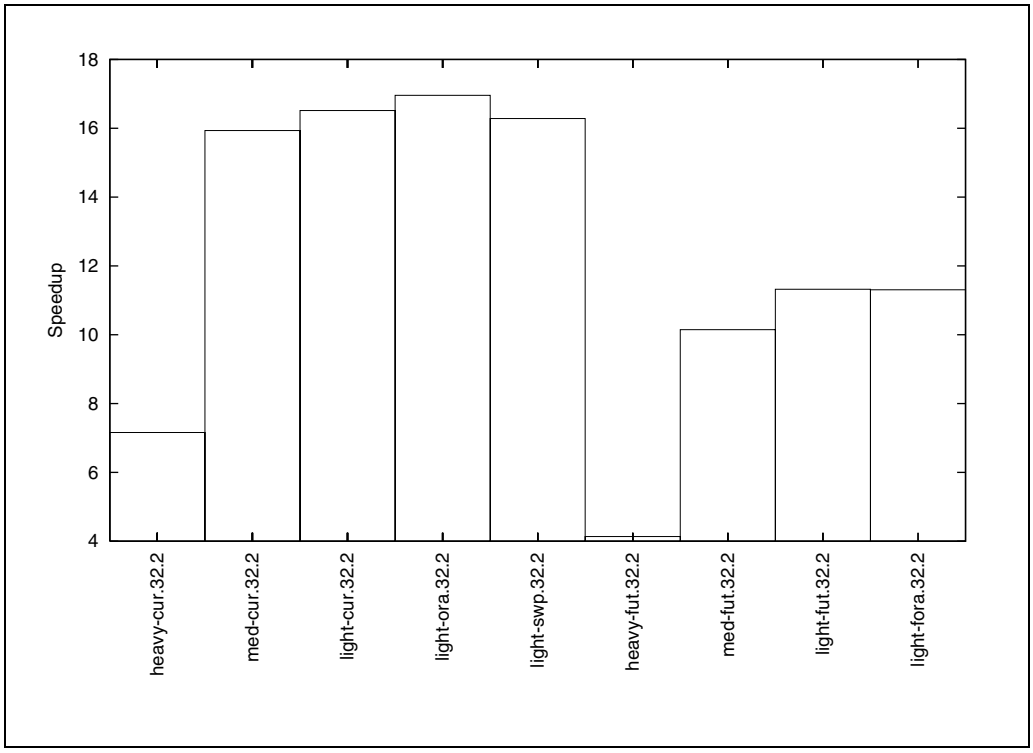


Figure 83: jmpeg2decode, multiprocessor speedup

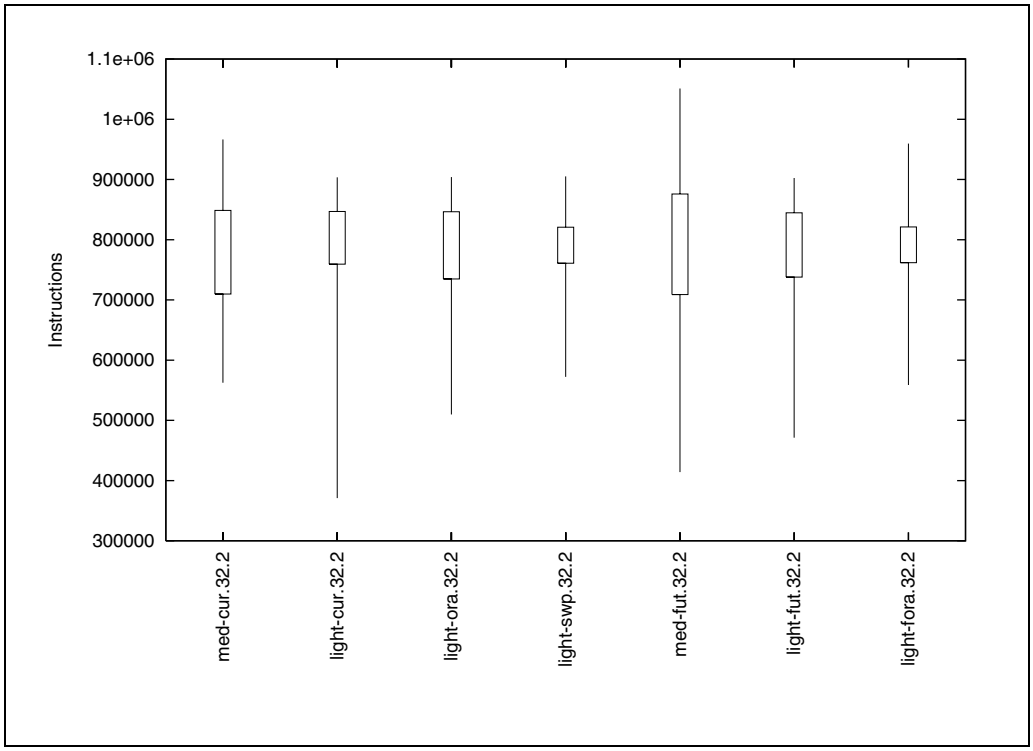


Figure 84: jmpeg2decode load balance

CHAPTER

11. Summary & conclusions

11.1 Summary

This work has introduced the Jamaica architecture, a single-chip multiprocessor with several interesting features: multithreading with hardware support for locating idle contexts and distributing threads to them, heap-allocated register windows for flexible sharing of a register file among the threads, and a cache coherence protocol to link the processors on a shared bus.

The first chapter motivated this design by outlining the limitations of current superscalar designs, in terms of scalability and hiding memory latency. The contrasting requirements of Java, an object-oriented language usually compiled just-in-time, were also introduced: simple processors with short pipelines make a simpler target for JIT code generation and reduce the cost of frequent method calls.

Chapter 2 reviewed a wide range of architectures, from VLIW to simultaneous multithreading, and also included dataflow and graph reduction models. The conclusion was that single-chip multiprocessing is a good way of building scalable systems, and the partitioned construction allows easier design and more localised wiring than a single complex CPU. Multithreading introduces elements of dataflow, and load balancing for fine-grained threads has been borrowed from graph reduction machines.

Registers were covered in Chapter 3, from zero-address (stack-based) architectures up to windowed register files. Stacks may be attractive as a target from

Java bytecode, but there is too much overhead in their manipulation. Register windows were adopted because they allow low-cost function calls and easy register allocation.

Finally, Chapter 4 looked at memory and cache coherence protocols. High performance memory, like the Rambus system supposed here, is pipelined and banked; it can supply good bandwidth, but needs large numbers of outstanding transactions to hide the latency. This is another point in favour of multithreading/multiprocessing. Most existing cache coherence protocols were not designed with single-chip multiprocessors in mind, and some shortcomings were noted. In particular, main memory should not be involved if it can be avoided.

11.1.1 The Jamaica architecture

Jamaica's instruction set is conventional, and based on that of the Alpha. The major difference is heap-allocated register windows (§5.1), which allow more flexibility in sharing a single register file between multiple threads than a static partition.

Several new instructions also support distribution of threads to idle contexts. All of the processors are connected by a ring, around which tokens travel – each token identifies an idle context. A thread may test rapidly for the availability of a token, and acquiring the token grants permission to fork a new thread onto the other processor. The fork is done over the shared bus, and is designed to look as much like a procedure call as possible. When a thread terminates, its token is automatically released back onto the ring. This allows dynamic parallelism, where decisions on where to create threads are taken at run time.

To allow efficient spinlocks in a multithreaded processor, an addition to the load-locked/store-conditional mechanism was proposed. The `WAIT` instruction uses basically the same mechanism, but suspends a thread until a particular cache line has been written.

11.1.2 The simulated system

A simulator has been developed which models one supposed implementation of the Jamaica architecture (Chapter 6). The CPUs have simple five-stage pipelines, although this is not a requirement of the architecture; future implementations could be based on multiple superscalar or SMT processors.

A detailed description of the cache coherence protocol was also given (Chapter 7). Each processor has private L1 instruction and data caches, connected over a shared bus to a single L2 cache. Second-level cache misses are serviced by (possibly several) Rambus DRAM channels; the RDRAM device timings are modelled in detail.

Jamaica's cache coherence protocol is pipelined, so that although a transaction takes eight bus cycles a new one may begin every two cycles. It also allows cache to cache transfers and negotiation of ownership, so that dirty lines may be evicted from one cache and given to another without updating memory.

The Java runtime system was also described, although only the thread-handling code was written as part of this work. Three different parallelisation schemes are used: the Javar source-level restructuring compiler creates explicit Java threads in the code, and they schedule work themselves from a pool of waiting tasks. Two dynamic systems were developed which use a modified Javar: loops are parallelised using recursive subdivision, with a decision on forking taken at each level of the recursion. The 'light' RTS uses Jamaica's token mechanism, whereas the 'medium' one models a more traditional queue of idle threads.

Java bytecode is translated to Jamaica assembler code by the jtrans compiler, which performs the kind of compilation a JIT compiler might do, and is then linked with the appropriate RTS libraries.

11.1.3 Experiments

Two separate sets of benchmarks were used to test out the system. The first set, written in assembler, used short programs to test out one particular aspect. The second set was written in Java, and loops were automatically parallelised. For each program two simulator configurations were tested, representing a

machine constructable now ('current') and a prediction for ten years time ('future').

The memory system was considered first, using Stream-like vector arithmetic. The 'current' results showed that eight threads are enough to saturate the memory system however they are distributed, with multiprocessing producing slightly better results than multithreading. The single RDRAM channel saturated at just over 60% of peak, resulting in a bandwidth of about 950 MB/s; the remainder of the time is taken up by bank conflicts and bubbles in the channel. This result is close to that predicted by a simple theoretical calculation. L1 cache interference proved to be a problem when many threads were sharing a processor, although this is partly due to unhelpful data layout within the programs, where the arrays' addresses are separated by a large power of two.

The 'future' results are generally more predictable, and approach memory saturation (also just over 60% of peak) for 32 total threads. At this point, bus utilisation is getting high (over 60%), and a higher memory bandwidth would need a faster on-chip bus or some other interconnection method.

The Bounce program tested contention on a lock, which could be a critical feature when fine-grained parallel threads are used. An acquire-modify-release operation took around 100 cycles, and is fundamentally limited by the shared bus transaction time. There was a very big advantage given by the `WAIT` instruction; some executions were up to 83 times faster, although some were only a 150 – 200% improvement. The spinlocks offer no fairness at all, in particular for the future parameters; it was common for a thread to release the lock and then immediately reacquire it itself. Bus utilisation is acceptable, at less than 36%, leaving plenty of spare capacity.

Finally, the nfib program tested the thread distribution mechanisms on a recursive program. For the 'current' machine, fine-grained (~40 instructions minimum size) threads did better than coarse-grained (~1000 instructions) ones. For the 'future' machine, with its relatively slower on-chip communication, coarse-grained was better: the fine-grained threads had saturated the bus with 32 processors. The speedups were around 16 on 32 processors, which is good given the size of the tasks.

