Technical Report

UCAM-CL-TR-358
ISSN 1476-2986

Number 358

Multithreaded processor design

Simon William Moore

February 1995

This report was also published as a book of the same title (Kluwer/Springer-Verlag, 1996, ISBN 0-7923-9718-5).

This technical report is based on a dissertation submitted October 1994 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Trinity Hall.

ISSN 1476-2986
DOI *https://doi.org/10.48456/tr-358*

Abstract

Multithreaded processors aim to improve upon both control-flow and data-flow processor models by forming some amalgam of the two. They combine sequential behaviour from the control-flow model with concurrent aspects from data-flow design.

Some multithreaded processor designs have added just a little concurrency to control-flow or limited sequential execution to data-flow. This thesis demonstrates that more significant benefits may be obtained by a more radical amalgamation of the two models. A data-driven microthread model is proposed, where a microthread is a short control-flow code sequence. To demonstrate the efficiency of this model, a suitable multithreaded processor, called Anaconda, is designed and evaluated.

Anaconda incorporates a scalable temporally predictable memory tree structure with distributed virtual address translation and memory protection. A temporally predictable cached direct-mapped matching store is provided to synchronise data to microthreads. Code is prefetched into an instruction cache before execution commences. Earliest-deadline-first or fixed-priority scheduling is supported via a novel hardware priority queue. Control-flow execution is performed by a modified Alpha 21064 styled pipeline which assists comparison with commercial processors.

Preface

Except where otherwise stated in the text, this dissertation is the result of my own work and is not the outcome of work done in collaboration.

This dissertation is not substantially the same as any I have submitted for a degree or diploma or any other qualification at any other university.

No part of this dissertation has already been, or is being currently submitted for any such degree, diploma or other qualification.

This dissertation does not exceed sixty thousand words, including tables, footnotes and bibliography, but excluding appendices, photographs and diagrams.

Publications

The paper entitled "Scalable Temporally Predictable Memory Structures" was presented, by the author of this thesis, at the 2nd IEEE Workshop on Real-Time Applications, Washington D.C., July, 1994. This paper contains the early ideas which form chapter 5 of this thesis.

Trademarks

Acknowledgements

I would like to thank my supervisor, Derek McAuley, for his guidance and encouragement. Eoin Hyden has been a great friend and colleague, offering sound advice and copious comments on several drafts of this dissertation. Brian Graham's skills in formal methods have been invaluable and his work, in appendix A, is suitably accredited. Thanks are also due to Timothy Roscoe for his lively comments on single address space operating systems and memory management techniques. I would also like to thank Alan Jones, Peter Robinson and David Wheeler for may thought-provoking discussions during the hardware discussion group meetings.

For reading and commenting on drafts of this dissertation, I am also indebted to Richard Black, Paul Byrne, Shaw Chuang, David Evers, Robin Fairbairns, Daniel Gordon and Johanna Stiebert.

The Computer Laboratory, and particularly the Systems Research Group, has proven to be a stimulating environment in which to work. Special thanks go to: Jean Bacon, Paul Barham, Ralph Becket, Simon Crosby, James Hall, Mark Hayter, Mike Hinchey, Andy Hopper, Naeem Khan, Ian Leslie, Ken Moody, Roger Needham, Arthur Norman, Ian Pratt, Martin Richards, Cormac Sreenan, Quentin Stafford-Fraser and Neil Wiseman.

In 1993 I spent a most enjoyable summer at Digital's Western Research Laboratory under the guidance of Jeremy Dion. My thanks go to them all for honing my knowledge of processor design and simulation techniques.

Many members of my college, Trinity Hall, have also provided encouragement and support. I would particularly like to thank: William Clocksin, Stephen Hodges, Mari Jones, David Moore, Andrew Pauza and Ishtla Sing. I am also indebted to Martin Atkins, Gary Morgan and David Pumfrey, formelly from my undergraduate days at the University of York, who continue to encourage my work. My parents, Rosalie and Peter, and my sister, Helen, have also been a tower of strength.

This work was supported by a studentship from the UK Science and Engineering Research Council.

Contents

List of Figures

Glossary

CMOS	Complementary Metal Oxide
CPU	Central Processing Unit
DMA	Direct Memory Access
DRAM	Dynamic Random Access Memory
EDF	Earliest Deadline First
FIFO	First In First Out
FP	Fixed Priority
HOL	Higher Order Logic
IEEE	Institute of Electrical and Electronic Engineers, Inc.
I/O	Input/Output
LIFO	Last In First Out
MIMD	Multiple Instruction Multiple Data
OS	Operating System
PC	Program Counter
PE	Processor Element
QOS	Quality Of Service
RAM	Random Access Memory
RPC	Remote Procedure Call
SIMD	Single Instruction Multiple Data
SISD	Single Instruction Single Data
SPEC	Standard Performance Evaluation Corporation
TLB	Translation Lookaside Buffer
TTL	Transistor Transistor Logic
VLIW	Very Long Instruction Word
VLSI	Very Large Scale Integration

Chapter 1

Introduction

Multithreaded processors aim to improve upon both control-flow and data-flow processor models by forming some amalgam of the two. They exhibit sequential behaviour from the control-flow model and concurrent execution due to data-flow influences. This dissertation demonstrates the efficiency of a novel multithreaded processor designed to execute data-driven microthreads, where a microthread is a short control-flow unit of code.

1.1 Hardware motivations

Whilst control-flow processors dominate the current computer market, they have fundamental problems, primarily due to the lack of concurrency supported by the hardware. Too little concurrency results in processor stalls when waiting for long latency operations. A particularly thorny problem results from faster processors being used to tackle larger problems which require larger memory. Unfortunately this prohibits memory latency scaling with processor performance. Temporal and spatial locality of memory accesses may be used to cache frequently used values. However, an increasing number of ever larger caches are required to maintain a statistically low access latency. Furthermore, caches introduce a huge temporal variance which is hard to predict. This is inadequate for real-time applications.

Data-flow processors offer instruction level concurrency which allows long latency operations to be tolerated. The lack of sequential ordering, however, makes multiple assignment, to the same memory location, difficult. Thus, memory reuse and input/output operations are complex. Furthermore, instruction level concurrency implies significant synchronisation overhead which can often be greater than the time to evaluate the instruction. Also, there is insufficient time to make sensible scheduling decisions so poor use is made of the processor resource.

1.2 Software motivations

A significant number of today's applications only require single threaded best effort performance. This is reflected in the synthetic benchmark applications which are often used to assess a computer's performance. However, with the increasing appearance of multimedia, and our reliance on hard real-time applications, temporal predictability is important. Unfortunately, commercial processor designers are heavily motivated by the current de facto standard benchmarks. Until industrially recognised multimedia and hard real-time benchmarks are established, processor design will not change direction. Fortunately, academia is not so short-sighted

Multithreaded languages, and suitable operating system support, are becoming increasingly common, particularly for writing large, user interactive, applications. When using commercial control-flow processors, significant software support must be provided for these applications. Also, more concurrency reduces the efficiency of the cached memory structure. On the other hand, a multithreaded processor should be able to take advantage of concurrency to tolerate various latencies, including memory access latency.

1.3 Aims

This dissertation aims to investigate the data-driven microthread model to assess whether it can successfully underpin a general purpose processor designed with future hardware and software requirements in mind.

1.4 Synopsis

Chapter 2 reviews the hardware technology limitations and software application requirements which bound and guide successful processor designs.

Chapter 3 presents a commentary on existing processor designs.

Chapter 4 looks at hardware scheduling issues which are paramount for efficient use of a multithreaded processor resource.

Chapter 5 discusses scalable memory structures, virtual addressing support and protection mechanisms.

Chapter 6 presents a novel multithreaded processor design called Anaconda.

Chapter 7 evaluates the Anaconda design.

Chapter 8 reviews the work contained in previous chapters, draws a number of conclusions and suggests areas for future work.

Chapter 2

Design Motivations

2.1 Introduction

This chapter discusses current and possible future trends in software and hardware to determine desirable characteristics which processor designs should exhibit.

2.2 Software motivations

All processors are *Turing complete* (i.e. they can perform all the operations that a Turing machine can) and so can, in theory, run any application. However, in practice we must take the run-time into account. A processor must support a range of primitives, from which applications may be constructed, which allow a desirable performance to be met. Different application areas suggest different primitives and optimisations; however, there is some commonality (see section 2.2.1).

Benchmarks are synthetic applications used to assess the various performance characteristics of computers, from integer and floating-point arithmetic to memory and I/O speed. The performance of operating system support operations, like interrupt latency and context switch time, may be assessed. The efficiency of the compiler and linker also play an important role. Although benchmarks only allow an estimate to be made of the execution characteristics for a specific application, they provide essential metrics for processor designs.

The majority of processors are designed to attain good performance with respect to current industrial benchmarks. Most of these benchmarks are designed to assess uniprocessor computers where a single application requires best effort performance (section 2.2.2). However, computer control of mechanical machines, and other hard real-time applications, require temporally predictable performance (section 2.2.3). Other application areas — like multi-media — require some temporal predictability, but not the temporal guarantees imposed by hard real-time systems; an alternative is to guarantee a certain minimum *quality of service* [35] (section 2.2.4).

As processors become cheaper, multiprocessor computers become more practical. The software implications are discussed in section 2.2.5 and new areas to benchmark are proposed.

2.2.1 General primitives to support software

It is commonly accepted that some or all of the following collection of functions are required for many applications. The processor may explicitly provide a primitive for a particular function, or provide some efficient combination of primitives.

- *integer arithmetic* — addition, subtraction, comparison, multiplication and division; although division may not be supported by a single primitive it is now usual to provide a multiplication primitive.
- *floating-point arithmetic* — it is now common to support floating-point arithmetic, particularly with conformance to the IEEE standard (or some workable subset thereof).
- *logical operations* — boolean functions: AND, OR, NOT etc. and bitwise shifting.
- *conditionals* — to allow data dependent decisions to be made upon which code is executed.
- *subroutines* — to allow sharing of subprograms/software functions without code duplication.
- *multiple assignment* — allow values to be written to a memory and for old values to be overwritten. The ordering of reads and writes to a given location is vital if memory is to be reused. Furthermore, it is a prerequisite for input and output operations.
- *indirect addressing* — indirect memory addressing modes are required to allow calculated addresses to be used without requiring self-modifying code.
- *virtual addressing* — to assist memory allocation and garbage collection and to support *virtual memory*, by allowing paging of memory to some other storage server (e.g. a disk or a distributed memory structure), if required.
- *protection* — to prevent undesirable interference between applications executing concurrently in case one or more of them misbehaves either due to malicious

intent or a bug. This is typically provided by a memory protection subsystem which prevents applications from accessing memory areas for which they have not been given permission. As a corollary to this, a *supervisor mode* (also called a *privileged mode* or *trusted mode*) must be provided so that the operating system can initialise protection areas, or *domains*, and assign them to applications.