The token distribution ring, as a means of finding idle contexts, performed well when compared with an imaginary 'oracle' having instantaneous global information. With at least four threads per processor the results are almost identical, showing that a more accurate method is not needed.

Heap allocation of register windows gives similar results to static partitioning when two contexts per processor were used, but was up to 23% faster for four contexts. Taking into account compulsory cache misses, this advantage is decreased to 10%.

The first Java program was a recoding of nfib: jtrans performed well, as the serial Java version was only 10% slower than the hand-coded assembler. Unfortunately, parallel speedups were not so impressive, reaching only 8–10, and coarser granularity was better. The results were much improved by increasing the problem size, and achieved almost full processor utilisation on the coarse-grained program; however, the relative instruction count, representing the parallelisation overhead, levels out at just over 1.4, which gives a maximum speedup of about 22 on 32 processors. The token-passing was also successful, as the light RTS was up to 70% faster than the medium RTS's software queue.

To investigate the effects of granularity and problem size further, the Empty program was run on 32 processors with two contexts/processor for a range of inner and outer loop counts. Some very good results were achieved, almost a 30-times speedup in some cases. More than 15-times speedup can be had with a minimum granularity of 400 cycles, provided that sufficient tasks are available. Four times excess parallelism (i.e. 256 possible tasks here) and a 3,000 cycle granularity were close to optimal. The 'future' machine, although giving slightly worse speedups, has very similar behaviour.

Some rules of thumb were established to guide the trade-off between granularity and the number of tasks: for small workloads, less than 400k cycles in total, the threads should be kept large (only one potential thread per context) to reduce the parallelism overhead. When the granularity can reach 3k – 10k cycles, start increasing the parallelism.

Finally, the system was tested on kernels from two real applications: an MPEG2 encoder and decoder, working on 1/4-size video frames. The encoder's motion estimation loop has very large granularity, about 3M cycles, and

the dynamic parallelism systems could not compete with the good load balance of Javar's software scheduling.

The decoder, with a granularity nearer 57k cycles, gave relatively better results: the dynamic systems were over twice as fast as Javar, and the token-passing 'light' RTS had a small advantage over the 'medium' software queue. Neither of the MPEG programs has much parallelism compared to the size of the machine, only around 400 tasks; the best results on Empty were achieved with around five times this many, and a full-size video frame should give better results.

11.2 Conclusions

This work has shown that fine-grained threads can be used efficiently on a single-chip multiprocessor. Jamaica's simple hardware support for dynamic parallelism, in the form of passing tokens around a ring, offered a clear advantage over software methods. This technique could be adapted to a CMP built of superscalar, VLIW or SMT processors. Performance on very coarse-grained threads is not so good, so it should be viewed as complementary to software scheduling rather than replacing it completely.

Loops in Java programs can be semi-automatically transformed into a dynamically-parallel form, to take advantage of these hardware mechanisms. A runtime system which integrates the dynamic threads into Java source code was also presented. Due to overheads in the RTS, the minimum granularity which can be used well[1] is around 500 cycles, and then only if many tasks are available; assembler programs give reasonable speedups down to below 100 cycles.

Multithreading gives more mixed results. A high-bandwidth pipelined memory system can indeed be well utilised when required, and for memory-intensive programs multithreading and multiprocessing are almost interchangeable. About eight threads per memory channel are needed. However, programs with very good cache behaviour suffered under multithreading. Caches with a greater associativity than normal are recommended, to reduce

1. Giving more than 16 times speedup on 32 processors.

interference among threads. When the token-passing mechanism was used, some multithreading was needed to give good results on fine-grained programs. It may be possible to introduce a dependence into the token allocation, so that well-utilised processors do not have more threads given to them.

A single shared bus is a reasonable interconnection method for a CMP of up to 32 processors; only one of the programs here was limited by bus capacity, and that was because of a very fine grain size. A better mechanism for returning the results of forked calls would help, rather than using shared memory. The simple CPUs here are not as bandwidth-hungry as a superscalar CMP would be; in that case, a faster bus, several buses or some other connection would be needed. Jamaica's split-transaction coherence protocol has been demonstrated to work, and its pipelining allows high bandwidth and low-latency cache to cache transfers.

The `WAIT` instruction was a great advantage to a multithreaded machine with severe lock contention. Since it is such a straightforward extension of the existing load-locked/store-conditional mechanism, it can be recommended for any single-chip multithreaded multiprocessor.

To combine multithreading with register windows, a heap-allocated register file was used. The performance increase over a static partition between the threads turns out to be fairly small, at most 10% on the recursive programs considered here. This suggests that the extra hardware complexity of heap allocation is probably not worth the effort, if there is any resulting effect on cycle time. The Jamaica processor has hardware spilling and filling of windows: if spills were more expensive, heap allocation may have a bigger advantage.

11.3 Future work

There are many aspects of Jamaica's architecture and supposed implementation which have not been addressed in this work. The simulator has over 40 configurable variables, including memory transaction timings, cache organisations and register set sizes. Even measuring the sensitivity of performance to each parameter individually would be an enormous task. Instead, two config-

urations were chosen to represent current and future designs, to assess the performance of the entire system.

This leaves many open questions about these and other choices. For example, the choice of write-invalidate over write-update in the coherence protocol was motivated, but has not been evaluated; a more relaxed model, like release consistency, could also be used. Similarly, register windows have been assumed, since earlier work showed their advantage; however, they have not been compared with flat files in the full Jamaica environment. To do this would mean rewriting much of the RTS and compilation system.

Several inefficiencies still remain in the RTS software, and may be removable with a more complex compilation procedure (§8.6). A full-blown JIT compiler, with garbage collection and dynamic optimisation, may have quite different behaviour to the relatively simple static compiler used here. Some hardware features may help Java-language programs: for example, object-oriented caches [Wil89][VRG98].

Other features are common in production microprocessors, but have not made it into Jamaica's simple CPUs: the best two examples are branch prediction (for both the outcome and target) and floating point arithmetic. Very accurate branch prediction is less necessary than for deep-pipeline machines. A floating-point unit was not needed for the benchmarks considered here, but would be for real programs.

Further into the future, so many simple CPUs will fit onto a single chip that it may be worth returning to superscalar or SMT elements. A good investigation would test Jamaica's thread distribution capabilities in such a system.

Some enhancements to the token mechanism are also possible. The current system is unreliable, in the sense that a token may or may not be returned. Load balancing for standard Java threads might be eased if a reliable system were available, e.g. an instruction which blocked until it could return a token (maybe waiting for a limited time). The window of opportunity for catching a token during a dynamic fork could be improved with a 'request token' – 'test for success' pair, where a token may be acquired at any time between the two. This may help programs with more limited available parallelism.

The ultimate test would be the design and layout of a Jamaica chip. This would assess the impact on cycle time of the more complex features: the bus protocol and the register windows.

Appendices

A. Results

A.1 Vector copy

Table 8: Vector copy, current

P	T	SpUp	CBW MB/s	MBW MB/s	Bus %	Act %	None %	RWB %	WRB %	Bank %
1	1	1.00	263	394	4.3	25.8	74.0	0.0	0.0	0.1
1	2	1.84	483	723	7.9	47.4	44.1	1.9	4.0	2.7
1	4	1.64	432	918	11.0	60.2	3.0	7.8	15.9	13.0
1	8	2.32	610	918	10.1	60.2	0.0	8.4	16.9	14.4
1	16	2.33	612	919	10.2	60.3	0.0	8.4	16.8	14.5
2	1	1.97	518	774	8.5	50.8	47.4	0.0	0.0	1.8
2	2	2.19	576	860	9.4	56.4	1.9	8.4	17.3	16.0
2	4	2.16	569	942	11.8	61.7	0.0	8.2	16.4	13.6
2	8	2.37	624	937	10.4	61.4	0.0	8.5	17.1	13.0
2	16	2.16	568	933	11.8	61.1	0.1	8.2	16.3	14.3
4	1	2.24	589	879	9.6	57.6	2.2	8.5	17.6	14.1
4	2	2.52	662	987	10.8	64.7	0.0	9.3	18.6	7.3
4	4	2.12	557	940	11.8	61.6	0.0	8.2	16.4	13.7
4	8	2.16	568	937	12.5	61.4	0.1	8.1	16.3	14.1
8	1	2.75	723	1078	11.8	70.6	0.0	7.3	14.6	7.5
8	2	2.43	639	950	10.4	62.2	0.0	9.0	17.9	10.9
8	4	2.11	555	937	13.3	61.4	0.0	8.0	16.0	14.6
16	1	2.46	647	961	10.6	63.0	0.0	9.1	18.1	9.8
16	2	2.40	631	935	14.4	61.2	0.0	8.6	17.1	13.1
32	1	2.41	633	936	17.4	61.3	0.0	8.7	17.4	12.5

Table 9: Vector copy, future

P	T	SpUp	CBW MB/s	MBW MB/s	Bus %	Act %	None %	RWB %	WRB %	Bank %
1	1	1.00	968	1450	6.3	7.6	92.1	0.0	0.3	0.0
1	2	2.00	1933	2892	12.7	15.2	83.9	0.3	0.5	0.2
1	4	2.95	2853	5100	24.5	26.7	66.5	1.2	2.6	3.1
1	8	5.52	5348	8275	39.1	43.4	39.5	3.1	6.8	7.2
1	16	7.05	6826	10587	50.3	55.5	15.3	5.5	11.3	12.3
2	1	1.99	1925	2880	12.6	15.1	83.9	0.2	0.6	0.1
2	2	3.62	3509	5244	23.0	27.5	65.3	1.2	2.9	3.1
2	4	5.59	5413	8539	41.0	44.8	37.1	3.5	7.4	7.2
2	8	7.18	6950	10786	50.4	56.6	13.0	5.9	12.0	12.6
2	16	7.56	7319	11456	54.0	60.1	4.5	6.8	13.7	15.0
4	1	3.67	3551	5302	23.3	27.8	65.4	1.1	2.8	3.0
4	2	6.00	5806	8683	38.1	45.5	35.5	3.7	7.8	7.5
4	4	7.17	6944	10811	50.5	56.7	12.6	5.9	12.2	12.6
4	8	7.60	7360	11497	53.5	60.3	4.2	6.9	13.8	14.8
8	1	6.17	5975	8920	39.1	46.8	34.9	3.6	7.8	7.0
8	2	7.33	7092	10614	46.9	55.6	12.2	6.0	12.2	14.0
8	4	7.61	7366	11476	53.0	60.2	4.6	6.8	13.8	14.7
16	1	7.73	7484	11142	48.9	58.4	9.6	6.4	12.9	12.7
16	2	7.97	7715	11531	51.2	60.5	4.9	6.9	13.8	14.0
32	1	8.22	7960	11791	51.9	61.8	2.6	7.2	14.5	13.9