- *exceptions* — a method which invokes software to take care of errors, like *division by zero*, without explicitly having to add instructions to test for an error at every possible place.

- *synchronisation* — of external events to appropriate handler code to be executed. One approach is to extend the exception mechanism so that an exception is taken whenever an external event occurs; known as an *interrupt*.

2.2.2 Best effort systems

A typical workstation is used for computational and user interactive applications which simply require a reasonably short runtime. If, for example, a calculation takes a little longer than usual or the screen is not redrawn at the same rate, then it is at worst an annoyance to the user. In other words, the user expects the computer to make the best effort it can to finish jobs as quickly as possible.

Whilst the computer's speed places an upper bound on the possible rate of execution of a single application, the operating system plays an important role in dividing processor, memory and I/O resources between applications running concurrently. These resources are typically allocated on a first come first served basis [14]. The processor resource is typically allocated on a priority basis where an application is given a base priority which may be increased if it is waiting for I/O. This is computationally simple and allows interactive applications, which spend most of their time waiting for user input, to obtain processor resources rapidly. The only hardware that is required is a timer to invoke a reschedule (known as *preemption*) so that one task does not monopolise the processor resource. From an efficiency point of view it is important that the processor supports a quick method for switching context. Memory and I/O resource allocation is typically supported by the operating system.

The SPEC[1] mark suite of benchmarks is, at the moment, widely used for evaluating workstations. Only total execution time is measured for each benchmark. Different benchmarks emphasise the performance characteristics of different combinations of the computer's constituent parts. For example, the `compress` benchmark tests file I/O (both hardware and operating system support), memory and integer arithmetic but not

[1]SPEC marks are produced by the non-profit making company *Standard Performance Evaluation Corporation* which is supported by manufacturers of control-flow processors.

floating-point arithmetic; where as `tomcatv` mainly tests memory and floating-point facilities.

These benchmarks have their deficiencies. For example, the efficiency of concurrent execution of several benchmarks is rarely addressed even though this may be more representative of a typical load. Concurrent execution has profound effects upon memory performance, particularly if there are several caches in the memory hierarchy (see section 5.3.1 for a discussion on caches). Also, if a translation look-aside buffer (TLB) is used for virtual address translation and memory protection, then its efficiency is also highly dependent upon concurrent activities.

Performance of user interactive applications is also not usually benchmarked, probably because such statistics are difficult to obtain. However, to a large extent, general processor, memory and I/O benchmarks give a reasonable indication of the likely interactive performance.

2.2.3 Hard real-time system requirements

Hard real-time systems require output to be produced by specified times in relation to input stimuli [63]. Furthermore, premature output may also be undesirable. For example, a car engine management system needs to cause ignition (via the spark plugs) at the correct time, neither early nor late. The real-world is inherently parallel; thus, in order that there is a clear, logical mapping between the application and its environment, software is often constructed as a set of parallel tasks or *threads* (an abbreviated form of *threads of control* which makes particular reference to the control-flow model of computing — see section 3.2).

Real-time applications must be assessed to ensure that not only their functional requirements are met, but also their temporal ones. This is assisted by the fact that most real-time applications have the whole computer system to themselves. Simple systems may have a natural periodicity which allows the scheduling of these threads to be determined off line, or *statically*. More complex systems with sporadic stimuli require run time, or *dynamic*, scheduling. Some of these systems may be scheduled using a preemptive fixed priority algorithm [12] which, as with the best effort scheduler, requires little extra hardware support. However, other applications require a more sophisticated scheduler, for example, one which schedules the task with the earliest deadline. An *earliest deadline first* (*EDF*) scheduler requires a priority queue to sort the deadlines of runnable tasks. This is computationally expensive, but if there are not too many tasks and scheduling does not need to occur too often, then a software implementation is satisfactory. However, this may not be the case and hardware support may be necessary (see chapter 4 for further details).

No matter how good the scheduler is, there must be sufficient processor resources for a schedule to be possible. Missing a hard real-time deadline can be catastrophic, so it is important that the worst case processor time required for each thread can be assessed. Furthermore, in a dynamically scheduled application it is vital that the processing time of threads is independent of the order in which they are scheduled. Unfortunately, as will be seen in section 3.2 on control-flow machines, this is currently far from the case and is becoming increasingly bad. One may deduce from this that most processor designers are motivated by the need to produce fast processors in the best effort sense, rather than making them temporally predictable and fast.

One reason for this may be that there are few standard real-time benchmarks; most real-time applications programmers opt for their own benchmarks to suit their specialist application. This is of little use to the processor designer. The Hartstone benchmark [67] is an exception to this but it is primarily designed for assessing Ada compilers and their associated run time support system rather than processor characteristics. Experimental results from using Hartstone indicate that a high resolution hardware real-time clock is of great importance. However, little more can be deduced.

2.2.4 Multi-media and quality of service requirements

Multi-media, and in particular continuous media like audio and video, require end-to-end guarantees between the source and sink. This encompasses processor, memory, disk and/or network resources. In order to reduce resource usage, compression techniques are employed, often in a layered form such that low quality, and thus low bandwidth, information is sent and processed first followed by several additional layers of information to improve the quality. This potentially allows an overloaded resource to drop the higher quality layers first if overloading occurs, thereby providing a degraded service rather than no service at all. From a processor scheduling viewpoint, applications require a minimum proportion of the processor resource which may be specified as a desired *quality of service*[2] [35]. If processor resources are left over then applications may receive extra resources to allow higher layers to be decoded.

As with hard real-time, predictable processor performance is essential. For multimedia systems, temporal predictability allows accurate calculation of the minimum amount of processor time required to deliver a desired minimum quality of service. I/O performance is also critical.

[2]Quality of service may also be used to schedule network resources to deliver an end-to-end quality of service [35].

2.2.5 Multi-processor computers

The last three sections have concentrated on applications running on uniprocessors. Transferring these to multiprocessors poses many problems for the software engineer. These include the reworking of algorithms to allow partitioning of the processor load together with problems of efficient movement of data and associated synchronisation to ensure correct ordering of updates.

Obviously the processor design must provide efficient means to move data to and from other processors. Message passing is one approach; another is to share memory using per-processor caches for recently used data. The two methods are equivalent in that any algorithm could be rewritten to use either method [40]. Which performs best is largely application specific and so it is advantageous if both models can be supported. For shared memory some form of cache coherency mechanism is required, e.g. using a directory [42]. Message passing requires low overhead synchronisation of messages to threads.

Currently, general purpose benchmarks for parallel computers are difficult to write because there is no accepted way of coding parallel algorithms so that they run reasonably efficiently on a wide range of parallel computers. Consequently, the current practice is to hand tune code for particular parallel computers resulting in benchmarks reflecting the programmer's ingenuity as well as processor performance and compiler technology. This obviously makes comparisons problematic.

2.3 Hardware motivations

Computer systems employ a variety of technologies from mechanical and ferromagnetics to electronics and optics. Consequently there are a huge number of implementation technology issues, the details of which are beyond the scope of this dissertation. However, the general problems, relating to signal transmission and active devices, are pertinent to this discussion (sections 2.3.1 and 2.3.2 respectively). Basic circuit techniques are also briefly reviewed in section 2.3.3 with a slightly more detailed discussion of self-timed circuits due to their comparative novelty.

2.3.1 Signal transmission limits

The transmission of signals is limited by the speed of light ($c \approx 2.99725 \times 10^8$ ms^{-1}). This is a fundamental limiting factor which is becoming particularly significant as circuit switching frequency moves beyond 1 GHz, since at this speed even light only travels approximately 300 mm in one period (1 ns).

Electrical pulses in a wave guide are somewhat slower at:

$$v_p = \frac{c}{\sqrt{\mu_r \epsilon_r}}$$

where v_p is the velocity of propagation which is dependent upon the speed of light (c), the permeability (μ_r) and the permittivity, or dielectric constant, (ϵ_r) relative to free space. By definition, μ_r and ϵ_r must be > 1 but careful design can keep them near 1. However, capacitance, self and mutual inductance only make matters worse.

The power required to source a signal is dependent upon transmission line attenuation and the fanout. A signal emanating from a point source (see figure 2.1a) must provide sufficient power to each transmission line. For a bus (see figure 2.1b) power is divided at each junction so the source signal must be strong enough.

Providing a sufficiently large amplifier for a large fanout is problematic. Since amplifier gain is finite, a large amplifier requires a proportionately large signal. Consequently, for very large fanouts, a cascade of amplifiers may be required (figure 2.1c). However, large amplifiers tend to be slow. Alternatively, a tree of amplifiers may be used (figure 2.1d) but the speed of each amplifier tends to be different due to manu-

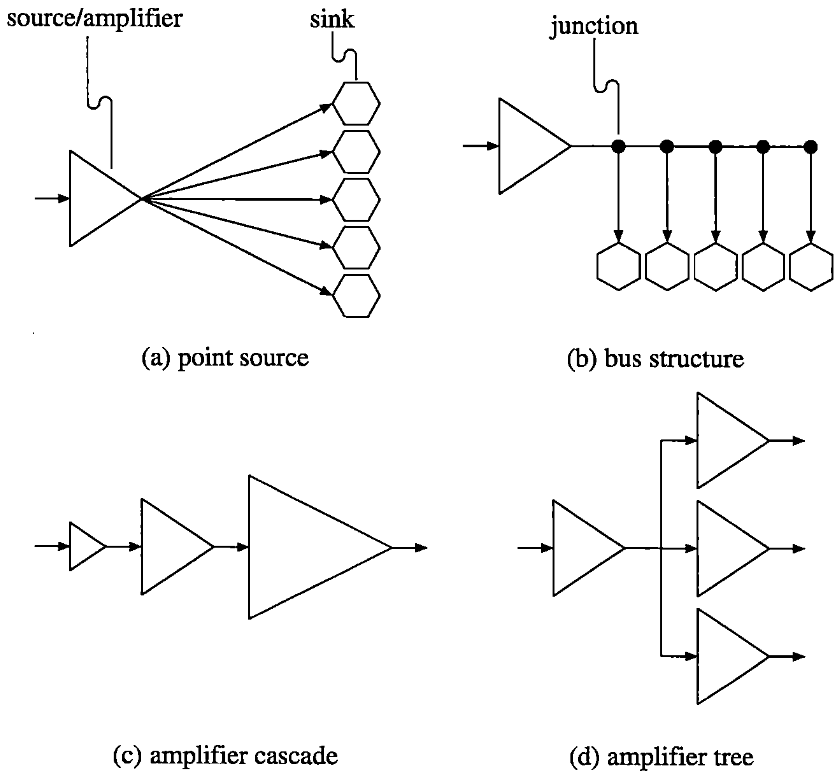


Figure 2.1: Signal fanout and amplification

facturing tolerances, resulting in temporal skew. This is obviously undesirable for synchronisation signals, or *clocks*.

To conclude, it is advantageous if transmission lines can be kept short with little fanout.

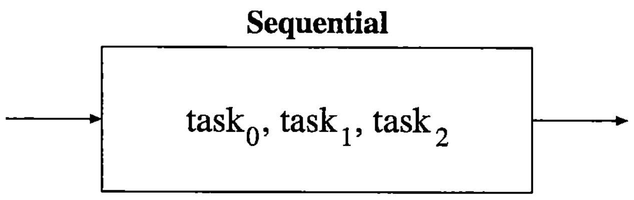
2.3.2 Active component limits

The switching, or *active*, components (for amplification and boolean function implementation etc.) which are widely used today, are often based upon *complementary metal oxide* (CMOS) *very large scale integration* (VLSI) technology [47]. For example, the Digital's Alpha series of processors are already available in CMOS with a feature size of 0.75μm and operating at 200 MHz [62]. Whilst feature sizes remain larger than 0.5μm, every development enabling the feature size to be reduced is rewarded by significantly improved performance. This is due to capacitance and distance being reduced whilst voltage remains at 5V to be compatible with other TTL circuits. For a fixed voltage $gate\ delay \propto (feature\ size)^2$. However, at less than 0.5μm the voltage has to be dropped below the usual 5V to prevent signals tunnelling through the ever diminishing insulation layers. This reduces the potential performance gain to $gate\ delay \propto feature\ size$ because $voltage \propto feature\ size$. Therefore, it is anticipated that the density of CMOS circuits will increase but the performance will not improve as dramatically [38].

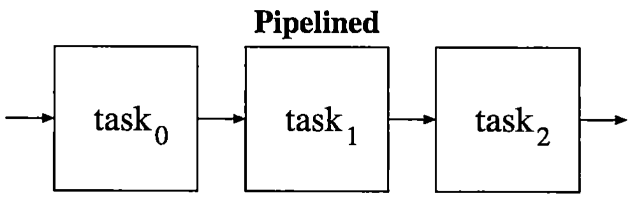
It would be foolish to attempt to predict which implementation technology will be used in the future. If room temperature superconductors become available, then quantum effect devices may be the future [31]. Alternatively, the future may be in nanomechanical machines [23]. However, whichever technology is used it is highly probable that active components will have a spatial locality which in turn enforces a spacial locality for information and computation. Furthermore, movement of information takes time and typically consumes power. Therefore, it is advantageous if efficient use can be made of local storage.

2.3.3 Circuit techniques

Functions to perform data manipulation range from bitwise boolean operations, integer addition or subtraction through integer multiplication and division to floating-point arithmetic. The more complex functions take a relatively long time to compute (i.e. the *latency* of the operation is long). However, to enable the *frequency* of data transfer to match the latency of simple operations, the more complex operations may be broken down into sequential stages (see figure 2.2). Latches are placed between stages to store the intermediate results. The resulting structure (a *pipeline*) allows computation



maximum latency = $\ell_s = \sum_{i=0}^{2} maxtime(task_i)$
maximum frequency = $\frac{1}{\ell_s}$



maximum latency = $\ell_p = 3 \times \text{MAX}_{i=0}^{2} maxtime(task_i)$
maximum frequency = $\frac{3}{\ell_p}$

Figure 2.2: Sequential vs pipelined — trading latency for frequency

and information flow to be localised. This scales well with decreasing discrete component size and increasing components on a chip. In later sections example processor and memory pipelines will demonstrate more complex structures in which some stages branch off into two or more pipelines (see section 3.2.3 and chapter 5 respectively).

Currently the most popular technique for controlling information flow between stages of pipelines is to use one or more global clock signals to trigger the latches to transfer the information on their inputs to their outputs. The clock rate is determined by the slowest stage of the pipeline, for example the integer addition/subtraction stage where the carry has to be propagated. However, transmitting a global clock so that all the latches update simultaneously is impossible because of the transmission time and the large capacitances which the global wiring introduces. The timing difference due to delay is known as *clock skew*. In practice, using today's technology, it is still possible to keep clock skew to a tolerable level across a chip but as chip sizes and clock frequencies increase the problem is going to become worse. These problems can already be witnessed at the circuit board level where the processor is typically clocked at 2, 4 or even 8 times the board clock rate because it is impractical to broadcast the higher speed clock across the whole circuit board. When high speed circuit board clocks are used (e.g. RAM-BUS [24]) the clock follows a single path along the same route as the data to ensure equal delays. Alternatively, circuits may be clocked from different sources but this introduces synchronisation problems and overheads. Similar techniques can be applied to signals across a chip.

Global clocks limit performance because they are set at a rate which always allows the slowest stage of the pipeline to complete its calculation correctly. Furthermore, by the time the design has taken into account manufacturing tolerances, and that silicon performance is temperature dependent, a significant margin of error (e.g. 20%) has to be allowed.

An alternative to global clocks is to use localised clocking to determine when pipeline stages have completed. This is known as *micropipelines* [65] which are a form of nonclocked, or *self-timed*, circuit. Local clocks may be derived from critical signal paths (e.g. the carry propagation of an adder) and by using delay lines which may be placed close to the associated function. This allows a stage to complete early, for example, if the carry propagation takes a fast path. Furthermore, temperature variations can be allowed for to some extent. However, current practice has shown that often a 20 to 30% margin of error has to be catered for, due to manufacturing variance [27], resulting in delays which have to be longer than one would wish, in order to avoid timing problems.

Another self-timed technique uses two bits to represent every binary digit of information together with a handshaking signal (see figure 2.3) which is known as *dual rail encoding* [18]. Functions are evaluated in a two stage process. Firstly, the function's inputs are cleared (each pair set to 00) until the outputs are all cleared. Then, data is placed on the inputs and when the outputs contain valid data (i.e. not 00 but 01 or 10) this value may be latched. Thus, the completion signal is encoded with the data. This makes dual rail encoding a very safe way of building self-timed circuits, because correct operation is independent of manufacturing tolerances and temperature effects. However, design methods have to be different from their clocked counterparts. For example, pipeline designs have to allow clear signals to propagate in the wake of data signals. An interesting example of this is a divider design which employs five pipeline stages arranged in a circle which allows enough time for clears to have been performed before the data comes full circle [71]. The primary disadvantage of this technique is that more logic is

A^1	A^0	Meaning
0	0	cleared
0	1	logical 0
1	0	logical 1
1	1	illegal

Figure 2.3: Dual rail encoding of binary numbers

required than for a clocked implementation but with increasing chip densities and the problems with fast clocks, dual rail encoding is looking attractive.

Dual rail encoding, micropipelines and variations on the clocking theme are active research areas and it is unclear which will be the most successful. However, it is possible to draw the conclusion that whichever method is employed, some form of localised signalling or clocking domains is essential.

2.4 Summary of hardware and software requirements

The section on software motivations has identified the basic needs to support most applications' functional requirements (see list of desirable processor requirements on the next page). Then more specific application areas were identified and categorised according to their temporal requirements. These categories are: best effort, hard real-time and multimedia or quality of service based. An overview of benchmarks was presented because they represent the execution requirements which motivate processor designs. However, to date the majority of standard benchmarks are representative of best effort applications. Inevitably, this results in processors which are tailored to provide good best-effort performance even when this is to the detriment of supporting other categories of application. Consequently there is a vicious circle where the software designers develop languages and coding methods which yield good performance using today's processors, leaving commercial processor designers to optimise current designs.

The section on hardware motivations introduced two fundamental problems with designing computers on large and increasingly dense VLSI technology, namely signal transmission speed limitations and the fact that the popular submicron CMOS technology does not become significantly faster as density increases beyond a feature size of 0.5μm. The problems of transmitting higher frequency global clocks were also addressed, concluding that control signals benefit from being localised. In the future some form of self-timed circuit technique may provide the required characteristics. However, it was also noted that there are many possible technologies on the horizon which will inevitably introduce different design parameters and techniques.

Desirable Processor Attributes

- functional primitives
  - arithmetic (both integer and floating-point) and logic primitives
  - conditional and subroutine primitives
  - synchronisation primitives
- memory structure and access mechanisms
  - local storage of intermediate results (to avoid over use of the main memory)
  - indirect addressing
  - virtual addressing
  - multiple assignment (and support for sequential algorithms)
  - scalable interconnect (to avoid signal transmission problems)
- protection mechanisms
  - memory protection
  - supervisor mode (to allow protection domains to be set up)
- concurrency (processor sharing)
  - including support for a sensible scheduling policy
- parallelism (multiple processors and multiple pipelines)
  - efficient synchronisation of internal and external events
- temporal predictability
- a distributed real-time clock at the highest practical frequency
- localised signalling (avoid using global signalling like processor frequency clocks)

Chapter 3

Current Processor Models

3.1 Introduction

This chapter is a commentary on current processor models with references to more in-depth material. To begin with, the control-flow processor model (section 3.2) is presented because it dominates today's computer market.

Control-flow processors primarily exhibit sequential behaviour. At the other extreme, there are data-flow processors (section 3.3) which can exploit instruction level concurrency. Both of these models have their advantages and disadvantages but one tends to have strengths where the other has weaknesses. Various amalgams of these ideas, often referred to as *multithreaded* processors, exhibit improved characteristics (section 3.4).

3.2 Control-flow processors

The fundamental control-flow processor model was originally proposed by Eckert, Mauchly and von Neumann in 1944 and was based upon the Analytical Engine design proposed by Babbage in 1838 [69]. Programs are constructed as a linear list of *order codes*, or *instructions*, which control the processing of data. The list is traversed using a pointer — the *program counter*. Decisions are supported through changing the flow of control by conditionally modifying the program counter.

Fundamentally, this model still underpins the ubiquitous control-flow machine of today. However, many improvements in functionality have been made from the support of reusable subprograms, or *subroutines* using the Wheeler jump [68], to interrupts (reputably first introduced on the UNIVAC 1103 [60]) and beyond. There have been many performance improvements which may be grouped into:

- memory structure and data locality (e.g. the use of a register file and cache)

- instruction coding
- instruction parallelism
- data parallelism

Other issues are less performance critical on today's control-flow machines, when assessed using many industrially recognised benchmarks, which include:

- concurrency and synchronisation primitives
- protection mechanisms

The following sections briefly review each of these areas. More detail may be found in the plethora of computer architecture texts, good examples of which are [32, 54, 60].