A.2 Vector add

Table 10: Vector add, current

P	T	SpUp	CBW MB/s	MBW MB/s	Bus %	Act %	None %	RWB %	WRB %	Bank %
1	1	1.00	265	353	4.0	23.1	76.9	0.0	0.0	0.0
1	2	1.16	307	668	9.0	43.7	46.0	3.2	5.6	1.5
1	4	0.30	79	844	9.7	55.3	4.2	6.9	13.8	19.7
1	8	0.28	75	792	9.1	51.9	0.0	6.4	12.9	28.8
1	16	0.28	73	749	8.6	49.0	0.0	6.0	12.0	33.0
2	1	1.74	460	612	7.0	40.1	44.1	4.9	10.0	0.9
2	2	0.28	75	798	9.2	52.3	3.8	6.5	13.1	24.4
2	4	0.38	99	1059	12.2	69.4	0.0	8.7	17.3	4.6
2	8	2.04	538	961	12.8	63.0	0.1	6.5	13.0	17.4
2	16	2.12	560	962	12.6	63.1	0.1	6.5	13.0	17.3
4	1	2.36	625	830	9.6	54.4	16.2	6.7	13.4	9.3
4	2	0.29	77	819	9.4	53.7	0.0	6.5	13.0	26.8
4	4	0.38	101	1076	12.3	70.5	0.0	8.2	16.3	5.0
4	8	2.11	559	968	13.0	63.5	0.0	6.6	13.2	16.8
8	1	2.84	753	999	11.5	65.5	0.0	7.6	15.1	11.8
8	2	0.29	76	807	9.3	52.9	0.0	6.5	13.1	27.5
8	4	0.37	97	1037	11.9	67.9	0.0	7.8	15.6	8.7
16	1	2.84	752	997	11.5	65.3	0.0	7.8	15.6	11.3
16	2	0.27	72	763	8.8	50.0	0.0	6.2	12.4	31.4
32	1	2.91	771	1017	11.7	66.6	0.0	7.7	15.3	10.4

Table 11: Vector add, future

P	T	SpUp	CBW MB/s	MBW MB/s	Bus %	Act %	None %	RWB %	WRB %	Bank %
1	1	1.00	978	1301	6.0	6.8	93.1	0.0	0.0	0.0
1	2	1.21	1180	2604	13.6	13.7	85.1	0.2	0.4	0.6
1	4	2.43	2372	4681	24.2	24.5	70.6	0.8	1.5	2.5
1	8	5.05	4938	7428	40.6	39.0	48.0	2.0	4.5	6.6
1	16	6.79	6637	10170	57.0	53.3	22.0	4.0	8.3	12.4
2	1	1.91	1872	2491	11.4	13.1	84.1	0.1	0.8	1.9
2	2	1.97	1926	4882	24.1	25.6	70.2	0.6	1.5	2.1
2	4	5.30	5184	7844	42.3	41.1	44.4	2.3	5.0	7.1
2	8	7.25	7092	10586	56.4	55.5	17.7	4.4	9.2	13.2
2	16	7.72	7552	11621	63.5	60.9	6.5	5.4	11.0	16.2
4	1	3.64	3561	4734	21.8	24.8	68.3	0.5	2.8	3.6
4	2	5.36	5245	8001	41.9	42.0	43.5	2.5	5.2	6.8
4	4	7.24	7084	10664	55.9	55.9	16.3	4.6	9.5	13.7
4	8	7.84	7668	11612	60.7	60.9	5.7	5.6	11.2	16.6
8	1	6.41	6270	8331	38.3	43.7	41.7	2.5	5.7	6.4
8	2	7.38	7216	10685	54.3	56.0	16.4	4.5	9.3	13.7
8	4	7.84	7668	11550	59.3	60.6	5.6	5.6	11.2	17.0
16	1	8.51	8328	11044	50.8	57.9	13.4	5.0	10.4	13.4
16	2	8.02	7843	11528	57.4	60.4	6.3	5.4	10.9	16.9
32	1	9.22	9016	11914	54.9	62.5	3.8	5.9	12.0	15.8

A.3 Vector c+=a

Table 12: Vector c+=a, current

P	T	SpUp	CBW MB/s	MBW MB/s	Bus %	Act %	None %	RWB %	WRB %	Bank %
1	1	1.00	369	369	4.0	24.2	75.8	0.0	0.0	0.0
1	2	1.84	679	678	7.4	44.4	46.1	2.8	4.1	2.6
1	4	1.71	632	907	10.8	59.5	5.3	7.5	15.3	12.5
1	8	2.11	780	943	10.9	61.8	1.3	7.7	15.3	13.8
1	16	2.51	928	936	10.4	61.3	0.0	8.0	16.0	14.7
2	1	1.99	736	734	8.0	48.1	51.5	0.0	0.0	0.4
2	2	2.34	863	859	9.4	56.3	2.3	8.1	17.1	16.2
2	4	2.35	867	937	11.8	61.4	0.1	8.2	16.5	13.9
2	8	2.53	933	939	10.5	61.6	0.1	8.7	17.4	12.3
2	16	2.33	862	933	11.8	61.1	0.1	8.2	16.4	14.3
4	1	2.34	864	860	9.4	56.4	2.5	7.9	16.9	16.3
4	2	2.66	982	976	10.7	63.9	0.0	9.2	18.4	8.4
4	4	2.30	848	940	11.9	61.6	0.0	8.3	16.6	13.4
4	8	2.32	855	933	12.4	61.1	0.0	8.1	16.2	14.5
8	1	2.87	1058	1051	11.5	68.9	0.0	8.6	17.1	5.4
8	2	2.61	964	956	10.5	62.6	0.0	9.6	19.2	8.5
8	4	2.26	835	935	13.2	61.3	0.0	8.0	16.0	14.7
16	1	2.65	979	971	10.7	63.6	0.0	8.5	17.1	10.8
16	2	2.57	948	936	14.3	61.3	0.0	8.4	16.9	13.3
32	1	2.60	959	945	17.0	62.0	0.0	8.6	17.1	12.3

Table 13: Vector c+=a, current, SH not excl.

P	T	SpUp	CBW MB/s	MBW MB/s	Bus %	Act %	None %	RWB %	WRB %	Bank %
1	1	0.95	349	349	4.6	22.9	77.1	0.0	0.0	0.0
1	2	1.74	642	641	8.4	42.0	49.5	2.9	3.0	2.5
1	4	1.76	651	879	12.5	57.6	8.4	7.2	14.4	12.3
1	8	2.18	807	939	13.2	61.6	0.2	7.9	15.9	14.4
1	16	2.35	868	943	13.3	61.8	0.1	7.8	15.7	14.6
2	1	1.89	699	697	9.2	45.7	54.2	0.0	0.0	0.1
2	2	2.36	871	867	11.4	56.8	3.0	7.8	16.7	15.6
2	4	2.29	845	932	13.9	61.0	0.0	8.5	17.0	13.4
2	8	2.38	878	935	13.3	61.2	0.1	8.4	16.7	13.6
2	16	2.33	859	928	13.8	60.8	0.1	8.2	16.5	14.5
4	1	2.37	876	873	11.5	57.2	2.7	8.2	17.2	14.8
4	2	2.60	960	954	12.6	62.5	0.0	9.8	19.5	8.2
4	4	2.27	836	930	13.7	60.9	0.0	8.7	17.4	12.9
4	8	2.28	841	925	14.6	60.6	0.1	8.3	16.6	14.4
8	1	2.70	998	992	13.1	65.0	0.0	8.7	17.5	8.8
8	2	2.68	989	979	12.9	64.2	0.0	9.1	18.2	8.5
8	4	2.21	815	926	16.7	60.7	0.0	8.3	16.7	14.3
16	1	2.67	987	978	12.9	64.1	0.0	9.0	17.9	9.0
16	2	2.58	951	939	16.1	61.5	0.0	8.5	17.0	13.0
32	1	2.59	955	941	18.3	61.7	0.0	8.6	17.2	12.4

Table 14: Vector c+=a, future

P	T	SpUp	CBW MB/s	MBW MB/s	Bus %	Act %	None %	RWB %	WRB %	Bank %
1	1	1.00	1425	1423	6.2	7.5	92.4	0.0	0.1	0.0
1	2	1.97	2812	2804	12.3	14.7	84.6	0.1	0.5	0.1
1	4	2.93	4175	4971	24.0	26.1	67.4	1.1	2.3	3.1
1	8	5.38	7668	7932	38.0	41.6	42.8	2.8	6.1	6.7
1	16	6.93	9885	10234	49.4	53.7	19.1	5.1	10.5	11.7
2	1	1.99	2832	2823	12.4	14.8	84.6	0.1	0.4	0.1
2	2	3.61	5141	5123	22.5	26.9	66.0	1.2	2.7	3.3
2	4	5.60	7973	8369	40.4	43.9	38.4	3.4	7.1	7.2
2	8	7.26	10345	10691	50.0	56.0	13.6	5.8	11.8	12.8
2	16	7.74	11019	11499	54.4	60.3	4.5	6.8	13.6	14.9
4	1	3.66	5215	5192	22.8	27.2	66.3	1.0	2.6	2.9
4	2	6.08	8664	8636	37.9	45.3	36.9	3.6	7.6	6.6
4	4	7.25	10336	10765	50.5	56.4	12.8	5.9	12.1	12.7
4	8	7.70	10979	11455	53.5	60.1	4.4	6.8	13.8	14.9
8	1	6.19	8824	8783	38.5	46.1	36.0	3.5	7.5	7.0
8	2	7.47	10648	10621	46.9	55.7	12.8	6.0	12.1	13.5
8	4	7.76	11050	11481	53.1	60.2	4.5	6.9	13.8	14.6
16	1	7.85	11184	11102	48.7	58.2	9.9	6.3	12.8	12.8
16	2	8.13	11583	11526	51.1	60.4	5.0	6.9	13.8	13.9
32	1	8.38	11940	11786	51.8	61.8	2.5	7.2	14.5	14.0

Table 15: Vector c+=a, future, SH not excl.