3.2.1 Memory structure and data locality

A memory is required to store programs and data. To an application the memory appears as a linear collection of bits, bytes or words which may be indexed by an *address*. However, virtual addressing may be employed to simplify memory allocation by allowing blocks, or *pages*, of virtual addresses to be mapped on to any physical page. If there are more active virtual pages than physical pages in main memory, then backing store may be used as an overflow memory, pages being *paged* between backing store and main memory on demand. Each application may have an independent set of virtual addresses. Alternatively applications may reside in a single virtual address space which is particularly useful if data and code are to be shared. 64 bit address systems are well suited to the latter approach because the large virtual address space is unlikely to be totally consumed.

Intermediate results are usually held in a very small multi-ported memory which is local to the processor (a *register file*). This allows several values to be read and written simultaneously and at high speed which is not possible with main memory due to its large size and distance from the processor (see section 5.2). Instead of a register file, a stack model may be presented. To allow efficient access to the most frequently used values at the top of the stack, they must be cached in some small, high speed multi-ported memory (e.g. Crisp [7], which is the forerunner to AT&T's Hobbit [5]). Both register file and stack based approaches usually use some short form of addressing to reduce instruction size; in the case of a register an index into the register file is used and for stacks a short offset from the top of the stack suffices.

To bridge the performance gap between main memory and the register file, one or more levels of caching may be used to store intermediate results. This relies on the *principle of locality* which is that a few addresses are accessed most of the time (*temporal locality*) and that many of the accesses are within a few groups of addresses (*spatial locality*). When caches were first introduced to microprocessors they were often shared between the instruction and data fetch sections. More recent systems have a separate cache for instructions and data (e.g. the Motorola MC88000 [49] and Digital's Alpha [20]). However, a recent fashion is to have one large wide unified cache and a short instruction buffer which is capable of reading several instructions from the cache at once (e.g. the PowerPC 601 [62]). Since data accesses typically account for 40% of instructions executed (at least for RISC processors [54]), the instruction and data accesses may be successfully interleaved with few occasions when the processor is starved of instructions. However, when the density of chips increases we are likely to see the re-emergence of separate caches because the single cache cannot be made much larger and still be sufficiently fast.

Memory for multi-processor systems is frequently based upon caching data which is currently being accessed and/or modified. Access mechanisms must be provided to ensure data integrity is maintained when multiple processors compete over writing to the same memory area. One approach is to broadcast writes to all processors so that they may update their cache. A specialisation of this, called *snooping*, broadcasts the information on a bus. However, broadcasting information from many sources becomes increasingly costly as the number of sources increases. Alternatively, a directory may be used to record which processors have a read only copy of information so that invalidation of the information may be achieved by multicasting to the appropriate processors. For example, the Stanford Dash multiprocessor [42] supports a distributed directory structure for shared memory between clusters of processors, and snooping within clusters.

3.2.2 Instruction coding

Typically, instructions consist of one or more operations (specified by *op-codes*[1]) and zero or more operands[2] to specify data source and destination. The type of the operands may be specified as part of the op-code, some separate field or by the data itself in the case of typed memory (see sections 6.6 and 6.7). Some operands may be implicit in the op-code so do not need to be specified. For example, *set interrupt mask bit* or *skip next instruction if the carry is set* instructions do not need to specify any operands. Operands may be in terms of an index into a register file, an index into a stack or a memory address, where the address may be a constant or be calculated using registers or a stack.

[1]Although an op-code is unnecessary on a single instruction computer, e.g. r-move [48].

[2]Single operand instructions being referred to as *monadic*, double operand instructions as *dyadic* and triple operand instructions as *triadic*, etc.

When main memory was expensive it was advantageous for instructions to be as compact and full functioned as possible. This lead to the design of a breed of *complex instruction set computers* (*CISCs*) which relied upon variable length instructions for compactness, where each instruction could perform some very complex operation. For example, VAXs had a `POLY` instruction for evaluating polynomials [43].

Stack based machines also require a variable length instruction because operands are indexed off the stack, the index often being short but sometimes very long.

Memory became cheaper and it was realised that high level language compilers produced code which only made frequent use of a subset of the CISC instructions. This prompted the design of *reduced instruction set computers* (*RISC*) where the most frequently used, and also the indispensable, instructions were supported and made as fast as possible. The infrequent operations were synthesised from a combination of the frequent instructions. In order to make these simple instructions execute as fast as possible, simple decoding was desirable. Therefore, RISC instructions were fixed length and came in a few basic forms. Fixed length instructions mean that multiple main memory operands were impractical because they would make the instruction length prohibitively long (assuming reasonable offsets need to be specified). This resulted in the *load/store* approach where the only instructions which can access memory are the dyadic `load` and `store`, triadic data manipulation instructions being register-to-register.

3.2.3 Instruction parallelism

Executing an instruction takes some or all of the following steps: instruction fetch, instruction decode, operand fetch, execution/calculation, memory access, register file write. Some of these steps may be performed in parallel but many have interdependencies (e.g. instruction fetch must occur before it is known which operands are required) which imposes some ordering. However, steps for different instructions may be overlapped using a pipeline. For example, whilst operands are being fetched the next instruction may be fetched. Pipelines may be a simple linear structure but more often subpipelines are used to process instructions which require more complex processing. For example, floating point and integer multiply and divide operations require a longer execution time. This leads on to parallel execution of integer and floating point operations.

Provided several instructions may be fetched and their interdependencies resolved, it is possible to simultaneously issue multiple instructions to several integer and floating point pipelines. Such an execution structure is referred to as *superscalar*. An alternative to the superscalar approach, where the grouping of instructions for parallel execution is dynamic, is to perform the grouping statically to form a *very long instruc-*

tion word (*VLIW*). For example, the Multiflow TRACE 14/300 VLIW computer [59] with a 512 bit instruction format which takes two cycles to execute but can issue up to 8 integer operations, 4 floating-point operations and one branch operation concurrently.

An alternative to VLIW is to specify the data movements between ALUs, register files, etc., rather than specifying which operations need to be performed. This technique is know as *transport triggered* [17, 16] because data arriving at, say, an ALU triggers some function evaluation to be performed; the result of which is placed into an output buffer or FIFO until it is moved by another transport operation. This idea is very similar to microcode [60] which is composed of instructions with a fairly verbose encoding of all the hardware control signals required on a cycle by cycle basis.

As we move from the high-level CISC, RISC and superscalar instructions through to VLIW, transport triggered and low-level microcode instructions, more potential parallelism may be elicited. However, the encodings become more verbose which often results in poor memory usage. Furthermore, the *programmer's model*[3] becomes more complex as we move from the high-level CISC to the low-level microcode. As the programmer's model becomes more processor implementation specific, the code becomes less portable. However, lower level instructions which are tailored to a particular processor implementation (e.g. transport triggered) can be simpler to execute and can allow more parallelism to be elicited. One solution to the problem of portability of code is to compile to some intermediate form (either completely general or processor family specific) which may then be converted into a highly optimised low-level processor specific form.

Another form of instruction parallelism is exhibited by multi-processor computers, often called multiple-instruction, multiple-data (*MIMD*) after Flynn's classification [25]. Whilst, for example, superscalar systems offer instruction level parallelism for a single control-flow program, MIMDs offer a coarse level of parallelism with data communication, either by shared memory or message passing. Of course superscalar and MIMD approaches may be combined.

3.2.4 Data parallelism

Data parallelism is often achieved as a consequence of instruction parallelism. Flynn's classification [25] also identifies *single instruction, multiple data* (*SIMD*) computers (e.g. CLIP7A [26], DAP [53] and CM2 [33]). They tend to be used for specialist tasks, like graphics and finite element analysis, which can easily be split into many identical parallel tasks with little data dependency. Vector processing may be considered as a subset of the SIMD technique (e.g. Cray-1 [34]).

[3]The programmer's model is one which is sufficient to enable working code to be written. A more detailed model may be required to write efficient code.

3.2.5 Concurrency and synchronisation primitives

Synchronisation of data between instructions is achieved by the sequential ordering imposed by the control-flow model. This makes multiple assignment and access to I/O relatively easy. Whilst superscalar systems may perform instructions out of order they may only do so when a change of ordering will not affect the result.

Synchronisation of data coming in from an input device[4] is commonly achieved by sending the processor an *interrupt* signal which causes the thread of control to be suspended and forces execution to start at a particular address — usually the start of an interrupt handler routine. Concurrency is also achieved using this mechanism by using a timer to interrupt the processor at a regular period. The software interrupt handler determines where the interrupt came from and restarts a thread based upon a scheduling decision. It should be noted that *context switching* (changing threads) and scheduling decisions are typically performed in software.

3.2.6 Memory protection mechanisms

A memory access protection mechanism is provided by most processors to prevent separate applications from accidently or maliciously modifying or reading another application's data. This is extended to protect I/O devices which are often memory mapped. Protection may be at several levels and at a varying granularity; for example, capabilities [70] and rings [58]. However, complex protection mechanisms do incur costs, both in terms of execution time and processor complexity.

Multi-level and fine grained protection is typically used for intra-application protection as well as inter-application protection. However, modern high level language compilers can statically check many intra-application memory accesses. Furthermore, inter-application protection is often sufficient at the page level and may efficiently be combined with the virtual to physical address translation mechanism. Thus, page based protection is the norm.

Setting up a simple page based memory protection mechanism requires a *supervisor mode* (sometimes called a *trusted mode* or *privileged mode*), where the protection is turned off. Usually only a trusted part of the operating system (or *trusted kernel*) is allowed to obtain the supervisor privilege. Entering supervisor mode usually only occurs when a hardware interrupt, or its software counterpart (a *trap* or *call_pal*[5]), occurs and usually the code which is invoked is the trusted part of the operating system kernel. Thus, supervisor mode cannot be obtained subversively, which completes the memory protection system.

[4]Arriving data may simple be placed in an input queue which needs to be read by software, or it may be transferred directly to the main memory (*direct memory access* or *DMA*).

[5]*call_pal* is the Alpha version of a software trap which will be referred to again in chapter 6.

3.2.7 Assessment of the control-flow model

Control-flow processors completely dominate the computer industry and so many refinements have been made to improve performance. Electronics technology has allowed micro-processors to become more complex so that they can embody many of the desirable features developed for past mainframe computers. However, many of the architectural refinements, like the superscalar technique, have limited application. For example, the instruction level parallelism extracted by the superscalar approach is limited by data interdependencies which at best is limited by the number of registers.

The MIMD approach may be used to attain more parallelism but is limited by synchronisation and transfer overheads associated with sharing data. In particular, application level synchronisation primitives are not usually supported by the hardware. Furthermore, for many applications it is difficult to efficiently divide the program into separate cause grained tasks, or *threads of control.*

The inherently single threaded nature of control-flow machines means that the latency of cache misses tends to stall execution due to data dependencies. As memories become larger, and control-flow processors become faster, there is going to be an increasing reliance on good cache performance. Unfortunately, this means that the temporal characteristics of systems are going to become increasingly nondeterministic.