P	T	SpUp	CBW MB/s	MBW MB/s	Bus %	Act %	None %	RWB %	WRB %	Bank %
1	1	0.94	1346	1343	7.1	7.0	92.9	0.0	0.0	0.0
1	2	1.83	2612	2605	13.7	13.7	85.7	0.1	0.5	0.1
1	4	2.81	3999	4672	26.9	24.5	69.8	0.9	2.1	2.6
1	8	5.15	7339	7587	42.9	39.8	45.7	2.6	5.7	6.2
1	16	6.78	9662	10046	57.1	52.7	20.7	5.0	10.3	11.4
2	1	1.89	2688	2682	14.1	14.1	85.8	0.0	0.0	0.1
2	2	3.43	4886	4868	25.6	25.5	68.1	1.1	2.4	2.9
2	4	5.36	7634	8055	45.9	42.2	41.3	3.1	6.7	6.7
2	8	7.11	10135	10534	58.5	55.2	15.3	5.6	11.5	12.4
2	16	7.65	10899	11449	63.8	60.0	4.8	6.8	13.7	14.6
4	1	3.49	4979	4959	26.1	26.0	68.3	0.9	2.3	2.5
4	2	5.89	8397	8361	44.0	43.8	39.3	3.4	7.0	6.4
4	4	7.15	10187	10663	58.9	55.9	13.9	5.8	11.9	12.4
4	8	7.65	10899	11435	62.8	60.0	4.4	6.8	13.8	15.0
8	1	6.04	8602	8564	45.1	44.9	38.9	3.2	7.0	6.1
8	2	7.44	10601	10581	56.0	55.5	13.2	6.0	12.1	13.2
8	4	7.70	10979	11443	62.6	60.0	4.7	6.9	13.8	14.6
16	1	7.78	11080	10998	57.9	57.7	10.8	6.2	12.7	12.6
16	2	8.09	11538	11484	61.0	60.2	5.3	6.8	13.8	13.9
32	1	8.36	11905	11757	62.1	61.6	2.6	7.2	14.5	14.1

A.4 bounce

Table 16: Bounce, current

P	T	Cycles	Bus%
1	1	16439	0.3
1	2	16512	0.4
1	4	16642	0.6
1	8	16858	0.7
1	16	17016	0.6
2	1	29938	15.2
2	2	19217	11.5
2	4	73270	12.7
2	8	61204	6.1
2	16	98603	3.9
4	1	35290	19.1
4	2	47476	25.1
4	4	71336	12.6
4	8	84623	7.1
8	1	38030	20.8
8	2	49908	31.9
8	4	71024	14.1
16	1	39782	21.8
16	2	50126	31.5
32	1	41196	22.9

Table 17: Bounce, future

P	T	Cycles	Bus%
1	1	18475	1.2
1	2	18369	1.4
1	4	18495	1.5
1	8	18611	1.9
1	16	18374	2.5
2	1	35234	24.1
2	2	61874	27.0
2	4	75380	27.5
2	8	110858	31.2
2	16	157058	20.2
4	1	84674	34.3
4	2	88881	30.5
4	4	106890	31.4
4	8	134306	33.9
8	1	101642	35.1
8	2	100356	31.4
8	4	115378	32.1
16	1	110770	35.4
16	2	110428	31.7
32	1	116410	35.8

Table 18: Bounce, current, no wait

P	T	Cyc	Bus%
1	1	16439	0.3
1	2	24435	0.3
1	4	34492	0.3
1	8	45636	0.2
1	16	28727	0.3
2	1	29850	15.2
2	2	40726	15.4
2	4	996235	0.5
2	8	5087098	0.2
2	16	7379691	0.1
4	1	35286	19.1
4	2	132697	7.4
4	4	2000902	0.5
4	8	5256769	0.2
8	1	37996	20.7
8	2	174054	7.7
8	4	1890368	0.7
16	1	39666	21.7
16	2	159444	9.0
32	1	40966	22.6

Table 19: Bounce, future, no wait

P	T	Cyc	Bus%
1	1	18475	1.2
1	2	27274	0.9
1	4	37325	0.7
1	8	47399	0.8
1	16	18414	2.7
2	1	55322	30.0
2	2	140173	9.1
2	4	75678	22.3
2	8	2002538	1.7
2	16	6946685	0.4
4	1	84850	34.5
4	2	202513	10.1
4	4	376802	9.5
4	8	848674	5.4
8	1	101634	35.1
8	2	241237	10.2
8	4	338821	11.9
16	1	110658	35.5
16	2	282525	9.6
32	1	116650	35.9

A.5 nfib

Table 20: nfib, current, 80w

GS	P	T	Token				Oracle			
			SpUp	Bus%	Thr	CrRt	SpUp	Bus%	Thr	CrRt
3	1	1	0.87	0.0	0	596232.0	0.87	0.0	0	596232.0
3	1	2	0.86	0.0	24	23872.9	0.86	0.0	24	23872.9
3	1	4	0.86	0.0	36	16140.3	0.86	0.0	36	16140.3
3	1	8	0.85	0.1	93	6453.4	0.85	0.1	93	6453.4
3	2	1	1.52	0.3	128	2623.9	1.71	0.1	23	12537.8
3	2	2	1.68	0.2	82	3692.2	1.65	0.2	81	3821.7
3	2	4	1.66	0.5	254	1221.3	1.69	0.4	189	1610.9
3	2	8	1.67	0.8	415	742.1	1.68	0.7	398	772.0
3	4	1	2.95	1.1	220	792.1	3.21	0.4	75	2112.9
3	4	2	3.18	1.1	238	679.1	3.15	0.8	183	889.7
3	4	4	3.19	2.1	477	338.9	3.26	1.8	436	362.7
3	4	8	3.24	2.7	616	257.8	3.30	2.2	466	334.8
3	8	1	5.55	3.1	358	259.2	5.61	3.3	395	232.1
3	8	2	5.51	4.9	627	149.0	5.97	5.2	608	142.0
3	8	4	6.05	7.3	871	97.8	6.18	7.7	911	91.6
3	8	8	6.10	7.8	902	93.7	6.27	8.5	994	82.7
3	16	1	8.45	8.0	645	94.5	11.19	11.2	703	65.5
3	16	2	10.33	13.0	892	55.9	11.53	16.4	1014	44.1
3	16	4	11.25	22.7	1448	31.7	11.64	24.4	1505	29.4
3	16	8	11.44	19.8	1219	37.0	11.73	24.1	1435	30.6
3	32	1	15.78	19.9	844	38.7	21.25	21.7	704	34.4
3	32	2	19.69	28.7	1005	26.1	21.58	34.0	1080	22.1
3	32	4	21.55	41.8	1326	18.0	21.01	42.2	1377	17.8
3	32	8	20.91	53.6	1779	13.9	21.29	45.9	1455	16.6
9	1	1	0.88	0.0	0	588304.0	0.88	0.0	0	588304.0
9	1	2	0.88	0.0	8	65387.6	0.88	0.0	8	65387.6
9	1	4	0.88	0.0	8	65392.9	0.88	0.0	8	65392.9
9	1	8	0.86	0.1	41	14208.6	0.86	0.1	41	14208.6
9	2	1	1.48	0.0	12	26829.1	1.71	0.0	8	33544.0
9	2	2	1.70	0.1	22	13208.3	1.66	0.1	15	19419.8
9	2	4	1.69	0.1	49	6102.5	1.71	0.1	50	5901.9
9	2	8	1.71	0.2	83	3583.9	1.71	0.2	72	4137.1
9	4	1	2.97	0.1	17	9663.7	3.16	0.1	16	9604.4

Table 20: nfib, current, 80w (Continued)

GS	P	T	Token				Oracle			
			SpUp	Bus%	Thr	CrRt	SpUp	Bus%	Thr	CrRt
9	4	2	3.27	0.2	29	5258.9	3.27	0.3	51	3038.6
9	4	4	3.27	0.4	77	2022.4	3.31	0.4	87	1768.9
9	4	8	3.36	0.6	107	1422.6	3.23	0.6	100	1581.6
9	8	1	5.27	0.4	27	3495.3	5.56	0.5	48	1892.8
9	8	2	5.14	0.8	85	1166.3	6.23	0.8	83	985.5
9	8	4	5.89	1.2	128	679.1	6.25	1.4	150	546.7
9	8	8	6.01	1.5	147	579.8	6.34	1.6	143	565.5
9	16	1	7.21	1.0	70	1007.4	11.12	1.6	73	627.1
9	16	2	9.10	1.9	126	446.5	11.80	2.5	124	349.8
9	16	4	11.86	3.2	168	257.3	12.41	3.5	176	234.9
9	16	8	10.98	3.2	175	267.0	11.63	3.4	173	255.0
9	32	1	10.87	2.1	78	600.7	18.27	3.7	104	269.0
9	32	2	18.25	4.3	128	219.2	22.16	6.2	159	145.5
9	32	4	16.96	5.4	184	164.5	20.27	6.4	183	138.4
9	32	8	16.52	5.6	193	161.0	18.67	6.2	189	145.5

Table 21: nfib, future, 80w

GS	P	T	Token				Oracle			
			SpUp	Bus%	Thr	CrRt	SpUp	Bus%	Thr	CrRt
3	1	1	0.86	0.0	0	598722.0	0.86	0.0	0	598722.0
3	1	2	0.86	0.1	24	23971.6	0.86	0.1	24	23971.6
3	1	4	0.86	0.1	44	13331.4	0.86	0.1	44	13331.4
3	1	8	0.84	0.4	76	7975.3	0.84	0.4	76	7975.3
3	2	1	1.47	1.3	141	2477.6	1.69	0.4	34	8750.5
3	2	2	1.63	0.5	49	6323.1	1.64	0.3	36	8519.4
3	2	4	1.67	1.3	166	1856.3	1.69	1.3	180	1687.7
3	2	8	1.64	2.7	358	879.2	1.65	2.4	337	925.4
3	4	1	2.80	4.5	244	753.4	3.07	2.4	115	1451.2
3	4	2	2.89	6.0	362	492.3	3.06	4.3	256	656.1
3	4	4	3.10	7.8	489	340.0	3.18	7.0	424	381.9
3	4	8	3.12	10.5	607	272.7	3.12	9.6	577	286.1
3	8	1	4.76	13.6	462	234.3	4.96	14.2	481	216.2
3	8	2	5.51	14.3	462	202.5	5.51	19.1	614	152.6
3	8	4	5.71	27.6	894	101.1	5.65	37.7	1275	71.7
3	8	8	5.56	37.7	1243	74.8	5.69	37.2	1194	76.0

Table 21: nfib, future, 80w (Continued)

GS	P	T	Token				Oracle			
			SpUp	Bus%	Thr	CrRt	SpUp	Bus%	Thr	CrRt
3	16	1	6.97	31.3	763	97.0	8.90	43.2	840	69.1
3	16	2	8.34	48.5	1036	59.8	9.23	56.0	1082	51.7
3	16	4	9.34	59.6	1137	48.6	8.99	68.3	1384	41.5
3	16	8	8.87	70.2	1443	40.3	8.46	75.9	1606	38.0
3	32	1	11.61	72.2	1069	41.6	12.77	79.4	1094	36.9
3	32	2	11.38	83.7	1291	35.1	11.36	83.9	1325	34.3
3	32	4	10.20	87.6	1525	33.2	8.91	90.1	1839	31.5
3	32	8	7.92	91.3	2050	31.8	7.12	91.5	2368	30.6
9	1	1	0.87	0.0	0	590786.0	0.87	0.0	0	590786.0
9	1	2	0.87	0.1	8	65661.6	0.87	0.1	8	65661.6
9	1	4	0.87	0.1	18	31115.5	0.87	0.1	18	31115.5
9	1	8	0.85	0.4	31	18945.8	0.85	0.4	31	18945.8
9	2	1	1.47	0.2	12	27074.6	1.68	0.2	8	34103.3
9	2	2	1.64	0.2	20	14967.7	1.65	0.2	15	19579.6
9	2	4	1.71	0.4	41	7210.1	1.73	0.4	49	5989.6
9	2	8	1.70	1.0	74	4064.6	1.68	0.8	57	5311.9
9	4	1	2.89	0.6	25	6878.8	3.17	0.5	15	10184.1
9	4	2	2.84	1.0	55	3246.9	3.13	1.0	56	2900.8
9	4	4	3.18	1.6	89	1803.3	3.21	1.5	86	1848.9
9	4	8	3.16	2.3	90	1795.0	3.10	2.5	123	1342.6
9	8	1	4.07	1.7	51	2444.8	5.20	1.5	40	2424.0
9	8	2	5.52	3.0	82	1128.8	5.74	3.7	110	811.1
9	8	4	5.90	4.5	128	678.9	5.85	4.7	144	609.4
9	8	8	5.70	6.2	163	552.9	6.13	6.0	143	585.8
9	16	1	6.41	4.3	90	885.3	9.83	5.7	80	648.8
9	16	2	9.58	7.1	112	477.5	10.32	8.8	136	365.6
9	16	4	10.32	10.7	158	315.1	10.79	11.5	175	272.2
9	16	8	10.88	12.3	163	289.6	10.42	12.4	172	286.8
9	32	1	9.16	8.8	117	477.9	17.15	16.2	138	216.8
9	32	2	16.74	16.0	133	230.3	17.97	20.0	172	166.2
9	32	4	18.62	22.3	179	154.2	17.63	20.9	179	162.9
9	32	8	16.94	23.2	187	162.3	15.86	20.6	188	172.4

Table 22: nfib, current, 24w

GS	P	T	Static				Heap			
			SpUp	Bus%	Thr	CrRt	SpUp	Bus%	Thr	CrRt
3	1	1	0.87	0.0	0	596232.0	0.87	0.0	0	596232.0
3	1	2	0.86	0.0	12	46293.7	0.86	0.0	16	35389.2
3	1	4	0.78	0.1	20	31607.0	0.85	0.1	28	20993.7
3	2	1	1.52	0.3	128	2623.9	1.52	0.3	128	2623.9
3	2	2	1.68	0.2	68	4454.3	1.66	0.3	102	3011.1
3	2	4	1.53	0.4	72	4629.9	1.61	0.6	198	1605.7
3	4	1	2.95	1.1	220	792.1	2.95	1.1	220	792.1
3	4	2	3.02	1.3	243	700.5	3.02	1.6	322	528.6
3	4	4	2.84	2.8	453	400.7	3.02	2.9	529	322.6
3	8	1	5.55	3.1	358	259.2	5.55	3.1	358	259.2
3	8	2	5.39	5.6	634	150.8	5.60	4.6	521	176.4
3	8	4	5.05	9.7	863	118.3	5.53	10.3	1025	90.9
3	16	1	8.45	8.0	645	94.5	8.45	8.0	645	94.5
3	16	2	10.40	14.3	827	59.9	10.12	14.6	945	53.9
3	16	4	8.10	20.6	1056	60.3	9.23	24.0	1395	40.1
3	32	1	14.10	19.6	940	38.9	14.10	19.6	940	38.9
3	32	2	17.77	34.7	1176	24.7	19.24	29.7	952	28.1
3	32	4	9.70	35.3	1336	39.8	11.89	37.4	1535	28.2
9	1	1	0.88	0.0	0	588304.0	0.88	0.0	0	588304.0
9	1	2	0.87	0.0	6	84816.3	0.87	0.0	6	84784.3
9	1	4	0.79	0.1	10	59600.0	0.86	0.1	14	40083.5
9	2	1	1.48	0.0	12	26829.1	1.48	0.0	12	26829.1
9	2	2	1.69	0.1	17	16980.0	1.67	0.1	18	16229.8
9	2	4	1.53	0.2	30	10877.9	1.64	0.4	47	6565.2
9	4	1	2.97	0.1	17	9663.7	2.97	0.1	17	9663.7
9	4	2	3.13	0.4	35	4582.6	3.17	0.4	41	3873.5
9	4	4	2.81	1.2	82	2211.4	3.18	1.1	59	2705.2
9	8	1	5.27	0.4	27	3495.3	5.27	0.4	27	3495.3
9	8	2	5.37	1.4	90	1056.6	5.24	1.3	90	1082.4
9	8	4	4.82	3.1	118	900.1	5.25	3.0	131	744.6
9	16	1	7.21	1.0	70	1007.4	7.21	1.0	70	1007.4
9	16	2	10.09	3.6	115	440.9	8.28	2.7	116	532.4
9	16	4	7.37	7.3	169	412.1	8.41	7.5	184	331.8
9	32	1	12.25	2.4	85	489.6	12.25	2.4	85	489.6
9	32	2	15.55	7.6	141	233.7	17.44	6.9	125	234.9
9	32	4	9.06	13.3	169	335.2	9.77	11.8	178	295.1

Table 23: nfib, future, 24w

GS	P	T	Static				Heap			
			SpUp	Bus%	Thr	CrRt	SpUp	Bus%	Thr	CrRt
3	1	1	0.86	0.0	0	598722.0	0.86	0.0	0	598722.0
3	1	2	0.85	0.1	12	46813.7	0.85	0.1	18	32060.7
3	1	4	0.76	0.3	20	32357.4	0.83	0.4	53	11585.4
3	2	1	1.47	1.3	141	2477.6	1.47	1.3	141	2477.6
3	2	2	1.62	0.7	44	7081.6	1.61	0.9	79	4009.4
3	2	4	1.43	1.9	133	2696.3	1.56	1.9	95	3443.0
3	4	1	2.80	4.5	244	753.4	2.80	4.5	244	753.4
3	4	2	2.83	5.5	291	625.7	2.85	6.5	375	481.8
3	4	4	2.62	9.5	406	483.9	2.85	8.9	362	498.8
3	8	1	4.76	13.6	462	234.3	4.76	13.6	462	234.3
3	8	2	5.08	18.1	540	187.9	5.25	16.2	489	200.8
3	8	4	4.67	25.1	524	210.9	4.74	35.8	1044	104.3
3	16	1	6.97	31.3	763	97.0	6.97	31.3	763	97.0
3	16	2	8.83	39.8	661	88.4	8.50	42.8	818	74.2
3	16	4	6.71	56.0	779	98.8	6.75	72.7	1486	51.5
3	32	1	11.46	71.7	1083	41.6	11.46	71.7	1083	41.6
3	32	2	10.70	84.2	1246	38.7	11.18	84.5	1266	36.5
3	32	4	6.21	80.8	1134	73.3	6.06	92.2	2066	41.3
9	1	1	0.87	0.0	0	590786.0	0.87	0.0	0	590786.0
9	1	2	0.86	0.1	5	100076.3	0.86	0.1	7	75144.2
9	1	4	0.77	0.3	11	55914.2	0.84	0.4	15	38574.6
9	2	1	1.47	0.2	12	27074.6	1.47	0.2	12	27074.6
9	2	2	1.64	0.4	12	24192.2	1.62	0.5	16	18771.4
9	2	4	1.46	1.2	44	7846.8	1.61	1.2	29	10732.1
9	4	1	2.89	0.6	25	6878.8	2.89	0.6	25	6878.8
9	4	2	2.72	1.6	57	3281.3	2.82	1.6	52	3460.6
9	4	4	2.73	4.3	66	2828.0	2.92	4.3	76	2296.5
9	8	1	4.07	1.7	51	2444.8	4.07	1.7	51	2444.8
9	8	2	5.55	5.1	74	1241.0	5.30	4.3	65	1476.8
9	8	4	4.63	11.7	83	1328.1	4.98	12.0	131	786.1
9	16	1	6.41	4.3	90	885.3	6.41	4.3	90	885.3
9	16	2	9.08	12.0	100	563.4	9.50	11.5	111	485.7
9	16	4	7.14	26.3	115	624.3	7.38	24.4	164	424.2
9	32	1	9.16	8.8	117	477.9	9.16	8.8	117	477.9
9	32	2	15.41	27.6	132	252.2	14.94	23.2	141	243.6
9	32	4	8.52	50.2	168	358.8	9.33	45.0	187	294.7

A.6 jnfib

Table 24: jnfib(21), current

GS	P	T	Light				Medium			
			SpUp	Bus%	Thr	CrRt	SpUp	Bus%	Thr	CrRt
3	1	1	0.66	0.0	0	860559.0	0.68	0.1	0	835592.0
3	1	2	0.66	0.1	0	860986.0	0.68	0.1	0	836062.0
3	1	4	0.65	0.1	25	34010.3	0.66	0.1	24	34455.5
3	2	1	0.66	0.0	0	860541.0	0.68	0.1	0	835533.0
3	2	2	1.04	0.2	12	42370.6	1.06	0.2	17	29966.8
3	2	4	1.16	0.5	59	8186.1	1.17	0.8	77	6276.4
3	4	1	1.70	0.9	92	3618.9	1.76	1.2	64	5004.9
3	4	2	1.85	1.2	125	2444.4	2.03	2.0	91	3053.7
3	4	4	2.05	2.4	204	1358.3	1.50	4.3	268	1417.0
3	8	1	3.27	2.7	160	1083.6	3.23	5.4	136	1290.6
3	8	2	3.75	3.6	155	977.4	2.49	11.4	377	607.8
3	8	4	3.56	6.6	289	553.1	1.30	10.1	562	783.0
3	16	1	5.05	8.5	348	323.9	4.40	16.9	254	508.5
3	16	2	5.81	10.7	327	300.0	2.11	18.9	703	383.9
3	16	4	5.00	17.7	578	197.4	0.96	13.2	949	628.0
3	32	1	7.12	20.5	416	192.5	3.63	26.4	474	331.7
3	32	2	5.64	22.6	547	184.9	1.58	25.1	1114	325.2
3	32	4	4.13	24.7	720	192.0	0.74	19.4	1666	463.2
9	1	1	0.72	0.1	0	789215.0	0.72	0.1	0	788028.0
9	1	2	0.72	0.1	0	789638.0	0.72	0.1	0	788498.0
9	1	4	0.71	0.1	8	89440.1	0.71	0.1	14	53906.9
9	2	1	0.72	0.1	0	789197.0	0.72	0.1	0	787969.0
9	2	2	1.14	0.2	5	83821.0	1.13	0.2	7	63074.2
9	2	4	1.30	0.4	19	22007.5	1.26	0.5	29	15072.7
9	4	1	1.95	0.3	13	20942.8	1.96	0.4	12	22447.4
9	4	2	2.16	0.6	23	11003.2	2.21	1.0	27	9230.4
9	4	4	2.29	1.1	34	7123.5	2.12	2.2	73	3646.7
9	8	1	4.01	1.1	20	6776.0	3.80	1.6	28	5185.9
9	8	2	4.28	2.0	38	3418.4	3.86	3.5	61	2388.8
9	8	4	4.04	3.3	65	2143.0	2.71	5.4	129	1624.2
9	16	1	6.11	3.1	51	1797.8	6.34	5.1	45	1958.4
9	16	2	6.98	5.2	63	1278.8	5.18	10.4	123	888.6
9	16	4	6.08	8.7	115	809.7	3.61	12.6	200	787.7
9	32	1	9.70	7.4	66	878.5	9.19	11.8	58	1053.8
9	32	2	9.42	11.7	97	619.0	6.11	21.2	195	476.7
9	32	4	7.16	16.1	141	562.0	4.18	20.2	231	589.1