Despite the problems of the control-flow model it will no doubt continue to be refined and be used for a wide range of applications which require best effort performance and limited parallelism. The dominance of the control-flow model means than any serious move towards an alternative paradigm must provide some backward compatability.

3.3 Data-flow processors

Whilst control-flow programs explicitly define the order of execution which ensures that data is manipulated in the required order, data-flow programs specify the data-dependencies and allow the processor to choose one of the possible orderings. Data-dependencies may be represented as a directed graph where instructions form the nodes and the data-dependencies form the arcs (see figure 3.1). Any instruction which has received its operands may be executed, thereby allowing parallel execution. Furthermore, logically separate data-flow routines run at an instruction level of concurrency which is coordinated by a hardware scheduler.

The next sections present a brief overview of different data-flow paradigms. Then the preferred paradigm, tagged-token data-flow, is assessed.

Example function: f(a,b,c) := a.b + a/c

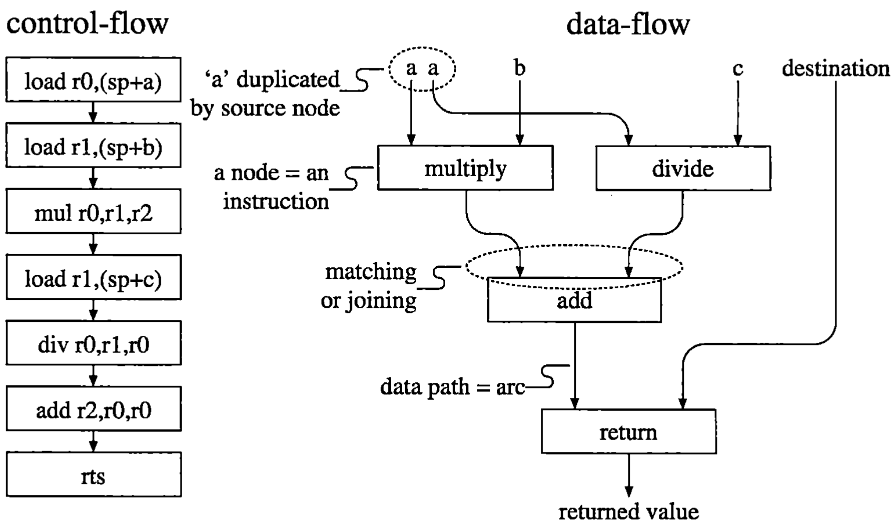


Figure 3.1: An example of control-flow and data-flow program structure

3.3.1 Static data-flow

With static data-flow [19] there is at most one *token* (datum) on an *arc* — path between one data-flow instruction and the next. The values on the arcs, or *operands*, are stored with the instruction. To ensure operands are not overwritten before they are used there are backward signal arcs which inform previous instructions when the instruction has been executed (i.e. when the operands have been used and the destinations have been written to). Figure 3.2 presents an example instruction format.

op-code	op1	(op2)	dest1 + dc1	(dest2 + dc2)	sig1	(sig2)

where	() indicates optional parameters
	op-code is the instruction identifier
	op1 and op2 are the spaces for operands (op2 missing for monadic operations)
	dest1 and dest2 are the destinations (dest2 being optional)
	dc1 and dc2 are destination clear flags (initially clear)
	sig1 and sig2 are the signal destinations (handshaking arcs)

Figure 3.2: Example static data-flow instruction format

There are problems with static data-flow:

- Shared functions are difficult to implement because mutual exclusion must be enforced upon writing all of the operands to the function. This severely limits concurrency and is often inefficiently solved by replicating functions either statically or dynamically.
- The backward signal arcs double the number of tokens to be matched.

3.3.2 Coloured dynamic data-flow

With dynamic data-flow there may be many tokens per arc. In the coloured data-flow paradigm associated tokens are given an unique identifier, or *colour*. This allows functions to be represented because each invocation of a function is given a unique colour. Only operands with the same colour may be matched in dyadic operations. An example of coloured data-flow is the Manchester prototype [30]. The RMIT hybrid [1] also utilises coloured data-flow to encapsulate functions but supports static data-flow within functions.

The main problems are:

- Matching colours is expensive and temporally unpredictable — often implemented using hashing.
- Uncontrolled fan-out can cause too many concurrent parts to be initiated resulting in matching store overflow.

3.3.3 Tagged token dynamic data-flow

An alternative paradigm (used for the MIT Monsoon machine [51]) is to remove operand storage from the instruction and to place it in a data page, or *activation frame*. Activation frames effectively store the context of a function. This may be implemented using a conventional flat memory store with the addition of a *presence* bit (an empty/full flag) for each word in memory. The advantage is that each instantiation of a function has its own separate activation frame which makes matching of operands easy:

- *instantiating a function* — obtain an unused activation frame and set all of the flags to *empty*.
- *matching a token to a monadic instruction* — the token's *statement pointer* (see figure 3.3) is used to lookup the instruction to be executed which forms an executable packet to be queued for execution.

- *matching a token to a dyadic instruction* — the token's *statement pointer* is used to look up the instruction to be executed. The instruction's *r* value (see figure 3.3) is then added to the token's *context pointer* value to form an address in the activation frame (i.e. *r* forms an offset into the activation frame). If there is a value in the activation frame then that value is read and matched with the token's *data* value which is sent, with the instruction information, to be executed and the activation frame location is set to *empty*. If there was not a value in the activation frame then the token's *data* value is stored in the activation frame to wait for its partner and the presence flag is set to *full*.

The matching scheme may be extended (e.g. the EM4 machine [57]) to allow intermediate results to be held in a register file, rather than being passed via a matching store, whilst instructions are executed in a control-flow manner. In the EM4 implementation the control-flow sections are limited by the number of input values into one instruction (dyadic or monadic) followed by a linear sequence of operations fetching their operands from the register file (i.e. the control-flow segments are limited to at most two input parameters).

Example instruction format

op-code	(r)	dest1	(dest2)

where () indicates optional parameters
op-code is the instruction identifier
r is the activation frame offset number for dyadic operations
dest1 and dest2 are the destinations (dest2 being optional)

Example tagged token

context pointer	statement pointer	port	data

where	context pointer	=	address of the start of the activation frame
	statement pointer	=	address of the target statement
	port	=	indicates if the destination is the left or right operand
	data	=	a word of data

Figure 3.3: Example tagged-token data-flow instruction and token formats

The problems with tagged token data-flow are:

- Every time a function is instantiated an activation frame has to be emptied by setting each of the presence bits to *empty* (128 words on the Monsoon machine [51]).
- Matching the first operand for a dyadic operation results in no operation to be performed and thus a bubble in the pipeline.

3.3.4 Assessment of tagged token data-flow

Tagged token data-flow can support the usual array of arithmetic and logical operations. However, unlike control-flow, data-flow supports instruction level parallelism. This fine grained parallelism results in assignment, and thus I/O, being problematic because a serial ordering is difficult to impose. Execution latency of serial code is long because an instruction must traverse the pipeline for each operand. The processor only really becomes efficient when executing more concurrent tasks than there are stages in the pipeline (8 for Monsoon). Furthermore, the scheduling mechanism has to be very quick and overly simple which is inadequate for meeting the temporal requirements of hard real-time systems. However, the inherent parallelism does allow latency to be tolerated. Thus, scalable local and distributed memory structures may be utilised.

The direct mapped matching store is an elegantly simple mechanism for joining dyadic operands. Operands arrive as tokens which may be sent from any of the distributed processors. This supports low overhead parallelism. However, clearing an activation frame in the matching store is an arduous task.

3.4 Multithreaded processors

Multithreaded processors aim to combine control-flow and data-flow ideas to form an amalgam which exhibits many of the advantages of both paradigms whilst trying to avoid the disadvantages [66, 50, 51].

Currently, multithreaded processors are in their infancy. Whilst a wide variety of models have been proposed, they all have some notion of a control-flow section of code, or *microthread*, which is often executed nonpreemptively. Many microthreads executing sequentially form a single logical thread.

At one extreme there have been attempts to add a little control-flow to data-flow machines (e.g. Monsoon [51] and EM-4 [57]) in order to reduce the load on the matching store by making use of a register file. At the other extreme a little concurrency has been introduced to control-flow machines (e.g. the INMOS Transputer [36]) to support concurrent languages (e.g. Occam [46]) and communications between processors (e.g. ⋆T [52], Sparcle [2] and the Transputer [36]). Hiding latency from shared memory has also been investigated (e.g. HEP [37] and Tera [4]).

The following sections look at the range of mechanisms employed to handle multiple contexts (section 3.4.1), communication (section 3.4.2), synchronisation and scheduling (section 3.4.3), memory structure (section 3.4.4) and how they impact upon microthread size (section 3.4.5).

3.4.1 Multiple contexts

A simple approach to context switching quickly is to avoid having context to switch. For example, the T800 INMOS Transputer [36] only has six registers worth of state — three words of evaluation stack, an operand register, a work space pointer (WP) and a program counter (PC). Furthermore, most context switches can only occur at certain instructions (e.g. a jump) where, by definition, the workspace and operand registers may be discarded. Thus, only the WP and PC have to be saved which may be performed quickly.

However, having so little state associated with a thread results in context being continually moved to and from memory rather than making efficient use of a closely coupled store (e.g. a register file). ⋆T [52] simply takes the approach of loading state into a register file at the beginning of a microthread and then saving it again before being descheduled. This is obviously expensive so Sparcle [2] uses a pageable register file to store a few active contexts. HEP [37], Tera [4], D-RISC [13], P-RISC [50], MDFA [28], etc., support a cached pageable register file to allow more context to be stored. However, in practice the size of a cached pageable register file is severely limited because it has to have multiple data paths and still perform at the rate of the rest of the pipeline.

For an efficient context switch, the microthread's code (or *text*) must be available locally to the processor. One could hope that the code was still present in a local cache from a previous execution (e.g. ⋆T [52] and Sparcle [2]). However, assuming a hardware scheduler is present, it is possible to preload code before a process is queued for execution (e.g. MDFA [28]).

3.4.2 Communication

Interprocessor communication may be supported by remote memory requests (e.g. HEP [37], EM-4 [57] and Monsoon [51]) or by message passing (e.g. ⋆T, Sparcle [2] and Transputer [36]). Remote memory requests utilise the usual memory access mechanism but memory may also be tagged with presence bits at each word to indicate whether the word is *empty* or *full* to assist synchronisation (see section 3.4.3).

Communication mechanisms need to be efficient and are, therefore, often positioned very close to the processor to allow transfer of messages to and from the processor's register file and a message processor's input and output queues (e.g. $\star$T [52] and Sparcle [2]).