Table 25: jnfib(21), future

GS	P	T	Light				Medium			
			SpUp	Bus%	Thr	CrRt	SpUp	Bus%	Thr	CrRt
3	1	1	0.67	0.2	0	873817.0	0.69	0.2	0	849134.0
3	1	2	0.67	0.2	0	874745.0	0.69	0.2	0	849928.0
3	1	4	0.64	0.5	21	41300.8	0.64	0.7	42	21050.4
3	2	1	0.67	0.2	0	873561.0	0.69	0.2	0	848654.0
3	2	2	1.01	0.6	13	41117.8	1.02	0.8	21	25912.0
3	2	4	1.08	1.7	44	12059.4	1.09	3.3	85	6228.6
3	4	1	1.64	2.2	55	6349.7	1.60	3.8	51	7011.3
3	4	2	1.79	3.4	83	3883.1	1.85	6.4	85	3668.1
3	4	4	1.86	7.8	167	1872.0	1.48	13.5	209	1875.0
3	8	1	2.55	9.6	194	1175.6	2.36	16.3	139	1764.7
3	8	2	3.02	11.9	177	1084.1	2.05	21.5	226	1254.1
3	8	4	2.87	23.5	351	577.2	0.92	31.7	855	742.8
3	16	1	4.16	20.0	238	586.9	1.75	36.6	405	821.8
3	16	2	3.95	33.7	427	345.5	1.09	34.9	783	683.3
3	16	4	3.55	51.0	630	260.6	0.73	33.3	1088	733.9
3	32	1	4.46	49.6	433	301.4	1.16	46.6	708	709.5
3	32	2	4.01	60.6	605	239.9	0.71	45.2	1379	598.3
3	32	4	2.75	74.1	1059	200.2	0.52	40.1	1635	680.6
9	1	1	0.73	0.2	0	802489.0	0.73	0.2	0	801606.0
9	1	2	0.73	0.2	0	803409.0	0.73	0.2	0	802368.0
9	1	4	0.70	0.4	8	92265.0	0.70	0.5	14	55596.3
9	2	1	0.73	0.2	0	802233.0	0.73	0.2	0	801126.0
9	2	2	1.11	0.6	5	87341.5	1.10	0.8	10	48217.5
9	2	4	1.24	1.5	26	17441.8	1.22	2.0	26	17733.6
9	4	1	1.88	1.2	13	22186.9	1.82	1.7	14	21355.5
9	4	2	2.15	2.0	19	13568.9	2.07	3.4	25	10837.8
9	4	4	2.21	4.3	38	6764.1	1.90	7.3	73	4148.3
9	8	1	3.41	4.6	39	4273.8	3.25	7.0	36	4856.4
9	8	2	3.57	6.4	41	3887.5	3.34	11.6	55	3123.1
9	8	4	3.41	11.0	64	2635.6	2.47	17.2	118	1986.6
9	16	1	5.47	9.7	44	2369.8	4.41	17.9	64	2034.1
9	16	2	5.35	15.8	71	1514.1	3.93	26.4	105	1402.6
9	16	4	4.68	24.4	106	1165.6	2.72	33.2	210	1015.6
9	32	1	7.20	20.8	70	1141.0	4.27	39.2	138	982.3
9	32	2	6.37	30.9	114	797.2	3.45	39.8	184	914.8
9	32	4	5.42	48.8	202	530.0	2.64	41.1	231	953.2

Table 26: jnfib(n), current, P=32, T=2

GS	n	Serial cycles	SpUp	MBW MB/s	Bus%	Util%	Rel. Inst	Thr	CrRt
3	21	571221	5.64	633.2	22.6	35.1	2.00	547	185.2
3	22	912570	7.06	637.1	21.6	41.9	1.91	697	185.5
3	23	1464136	7.43	672.3	20.9	43.8	1.89	1048	188.0
3	24	2357590	9.02	617.0	18.7	50.5	1.81	1280	204.3
3	25	3801408	8.96	644.7	18.5	50.2	1.82	2100	202.0
3	26	6136392	8.90	652.3	18.1	49.9	1.82	3428	201.1
3	27	9915478	9.99	609.4	17.1	54.5	1.78	4467	222.1
3	28	16029020	10.93	580.8	16.5	58.4	1.74	6191	236.9
3	29	25920350	10.82	587.5	17.2	57.7	1.74	9944	240.9
3	30	41927254	11.35	567.4	16.8	59.9	1.73	14689	251.4
9	21	571221	9.42	308.0	11.7	44.7	1.48	97	625.4
9	22	912570	11.81	300.4	10.4	54.9	1.46	115	672.1
9	23	1464136	13.43	276.6	9.5	62.5	1.46	175	623.0
9	24	2357590	16.06	256.5	8.5	73.8	1.45	217	676.5
9	25	3801408	17.18	223.2	7.2	78.9	1.45	314	704.5
9	26	6136392	18.66	200.3	6.3	84.7	1.44	448	734.2
9	27	9915478	19.18	168.8	5.3	86.6	1.44	612	844.6
9	28	16029020	19.80	151.6	4.6	88.9	1.44	902	897.4
9	29	25920350	20.78	137.3	4.0	92.8	1.43	1265	986.2
9	30	41927254	20.99	122.1	3.5	93.4	1.43	1821	1097.1

B. Simulation details

B.1 Instruction set

V_x is the value of register R_x (or the immediate value in its place). Ctrl[x] is a control register.

Arithmetic/logical

`OP Ra, Rb, Rc` / `OP Ra, Imm, Rc`

$(-128 \leq \mathrm{Imm} \leq 127)$

ADD	$R_c \leftarrow V_a + V_b$
SUB	$R_c \leftarrow V_a - V_b$
CMPEQ	$R_c \leftarrow (V_a = V_b)$
CMPLE	$R_c \leftarrow (V_a \leq V_b)$
CMPLT	$R_c \leftarrow (V_a < V_b)$
CMPULE	$R_c \leftarrow (V_a \leq V_b)$
CMPULT	$R_c \leftarrow (V_a < V_b)$
S4ADD	$R_c \leftarrow 4 \cdot V_a + V_b$
S8ADD	$R_c \leftarrow 8 \cdot V_a + V_b$
S4SUB	$R_c \leftarrow 4 \cdot V_a - V_b$
S8SUB	$R_c \leftarrow 8 \cdot V_a - V_b$
AND	$R_c \leftarrow V_a \wedge V_b$
BIC	$R_c \leftarrow V_a \wedge \neg V_b$
BIS	$R_c \leftarrow V_a \vee V_b$
EQV	$R_c \leftarrow V_a \oplus \neg V_b$

ORNOT	$R_c \leftarrow V_a \vee \neg V_b$
XOR	$R_c \leftarrow V_a \oplus V_b$
SLL	$R_c \leftarrow V_a \ll V_b$
SRL	$R_c \leftarrow V_a \gg V_b$ (unsigned)
SRA	$R_c \leftarrow V_a \gg V_b$ (signed)
CMOVEQ	if $(V_a = 0)$ { $R_c \leftarrow V_b$ }
CMOVGE	if ($V_a \geq 0$) { $R_c \leftarrow V_b$ }
CMOVGT	if ($V_a > 0$) { $R_c \leftarrow V_b$ }
CMOVLBC	if $((V_a \wedge 1) = 0)$ { $R_c \leftarrow V_b$ }
CMOVLBS	if $((V_a \wedge 1) = 1)$ { $R_c \leftarrow V_b$ }
CMOVLE	if $(V_a \leq 0)$ { $R_c \leftarrow V_b$ }
CMOVLT	if $(V_a < 0)$ { $R_c \leftarrow V_b$ }
CMOVNE	if ($V_a \neq 0$) { $R_c \leftarrow V_b$ }
MUL	$R_c \leftarrow V_a \times V_b$
TRQ	Token request; see §5.2.1
RCR	$R_c \leftarrow \text{Ctrl}[V_b]$
WCR	$\text{Ctrl}[V_b] \leftarrow V_a$
SIRQ	Send IRQ V_b to threadId V_a
EVICT	Evict a frame; $R_c \leftarrow 1$or 0

Memory

`OP Ra, disp(Rb)`, 16-bit signed displacement

`LDA`	$R_a \leftarrow \text{disp} + R_b$
`LDAH`	$R_a \leftarrow \text{disp} \ll 16 + R_b$
`LDL`	$R_a \leftarrow \text{Mem}[\text{disp} + R_b]$
`STL`	$\text{Mem}[\text{disp} + R_b] \leftarrow R_a$
`LDB`	$R_a \leftarrow \text{Mem}[\text{disp} + R_b]$, byte, sign-extended
`LDBU`	$R_a \leftarrow \text{Mem}[\text{disp} + R_b]$, byte, zero-extended
`STB`	$\text{Mem}[\text{disp} + R_b] \leftarrow R_a$, byte
`LDL_L`	$R_a \leftarrow \text{Mem}[\text{disp} + R_b]$, set lock_base, clear lock_flag
`STL_C`	if (!lock_flag) $\{\text{Mem}[\text{disp} + R_b] \leftarrow R_a ; R_a \leftarrow 1\}$ else $\{R_a \leftarrow 0\}$
`WAIT`	Sleep until lock_flag set

Control transfers

`OP Ra, disp`, signed 21-bit displacement

`BEQ`	Branch if $R_a = 0$
`BGE`	Branch if $R_a \geq 0$
`BGT`	Branch if $R_a > 0$
`BLBC`	Branch if $R_a \wedge 1 = 0$
`BLBS`	Branch if $R_a \wedge 1 = 1$
`BLE`	Branch if $R_a \leq 0$
`BLT`	Branch if $R_a < 0$
`BNE`	Branch if $R_a \neq 0$
`BR`	Branch
`BSR`	Branch to subroutine
`THB`	Thread branch (fork: requires a token)

`OP disp(Ra)`, signed 16-bit displacement

`JSR`	Jump to subroutine
`JMP`	Jump
`RET`	Return (takes address from `%i7`)
`THJ`	Thread jump (fork: requires a token)
`RTI`	Return from interrupt

B.2 PALcode routines

Low-level atomic routine needed by the runtime system, and some debugging functions:

ctxreplace	Atomic exchange of the threadId, flags and spill area information with registers `%o0`–`%o5`.
exit	Terminate the simulation
printi	Print all the integer registers
timestamp	Print the elapsed cycle count
zeroPerfCounters	Reset all the internal counters (cache hits/misses, instruction counts, etc)

Input/output, passed through to the simulator's C library: fopen, fputc, fgetc, fclose, ungetc, feof.

Floating point arithmetic (not yet supported through the instruction set): fadd, fsub, fmul, fdiv, frem, fneg, i2f, l2f, f2i, f2l, f2d, d2f, fcmpl, fcmpg, dadd, dsub, dmul, ddiv, drem, dneg, i2d, l2d, d2i, d2l, dcmpl, dcmpg, dsqrt, dsin, dcos, dtan, dasin, dacos, datan, dexp, dlog, dIEEEremainder, dceil, drint, datan2, dpow, dfloor.

B.3 The cache coherence protocol

This section supplements Table 4 and Table 5 in §7.2.4. Cycle numbers are in parentheses, and are illustrated in Figure 20 on page 97.

Read-shared (SH)

- Master (3): set Addr, Type (SH), and ID.
- Slaves (4): If the line is
  - *in cache*: Supply Data, and assert Found, Shared. Also, make the cache state non-exclusive.
  - *in WB*: Supply Data, and assert Found, Own. Also assert Excl if the line is exclusive in the writeback buffer. Cancel the writeback request.
  - *pending miss*: Assert Shared.

- L2/memory (4/5): Assert MAccept if the request could be accepted (i.e. the queues are not full; §7.5.3). If the line is
  - *in L2*: Supply Data if an L1 didn't indicate Found in cycle 5, and assert Found. Also assert Excl if the line is exclusive in the L2.
- Master (8): If Found, accept the data. Take ownership if Own is set; exclusive if (Excl & $\overline{\text{Shared}}$).
- Slaves (8): *pending miss*: If Found, accept the data (state S).
- L2/memory (8): If (MAccept & $\overline{\text{Found}}$), schedule the request to go to memory.

Read-exclusive (EX) / Upgrade (UP)

A read-exclusive proceeds as a read-shared, except that the line is invalidated from other caches, and piggyback reads are not allowed. Upgrades are the same, but there is no need to supply the data.

- Master (3): set Addr, Type (EX), and ID.
- Slaves (4): If the line is
  - *in cache*: Supply Data, and assert Found. Assert Excl if the line is exclusive, Own if it is owned. Invalidate the cache line.
  - *in WB*: Supply Data, and assert Found, Own. Also assert Excl if the line is exclusive in the writeback buffer. Cancel the writeback request.
  - *pending miss*: No action.
- L2/memory (4/5): Assert MAccept if the request could be accepted (i.e. the buffers are not full). If the line is
  - *in L2*: Supply Data if an L1 didn't indicate Found, and assert Found. Also assert Excl if the line is exclusive in the L2.
- Master (8): If Found, accept the data. Take ownership if Own is set; exclusive if Excl.
- L2/memory (8): If (MAccept & $\overline{\text{Excl}}$), buffer the request to go to memory. If $\overline{\text{Found}}$, then a ME reply will be needed to an EX request; if Found, or the request was UP, then a UA reply is used.

Writeback (WB)

For a writeback, it is the master which supplies the data rather than a slave.

- Master (3): set Addr and Type (WB).

- Slaves (4): If the line is in the cache or a pending miss: Assert Found and SRq
- L2/memory (4): Assert MAccept if the request could be accepted, or Found and SRq if the line is in the L2.
- Master (4): Supply Data; assert Own, and also Excl if the line is exclusive in the writeback buffer.
- Master (8): If Found or MAccept, the writeback is completed.
- Slaves (8):
  - *in cache*: If SGnt, change state from S to O.
  - *pending miss*: Accept the data, (state O if SGnt, otherwise state S).
- L2/memory (8): If (MAccept & $\overline{\text{Found}}$), write the line back to memory. If SGnt, take ownership in the L2.

Memory-originated (MS / ME / UA)

The memory's replies are similar to the cache-originated requests. The major difference is that the ID transmitted is that of the requester. If the line matches a pending read a cache will assert Found; for MS, if the ID does not match it will also assert Shared. The originally-requesting cache will accept the line exclusively if (Excl & $\overline{\text{Shared}}$); other caches accept it shared. These piggyback reads are not allowed for ME. The ID has to be transmitted on ME replies so that the original requester is guaranteed to complete its operation.

References

[AAC+94] Gail Alverson, Bob Alverson, David Callahan, Brian Koblenz, Allan Porterfield, and Burton Smith. Integrated support for heterogeneous parallelism. In Iannucci et al.[IGHS94], pages 253–283.

[AB86] James Archibald and Jean-Loup Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.

[AB95] Craig Anderson and Jean-Loup Baer. Two techniques for improving performance on bus-based multiprocessors. In *Proceedings of the International Symposium on High Performance Computer Architecture*, pages 256–275. IEEE, 1995.

[ABC+95] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Kenneth Mackenzie, and Donald Yeung. The MIT Alewife machine: Architecture and performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 2–13. ACM, June 1995.

[ACC+90] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. In *Proceedings of the 1990 International Conference on Supercomputing*, pages 1–6. ACM, June 1990.

[AD98] Haithim Akkary and Michael A. Driscoll. A dynamic multithreading processor. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, pages 226–236. ACM, 1998.

[AG94] George S. Almasi and Allan Gottlieb. *Highly Parallel Computing*. Benjamin/Cummings, second edition, 1994.

[AG95] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. Research report 95/7, DEC Western Research Laboratory, September 1995.

[AH93] Sarita V. Adve and Mark D. Hill. A unified formalization of four shared-memory models. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):613–624, June 1993.

[AKK] Gail Alverson, Simon Kahan, and Richard Korry. Processor management in the Tera MTA computer system. Technical report, Tera Computer Co.

[AKK+93] Anant Agarwal, John Kubiatowicz, David Kranz, Beng-Hong Lim, Donald Yeung, Godfrey D'Souza, and Mike Parkin. Sparcle: An evolutionary processor design for large-scale multiprocessors. *IEEE Micro*, pages 48–61, June 1993.

[AP98] Denis N. Antonioli and Markus Pilz. Analysis of the Java class file format. Technical Report 98.4, Dept. of Computer Science, University of Zurich, April 1998.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[BG97] A. J. C. Bik and D. B. Gannon. Automatically exploiting implicit parallelism in Java. *Concurrency, Practice and Experience*, 9(6):579–619, 1997.

[BGK96] Doug Burger, James R. Goodman, and Alain Kägi. Memory bandwidth limitations of future microprocessors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 78–89. ACM, May 1996.

[BGM+00] Luiz André Barroso, Kourosh Gharachorloo, Robert McNamara, Andreas Nowatzyk, Shaz Qadeer, Barton Sano, Scott Smith, Robert Stets, and Ben Verghese. Piranha: A scalable architecture based on single-chip multiprocessing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 282–293. ACM, June 2000.

[BW88] Jean-Loup Baer and Wen-Hann Wang. On the inclusion properties for multi-level cache hierarchies. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 73–80. IEEE, 1988.

[CDK+98] Andrew Chang, William J. Dally, Stephen W. Keckler, Nicholas P. Carter, and Whay S. Lee. The effect of explicitly parallel mechanisms in the Multi-ALU processor cluster pipeline. In *Proceedings of the 1998 International Conference on Computer Design*, pages 474–481. IEEE, 1998.

[CGL94] David Culler, Michial Gunter, and James Lee. Analysis of multithreaded microprocessors under multiprogramming. In Iannucci et al.[IGHS94], pages 351–371.

[CH95] Yong-Kim Chong and Kai Hwang. Performance analysis of four memory consistency models for multithreaded multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 6(10):1085–1099, October 1995.

[CJDM99] Vinodh Cuppu, Bruce Jacob, Brian Davis, and Trevor Mudge. A performance comparison of contemporary DRAM architectures. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 222–233. ACM, May 1999.

[CO98] Michael K. Chen and Kunle Olukotun. Exploiting method-level parallelism in single-threaded Java programs. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, pages 176–184. IEEE, 1998.

[Com99] Compaq Computer Corp. *Alpha 21264 Microprocessor Hardware Reference Manual*, July 1999.

[Cri97] Richard Crisp. Direct Rambus technology: The new main memory standard. *IEEE Micro*, 17(6):18–28, November/December 1997.

[CS98] David Culler and J. P. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1998.

[CSK+99] Robert S. Chappell, Jared Stark, Sangwook P. Kim, Steven K. Reinhardt, and Yale N. Patt. Simultaneous subordinate microthreading (SSMT). In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 186–195. ACM, May 1999.

[CW99a] Lucian Codrescu and D. Scott Wills. Architecture of the Atlas chip multiprocessor: Dynamically parallelizing irregular applications. In *Proceedings of the 1999 International Conference on Computer Design*, pages 428–435. IEEE, 1999.

[CW99b] Lucian Codrescu and D. Scott Wills. On dynamic speculative thread partitioning and the MEM-slicing algorithm. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, pages 40–46. IEEE, 1999.

[Die99] Keith Diefendorff. Power4 focuses on memory bandwidth. *Microprocessor Report*, 13(13), October 1999.

[Dig92] Digital Equipment Corp. *Alpha Architecture Handbook*. 1992.

[Dig97] Digital Semiconductor. *21164 Alpha Microprocessor Data Sheet*, February 1997.

[DM82] David R. Ditzel and H. R. McLellan. Register allocation for free: The C Machine stack cache. In *Proceedings of the International Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 48–56. ACM, March 1982.

[DSS95] Fredrik Dahlgren, Jonas Skeppstedt, and Per Stenström. Effectiveness of hardware-based and compiler-controlled snooping cache protocol extensions. In *Proceedings of the International Conference on High-Performance Computing*, pages 87–92. IEEE, December 1995.

[EEL+97] Susan J. Eggers, Joel S. Emer, Henry M. Levi, Jack L. Lo, Rebecca L. Stamm, and Dean M. Tullsen. Simultaneous multithreading: A platform for next-generation processors. *IEEE Micro*, 17(5):12–19, September/October 1997.

[EK89] Susan J. Eggers and Randy H. Katz. Evaluating the performance of four snooping cache coherency protocols. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 2–15. ACM, 1989.

[Fis83] J. A. Fisher. Very long instruction word architectures and the ELI-512. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 140–150. ACM, 1983.