3.4.3 Synchronisation and scheduling

EM-4 [57] and Monsoon [51] utilise tagged memory to synchronise messages to dyadic microthreads in the usual data-flow manner. Whilst this is an efficient mechanism, it is still expensive when compared to the amount of work required to execute many dyadic microthreads.

Tera [4] has four methods of using tagged memory:

1. wait for full
2. read and set empty
3. wait for empty
4. write and set full

When *wait* operations hit the memory the presence bit is returned to the particular processor's scheduling function unit (*SFU*) which polls the memory (up to a given maximum number of times) until the desired answer is returned. The SFU holds the process status word (*PSW*) and uses this information to reactivate a thread when the desired value has been returned by the memory. Whilst this is a simple mechanism, polling is inefficient even if a task has to wait just a few thousand clock cycles.

HEP [37] uses counters for synchronisation and provides a `join` instruction which decrements a counter at a particular address and if the result is not zero then the thread is descheduled. This mechanism is also simple but assumes that an atomic read/modify/write cycle can be performed on a counter. This is inefficient if one assumes the counter is stored in a memory with a long access latency.

MDFA [28] also uses counters for synchronisation but has a separate event coprocessor to manipulate a signal graph in a static data-flow manner. Like HEP, updating an event counter unfortunately assumes a low latency memory (e.g. as provided by a cache). However, having a separate signal graph is an interesting idea. Each node in the signal graph consists of an *event counter*, a *reset value* for the *event counter*, *acknowledgement addresses* for backward signalling, *forward signalling addresses* and a *code pointer*. When signals arrive at a node the *event counter* is decremented. Once the *event counter* reaches zero, it is reset to the *reset value* and the microthread pointed to by the *code pointer* is executed. Upon completion of the microthread a signal is returned to the node which then sends the forward and backward signals within the signal graph.

$\star$T [52] does not attempt to use concurrency to tolerate latency so does not need to synchronise and schedule on memory accesses. However, it does need to synchronise on incoming messages from other processors, either using conventional interrupts or by polling the message coprocessor. Thus, matching messages to threads is a software overhead.

If hardware support is provided for scheduling, then the prioritising mechanism is usually just in the form of a few FIFO or LIFO queues. For example, the Transputer has two priorities of FIFO queue. This is inadequate for hard real-time and multimedia applications.

3.4.4 Memory

Typically faster processors require larger memories which prevents memory latency from scaling with processor performance. However, it is possible to scale memory access frequency with processor performance provided a pipelined memory structure is used (see chapter 5 for a more complete argument). Some multithreaded processors (e.g. $\star$T [52] and Sparcle [2]) take the control-flow solution of adding caches despite the side effect of temporal nondeterminism. HEP [37], Tera [4], EM-4 [57] and Monsoon [51] all allow concurrency to be used to hide access latency to local and remote memories. Sparcle [2], with its higher context switch overhead, only supports latency tolerance of remote memory.

Memory protection and virtual address translation on most current multithreaded processors relies on a translation lookaside buffer (TLB) (see section 3.2.6). However, a TLB adds temporal nondeterminism and becomes inefficient when the number of threads reaches and exceeds the number of TLB entries. Tera [4] uses its memory latency tolerant characteristics to allow a pipelined memory protection and address translation mechanism to be efficiently used. Using pipelining allows the mechanism to be much larger than a conventional TLB. However, it still does not provide total memory coverage because the structure would be prohibitively large.

3.4.5 Microthread size

The data-flow oriented machines (e.g. Monsoon [51] and EM-4 [57]) can perform efficiently with microthreads which are only one instruction long. Although HEP [37] is more control-flow oriented, it too deals with single instruction microthreads; one instruction is picked in FIFO order from each of the runnable threads and is inserted into the processor's pipeline. Thus, if there are lots of threads then each stage of the pipeline will be executing an instruction from a different thread. However, if there are few threads then, assuming data dependencies are not violated, several instructions from the same thread may be in the pipeline at one time.

Other processors (Transputer [36], $\star$T [52], Sparcle [2], MDFA [28], etc.) all execute a single microthread at a time which is usually several instructions long. The desirable length of a microthread for a particular processor is dependent upon the efficiency of the context switch mechanism and whether a microthread is forced to be descheduled to access local or remote memory. If a microthread is forced to be short, then a large number of threads (e.g. around 70 for Tera [4]) are required to ensure that there is sufficient work for the processor when some threads are waiting due to memory access latency. However, if microthreads are larger then obviously fewer are required to hide memory latency. This is advantageous for the many algorithms which exhibit little parallelism.

3.5 Summary

The control-flow model supports sequential execution which is widely used and well understood. However, faster processors require larger memories which then do not scale with processor performance. Caches may be used to hide some of the latency but in practice they introduce temporal nondeterminism. Furthermore, control-flow processors do not support a strong synchronisation model but instead support some form of interrupt mechanism, relying upon software to complete the task. This obviously adds overhead which impinges upon performance of multithreaded and multiprocessor applications.

The data-flow models support instruction level concurrency which allows memory latency to be tolerated because alternative code can usually be executed whilst waiting for memory responses. Furthermore, a matching store is provided which not only allows synchronisation of memory accesses to instructions but also interthread and interprocessor synchronisation. However, instruction level parallelism makes multiple assignment problematic. This particularly makes input and output operations tricky because they must be serialised. Furthermore, instruction level parallelism forces scheduling decisions to be very rapid, which limits the scheduling policy to being overly simple, e.g. FIFO or LIFO. This is inadequate for hard real-time and multimedia applications. Also, most data-flow models do not support efficient use of a closely coupled store (e.g. a register file) which results in overuse of a relatively slow remote memory (usually the matching store) for intermediate results. Extensions to the tagged token data-flow model allow some serial execution so that a register file may be used but this really moves the model to the edge of the multithreaded processor camp.

Multithreaded processors aim to overcome the problems of both control-flow and data-flow by forming an amalgam of these two models. Thus, some sequential computation, or *microthreads*, is supported to assist the use of a closely coupled store (e.g. a register file) to avoid transmitting intermediate results over long distances. Microthreads typically represent a larger schedulable unit which potentially allows a more sophisticated hardware scheduler to be used. However, to date only FIFO or LIFO hardware schedulers have been deployed. The concurrency potentially allows memory latency to be tolerated but in practice this is dependent upon the efficiency of the synchronisation mechanism.

To conclude, various multithreaded processor designs have exhibited some improved characteristics over the control-flow and data-flow models. However, a good processor design is the result of searching a multidimensional problem space which inevitably leads to subtle tradeoffs in order to form a well rounded design. There is little point in improved characteristics in one area if other areas suffer. The current crop of multithreaded processors offer some solutions to the following problems but none solve the entire problem:

- very low overhead synchronisation and prioritisation of interprocessor and interthread messages
- a hardware scheduler which supports a sensible policy
- controllable concurrency so that overloading of synchronisation resources can be avoided
- prefetching of a microthread's context (both data and text) before being issued for execution
- reasonably efficient execution of purely serial code and efficient execution of code which exhibits only a little concurrency
- scalable and temporally predictable memory structure, virtual address translation and memory protection
- efficient use of closely coupled memory[6]

[6]Where a *closely coupled memory* is one which the processor can access within a single cycle.

Chapter 4

Hardware Scheduling

4.1 Introduction

If real-time requirements are to be met then a simple first-in first-out (FIFO) or last-in first-out (LIFO) scheduler is inadequate. Instead a more sophisticated mechanism of either earliest deadline first (EDF) or fixed priority (FP) is required [44]. Both EDF and FP may be implemented using a priority queue — a specialisation of sorting. For this application a hardware priority queue is require with the following characteristics:

1. A device which can perform an *insert* or *extract minimum* (or as a variant, an *extract maximum*) operation every clock cycle.

2. Records with identical keys should be extracted in FIFO order of insertion so that data-flow style fan-out is predictable.

The rest of this chapter assess current hardware sorting methods and concludes that none meet the above requirements for a priority queue. An extension to the up/down sorter algorithm [41] is proposed in order to meet the above requirements. Implementation issues for this new sorter are also addressed. A formal proof of correctness for the algorithm is presented in appendix A.

4.2 Background

There are many hardware sorting techniques, of which most aim to sort a complete set of data in the minimum time using as little hardware as possible (e.g. Batcher sorting networks [6, 22], heap sort on a systolic array [45] and others [9]). Unfortunately these do not meet our first objective of single cycle insertion and extraction.

In order to meet the first objective it is essential that any number inserted must be compared (and possibly swapped with) the current minimum value. An obvious solution would be maintain a sorted list but this would require $n - 1$ compare and swap units to sort n numbers in a single cycle.

4.2.1 Variations on the heap sort

Typically software implementations of priority queues utilise the heapsort technique which takes $O(\log(n))$ time to insert or extract [15]. Insertions are made at the bottom of the heap and the heap is then massaged into a correct ordering (this process is sometimes named the *heapify* function). Extraction of the minimum is from the top and the hole it leaves is filled by a value from the bottom followed by an invocation of the heapify function.

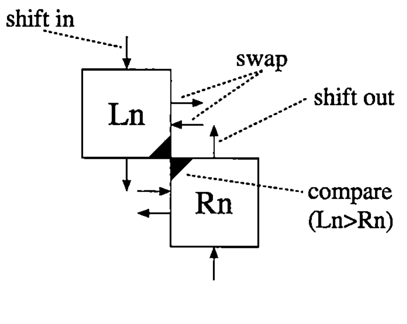
A hardware variant to meet the objectives would require insertion and extraction initiated from the top of the heap. A dedicated processing element (PE) could be placed at each level of the tree structure within the heap. Thus, large values (assuming an extract minimum is required) would ripple down through the levels dislodging even larger values and settling in their place.

The only problem now is to maintain a balanced heap in order to prevent the algorithm degenerating into a sorted list structure. One approach would be to maintain a count of the number of nodes below every node so that at each level of the tree a decision about which lower levels should store the next value. This appears to be inefficient in terms of storage but it should be noted that the number of PEs and the size of the counter for each node would grow $O(\log(n))$. Thus, in terms of silicon real-estate this would work well for large datasets. Unfortunately an insertion or extraction takes at least two cycles (read and examine followed by a write).

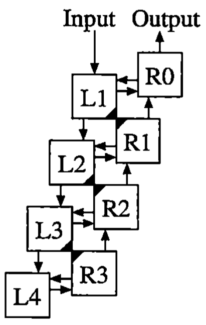
4.2.2 The rebound sorter

The rebound sorter was proposed by T.C. Chen et al. [9] and improved upon by Ahn and Murray [3]. Whilst it is unsuitable for this application, it forms the basis for more suitable approaches.