[FLR98] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1998.

[Fur89] Stephen B. Furber. *VLSI RISC Architecture and Organization*. Marcel Dekker, 1989.

[GKW85] J. R. Gurd, C. C. Kirkham, and I. Watson. The Manchester prototype dataflow computer. *Communications of the ACM*, 28(1):34–52, January 1985.

[Gol96] Ulrich Golze. *VLSI Chip Design with the Hardware Description Language VERILOG: An Introduction Based on a Large RISC Processor Design*. Springer, 1996.

[HHS+00] Lance Hammond, Benedict A. Hubbert, Michael Siu, Manohar K. Prabhu, Michael Chen, and Kunle Olukotun. The Stanford Hydra CMP. *IEEE Micro*, 20(2):71–84, March/April 2000.

[HNO97] Lance Hammond, Basem A. Nayfeh, and Kunle Olukotun. A single-chip multiprocessor. *IEEE Computer*, 30(9):79–85, September 1997.

[HO98] Lance Hammond and Kunle Olukotun. Considerations in the design of Hydra: A multiprocessor-on-a-chip microarchitecture. Technical Report CSL-TR-98-749, Computer Science Department, Stanford University, February 1998.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[HP96] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, second edition, 1996.

[HS99] Sébastien Hily and André Seznec. Out-of-order execution may not be cost-effective on processors featuring simultaneous multithreading. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture*, Orlando, USA, January 1999. IEEE.

[IGHS94] R. A. Iannucci, Guang R. Gao, Robert H. Halstead, Jr., and Burton Smith, editors. *Multithreaded Computer Architecture: A Summary of the State of the Art*. Kluwer, 1994.

[Inm88] Inmos Ltd. *Transputer Reference Manual*. Prentice Hall, 1988.

[JBSS97] Quinn Jacobson, Steve Bennett, Nikhil Sharma, and James E. Smith. Control flow speculation in multiscalar processors. In *Proceedings of the 3rd International Symposium on High Performance Computer Architecture*. IEEE, February 1997.

[Kan89] Gerry Kane. *MIPS RISC Architecture*. Prentice Hall, 1989.

[Kat85] Manolis G. H. Katevenis. *Reduced Instruction Set Computer Architectures for VLSI*. MIT Press, 1985.

[KHM89] David A. Kranz, Robert H. Halstead, Jr., and Eric Mohr. Mul-T: A high-performance parallel lisp. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 81–90. ACM, June 1989.

[KL84] Robert M. Keller and Frank C. H. Lin. Simulated performance of a reduction-based multiprocessor. *IEEE Computer*, 17(7):70–82, July 1984.

[Koo89] Philip J. Koopman, Jr. *Stack Computers: The New Wave*. Ellis Horwood Ltd, Chichester, UK, 1989.

[KP98] Christoforos E. Kozyrakis and David A. Patterson. A new direction for computer architecture research. *IEEE Computer*, 31(11):24–32, November 1998.

[Kra98] Andreas Krall. Efficient Java VM Just-In-Time Compilation. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*. IEEE, October 1998.

[Kro81] David Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pages 81–87. IEEE, May 1981.

[KSL87] Robert M. Keller, Jon W. Slater, and Kevin T. Likes. Overview of Rediflow II development. In *Graph Reduction: Proceedings of a Workshop held in Santa Fé, New Mexico*, number 279 in Lecture Notes in Computer Science, pages 203–214. Springer, 1987.

[LGH92] James Laudon, Anoop Gupta, and Mark Horowitz. Architectural and implementation tradeoffs in the design of multiple-context processors. Technical Report CSL-TR-92-523, Computer Systems Laboratory, Stanford University, May 1992.

[Lop94] Lanfranco Lopriore. Stack cache memory for block-structured programs. *The Computer Journal*, 37(7):610–620, 1994.

[LPU97] Allen Leung, Krishna V. Palem, and Cristian Ungureanu. Runtime versus compile-time instruction scheduling in superscalar (RISC) processors: Performance and trade-off. *Journal of Parallel and Distributed Computing*, 45:13–28, 1997.

[LY96] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Sun Microsystems, September 1996.

[Mat97] Doug Matzke. Will physical scalability sabotage performance gains? *IEEE Computer*, 30(9):37–39, September 1997.

[MD96] A. Mikschl and W. Damm. MSparc: A multithreaded Sparc. In *Proceedings of EuroPar '96*, number 1124 in Lecture Notes in Computer Science. Springer, 1996.

[Mil00] Aleksandar Milenkovic. Achieving high performance in bus-based shared-memory multiprocessors. *IEEE Concurrency*, pages 36–44, July-September 2000.

[MKH91] Eric Mohr, David A. Kranz, and Robert H. Halstead, Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, July 1991.

[MO98] Harlan McGhan and Mike O'Connor. PicoJava: A direct execution engine for Java bytecode. *IEEE Computer*, 31(10):22–30, October 1998.

[Moo96] Simon W. Moore. *Multithreaded Processor Design*. Kluwer, 1996.

[MPE96] MPEG Software Simulation Group. Mpeg-2 encoder/decoder version 1.2. http://www.mpeg.org/MSSG, July 1996.

[Muc97] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann, 1997.

[NHO96] Basem A. Nayfeh, Lance Hammond, and Kunle Olukotun. Evaluation of design alternatives for a multiprocessor microprocessor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 66–77. ACM, May 1996.

[NPA92] R. S. Nikhil, G. M. Papadopoulos, and Arvind. *T: A multithreaded massively parallel architecture. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 156–167. ACM, May 1992.

[ONH+96] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. In *Proceedings of the 7th International Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 2–11, October 1996.

[PJ87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[PJS97] Subbarao Palacharla, Norman P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 206–218. ACM, June 1997.

[PP84] Mark S. Papamarcos and Janak H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pages 348–354. IEEE, 1984.

[QDT88] Donna Quammen, David K. DuBose, and Daniel Tabak. A RISC architecture for multitasking. In *Proceedings of the 21st Hawaii International Conference on System Sciences*, pages 230–237, 1988.

[QMT89] D. J. Quammen, D. R. Miller, and D. Tabak. Register window management for a real-time multitasking RISC. In *Proceedings of the 22nd Hawaii International Conference on System Sciences*, pages 135–142, 1989.

[Ram99] Rambus Inc. *Direct RDRAM 128/144 Mbit (256Kx16/18x32s) Preliminary Information*, May 1999.

[Ram00] Rambus Inc. *Direct Rambus Memory Controller (RMC2) Preliminary Information*, July 2000.

[SBCvE90] Rafael H. Saavedra-Barrera, David E. Culler, and Thorsten von Eicken. Analysis of multithreaded architectures for parallel computing. In *Proceedings of the 2nd Annual Symposium on Parallel Algorithms and Architectures*, pages 169–178. ACM, July 1990.

[SIA98] Semiconductor Industry Association. *International Technology Roadmap for Semiconductors 1998 Update*, 1998.

[SS86] Paul Sweazey and Alan Jay Smith. A class of compatible cache consistency protocols and their support by the IEEE Futurebus. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 414–423. ACM, 1986.

[SS95] Thomas Scholz and Michael Schäfers. An improved dynamic register array concept for high-performance RISC processors. In *Proceedings of the 28th Hawaii International Conference on System Sciences*, pages 181–190, 1995.

[Str] Stream benchmark home page. http://www.cs.virginia.edu/stream/.

[Sud00] Subramia Sudharsanan. MAJC-5200: A high performance microprocessor for multimedia computing. Technical report, Sun Microsystems, Inc., 2000.

[Sun97] Sun Microsystems. *picoJava I Data Sheet*, December 1997.

[Sun99] Sun Microsystems. MAJC architecture tutorial. White paper, 1999.

[Tan95] Andrew S. Tanenbaum. *Distributed Operating Systems*. Prentice Hall, 1995.

[TEE+96] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levi, Jack L. Lo, and Rebecca L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 191–202. ACM, May 1996.

[TEL95] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403. ACM, June 1995.

[TIS93] Tool Interface Standards Committee. *Tool Interface Standard (TIS) Portable Formats Specification*, October 1993. Version 1.1.

[TJS95] Marc Tremblay, Bill Joy, and Ken Shin. A three dimensional register file for superscalar processors. In *Proceedings of the 28th Hawaii International Conference on System Sciences*, pages 191–201, 1995.

[TLEL99] Dean M. Tullsen, Jack L. Lo, Susan J. Eggers, and Henry M. Levy. Supporting fine-grained synchronization on a simultaneous multithreading processor. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture*, January 1999.

[TPB+91] Kenneth R. Traub, Gregory M. Papadopoulos, Michael J. Beckerle, James E. Hicks, and Jonathan Young. Overview of the Monsoon project. In *Proceedings of the 1991 International Conference on Computer Design*, pages 150–155. IEEE, 1991.

[Tre99] Marc Tremblay. MAJC microprocessor architecture for Java computing. Presentation at HotChips '99, 1999.

[TTKS96] Masafumi Takahashi, Hiroyuki Takano, Emi Kaneko, and Seigo Suzuki. A shared-bus control mechanism and a cache coherence protocol for a high-performance on-chip multiprocessor. In *Proceedings of the 2nd International Symposium on High Performance Computer Architecture*, pages 314–322. IEEE, 1996.

[Ung87] David M. Ungar. *The Design and Evaluation of a High Performance Smalltalk System*. MIT Press, 1987.

[VRG98] N. Vijaykrishnan, N. Ranganathan, and R. Gadekarla. Object-oriented architectural support for a Java processor. In *Proceedings of ECOOP-98*, number 1445 in Lecture Notes in Computer Science, pages 330–354. Springer, 1998.

[W+] Tim Wilkinson et al. The Kaffe Java virtual machine. http://www.kaffe.org.

[Wal93] D. W. Wall. Limits of instruction-level parallelism. Research Report 93/6, DEC Western Research Laboratory, 1993.

[WEMW00] Greg Wright, Ahmed El-Mahdy, and Ian Watson. Dynamic Java threads on the Jamaica single-chip multiprocessor. In *Proceedings of the 2nd Workshop on Hardware Support for Objects and Microarchitectures for Java*, Austin, TX, September 2000. In conjunction with ICCD-2000.

[WG94] David L. Weaver and Tom Germond, editors. *The SPARC Architecture Manual: Version 9*. Prentice-Hall, 1994.

[Wil89] Ifor Williams. *Object-Based Memory Architecture*. PhD thesis, Dept. of Computer Science, University of Manchester, 1989.

[WM95] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, 23(1):20–24, March 1995.

[WWEM99] I. Watson, G. Wright, and A. El-Mahdy. VLSI architecture using lightweight threads (VAULT) - choosing the instruction set architecture. In *Proceedings of the Workshop on Hardware Support for Objects and Microarchitectures for Java*, pages 40–44, Austin, TX, October 1999. In conjunction with ICCD-99.