The basic sorting element (see Figure 4.1) consists of two memory elements capable of storing one word, a comparator and various data paths. Incoming data consists of two words: a word of key and a word of associated $data$ to form a record. Records are inserted key first followed by the data on the subsequent cycle. The comparator is used to compare keys stored in the Ln and Rn parts of the sorting element in order to determine the direction in which the $(key, data)$ pairs should take; this is known as the *decision cycle*. The values in the following cycle will be the associated data so the decision made in the current cycle is used in order that the data follows its key; this is known as the *continuation cycle*.

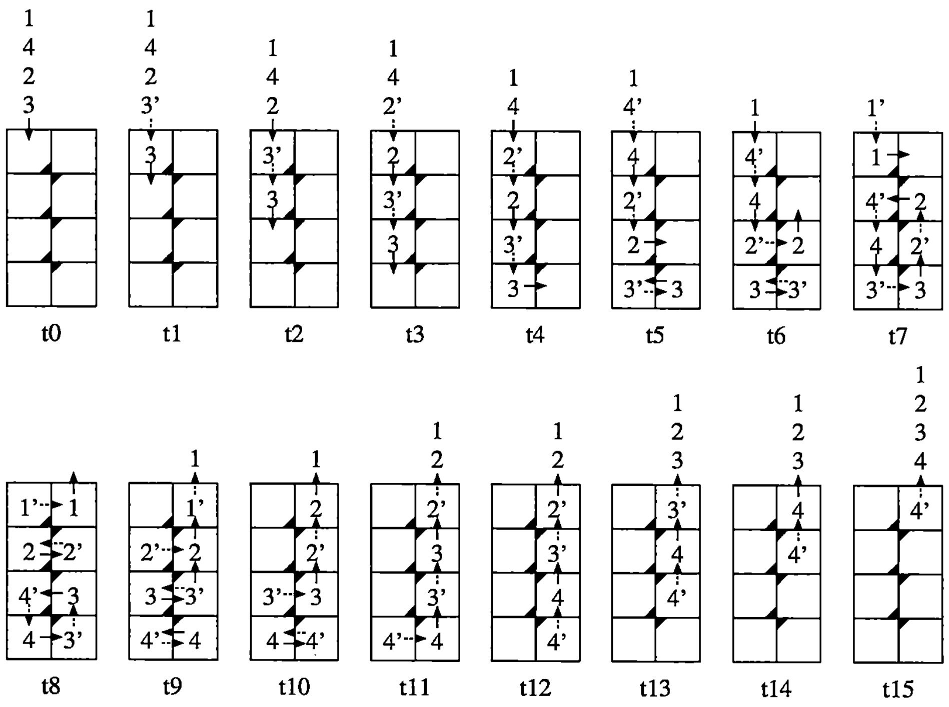


A Sorting Element



The Overall Structure

Figure 4.1: Structure of the rebound sorter



where n' = data associated with key n
→ result of making a decision
⇢ continuation from last decision
comparison point

Figure 4.2: An example of a rebound sort

Figure 4.2 illustrates the sorting behaviour. The principle of the algorithm is that incoming values proceed down the left side until they rebound off the bottom (hence the name) or hit a larger value on the diagonally lower right.

It can be seen that records take two cycles to insert or extract and all of the insertions must take place followed by all the extractions. Thus, this algorithm does not meet our objectives. Furthermore, it should be noted that $n-1$ comparators are required to sort n records and that these comparators are only used every other cycle.

4.2.3 The up/down sorter

The up/down sorting algorithm was originally designed to be implemented in bubble memory technology [41]. It is constructed as a linear array of sorting elements in a similar manner to the rebound sorter described in the previous section. However, $(key, data)$ pairs are inserted in parallel and the sorting element (see figure 4.3) is more complex, primarily because of the implementation technology.

Initially all of the sorting elements contain infinity. An inserted value arrives in A_n. Simultaneously a copy of C_n is made to B_n, and D_n is transferred to A_{n+1}. A compare and steer operation takes place resulting in the maximum of A_n and B_n being transferred to D_n and the minimum of A_n and B_n transferred to C_n. Extraction similarly involves C_n being removed or transferred to B_{n-1}, D_n copied to A_n and C_{n+1} transferred to B_n followed by the compare and steer operation. An example is given in figure 4.4.

Interestingly this algorithm allows an insert or extract operations to be interleaved and only requires $\frac{n}{2}$ sorting elements to sort n numbers. Unfortunately this implementation uses four storage areas per sorting element but if implemented in digital electronics this may be reduced to just two storage areas. The next section abstracts this algorithm, determines that FIFO ordering of identical keys is not maintained and suggests solutions. Then a clocked digital implementation is presented.

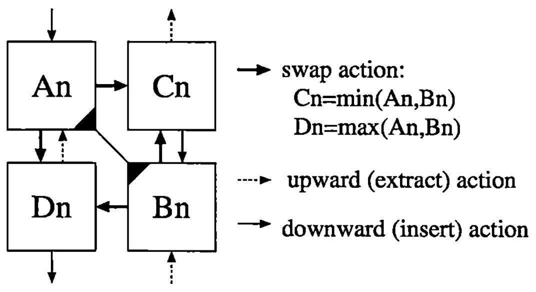


Figure 4.3: The up/down sorting element

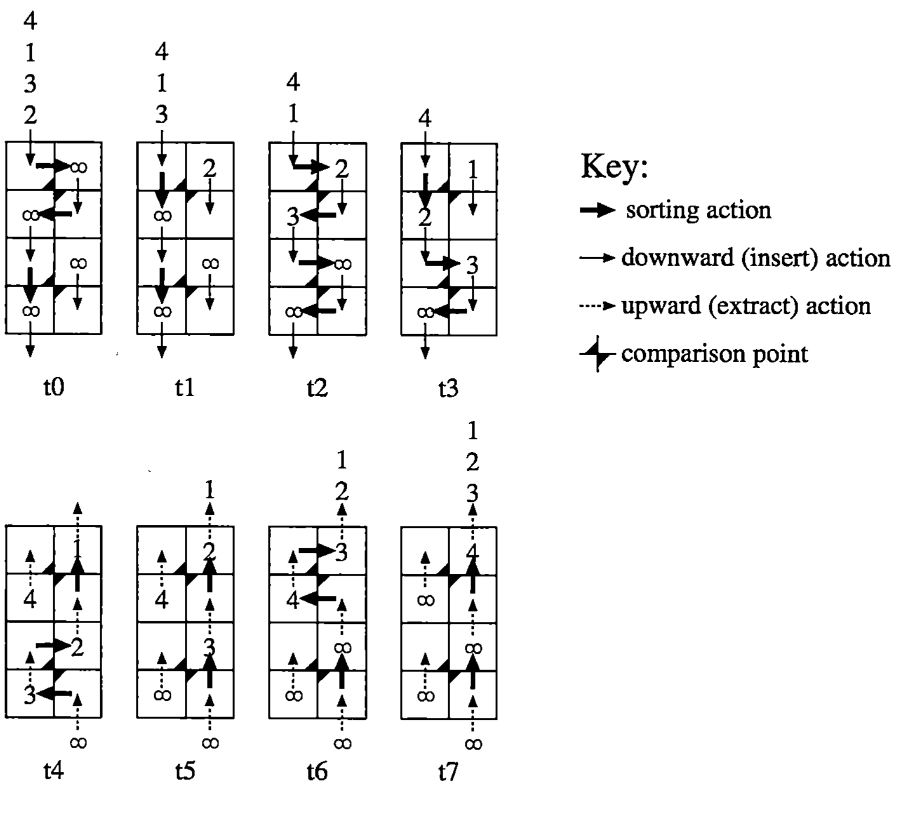


Figure 4.4: An example of an up/down sort

4.3 The tagged up/down sorter

4.3.1 Abstracting the up/down sorter algorithm

The up/down sorting element (see figure 4.3) may be abstracted to two memory elements which may be swapped, and a comparator (see figure 4.5). The algorithm may then be described as a two stage process:

1. Insert:

$$\begin{aligned}&(L'_0 := (k = \mathit{new\_key}, d = \mathit{new\_data}))\\ \wedge\ &(\forall\, n \geq 0.(L'_{n+1} := L_n))\end{aligned}$$

   or extract:

$$(\mathit{extracted} := R_0) \wedge (\forall\, n \geq 0.(R'_n := R_{n+1}))$$

2. Compare and swap:

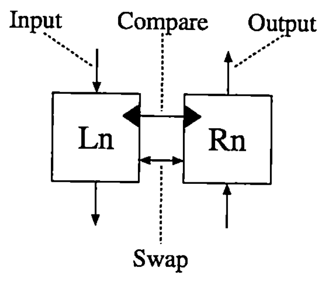
$$\forall\, n \geq 0.(L'_n, R'_n) := (L_n.k < R_n.k \Rightarrow (R_n, L_n) \mid (L_n, R_n))$$

where k=key part of the record
d=associated data part of the record
L_n,R_n=the current left and right records
L'_n,R'_n=the next left and right records
$(c \Rightarrow a \mid b)$=if c then a else b

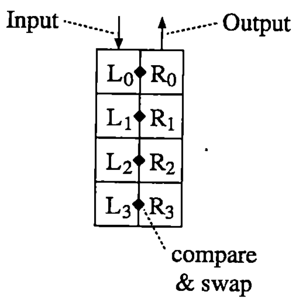
Whilst this algorithm sorts correctly, the FIFO ordering of records with identical keys is not maintained (see figure 4.6). This problem arises when records on the right are swapped back to the left.

4.3.2 Ensuring FIFO ordering

FIFO ordering could be assured by associating an order of entry number with each record. However, a cleaner solution is to tag records by setting a single tag bit when they arrive on the right so that if they are swapped to the left they can be forced to swap back to the right on the next cycle. This works because once a record arrives on the right it must be sorted with respect to the other keys on the right. If a record gets swapped to the left, then on the next cycle (regardless of whether an *insert* or *extract* takes place) it will be compared with the right value which was previously physically below it. Thus, the right key must be either greater than the one on the left or have the same key. However, the record on the right was inserted later than the record on the left. Therefore, a



(a) single element



(b) a four element structure to sort eight numbers

Figure 4.5: Abstraction of the up/down sorter

swap must be performed if the ordering on the right is to be maintained (see figure 4.7 for an example). This is formally proved in appendix A.

The tagged up/down sorting algorithm may thus be defined as a two stage process:

1. Insert:

$$(L'_0 := (k = new\_key, d = new\_data, t = 0)) \\ \wedge\ (\forall\, n \geq 0.(L'_{n+1} := L_n))$$

or extract:

$$(extracted := (R_0.k, R_0.d)) \wedge (\forall\, n \geq 0.(R'_n := R_{n+1}))$$

2. Compare and swap:

$$\forall\, n \geq 0.(L'_n, R'_n) := ((L_n.k < R_n.k) \vee (L_n.t = 1)) \Rightarrow \\ (R_n, (L_n.k, L_n.d, t = 1)) \mid (L_n, R_n)$$

where	k=key part of the record
	d=associated data part of the record
	L_n,R_n=the current left and right records
	L'_n,R'_n=the next left and right records
	$(c \Rightarrow a\|b)$=if c then a else b

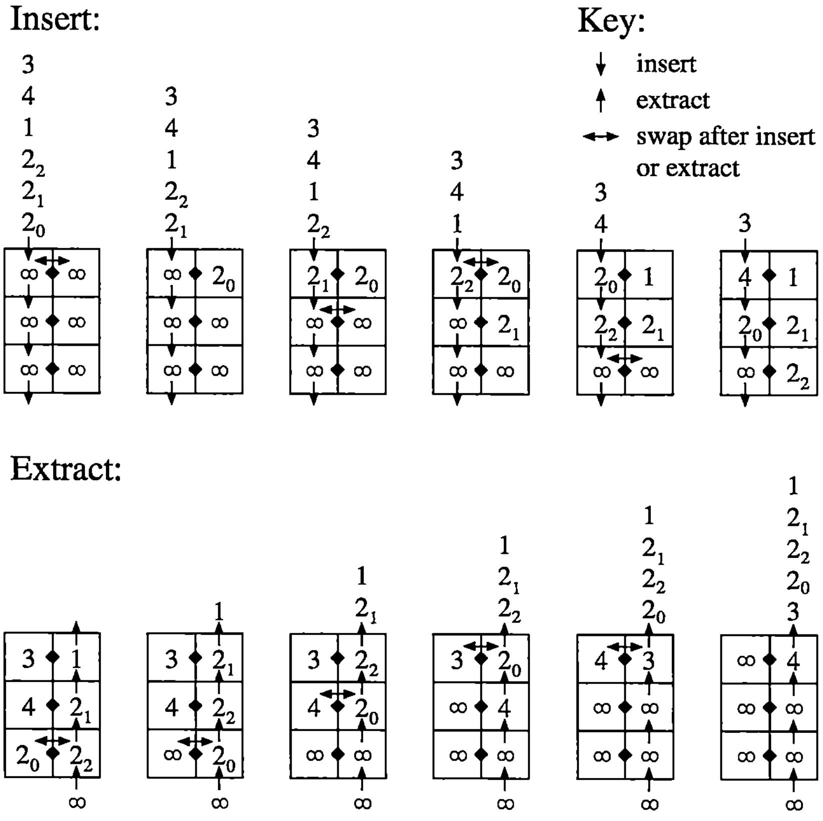


Figure 4.6: Ordering problem with the up/down sorter

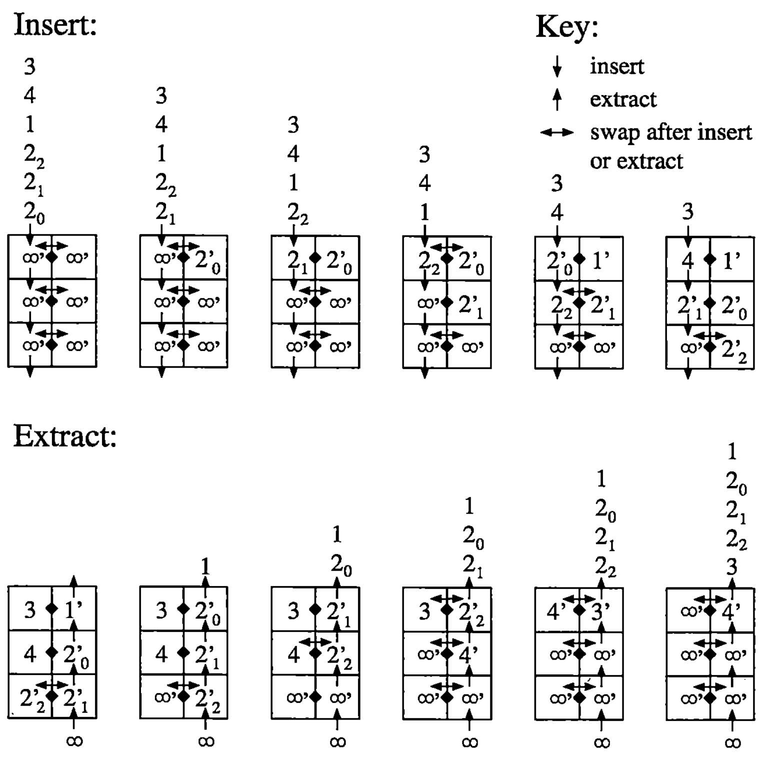


Figure 4.7: Using tagging to ensure FIFO ordering

4.4 Clocked digital implementation of the tagged up/down sorter

A naïve two step clocked digital implementation is presented in figure 4.8. Values are shifted in and compared in the first cycle and if the comparison requires it, a swap is performed on the next cycle. However, this may be reduced to a single cycle process by redirecting the inputs and outputs of the latches rather than swapping the values between the latches (see figure 4.9).

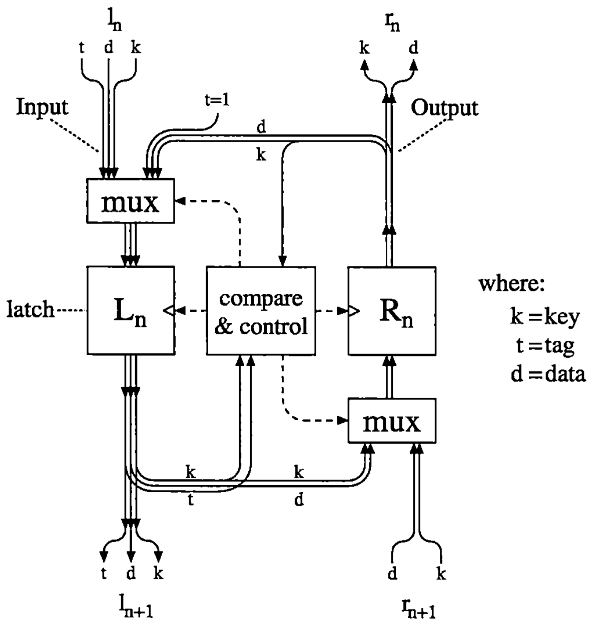


Figure 4.8: Two stage tagged up/down sorting element

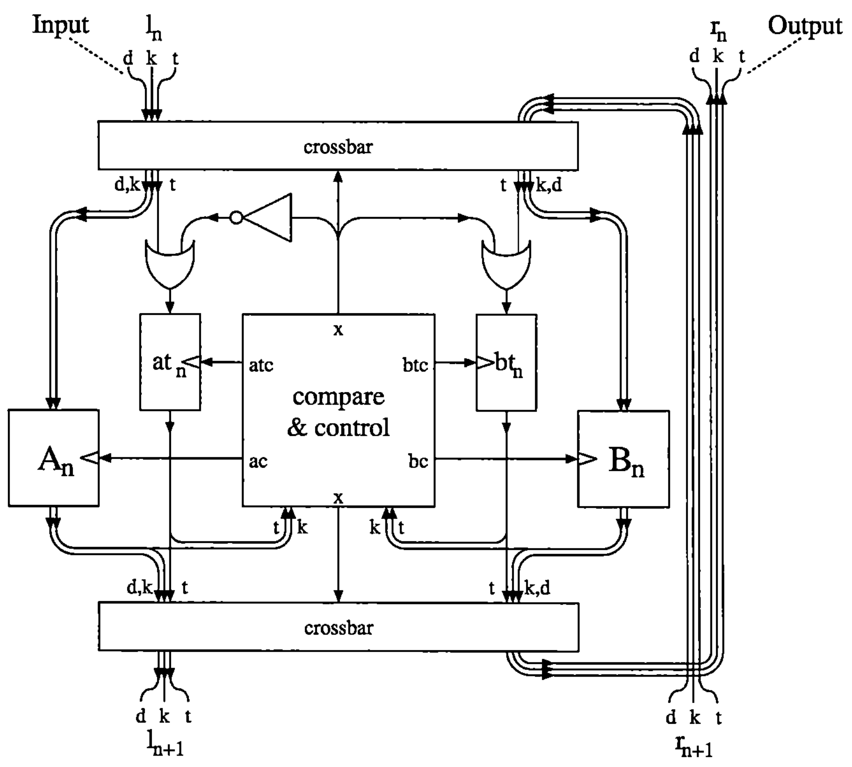
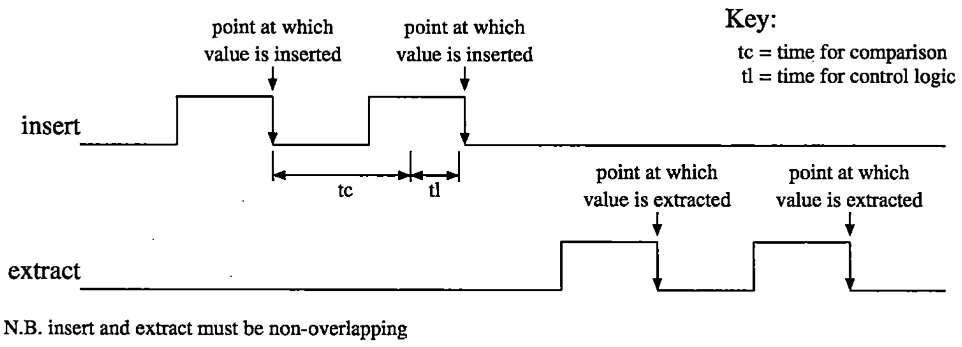


Figure 4.9: One stage tagged up/down sorting element



point at which value is inserted
point at which value is inserted
Key:
tc = time for comparison
tl = time for control logic
insert
tc
tl
point at which value is extracted
point at which value is extracted
extract
N.B. insert and extract must be non-overlapping

Figure 4.10: Example timing for the one step tagged up/down sorter to perform two inserts followed by two extracts

4.4.1 Controlling the single cycle implementation

The two crossbars are controlled by x which maps A_n to the left and B_n to the right if $x = true$, otherwise A_n maps to the right and B_n to the left. Insertion and extraction are controlled by $insert$ and $extract$ signals which are mutually exclusive. An insert or extract is performed by pulsing the appropriate control line high (see figure 4.10) which clocks the required latches for A_n, B_n and x on the falling edge. ac, atc, bc and btc control the clocking of the (A_n, at_n) and (B_n, bt_n) latches. If $x = true$ then the left value is in A so ac and atc will be clocked if $insert$ is pulsed. If $x = false$ then the right value is in A so ac and atc will be clocked if $extract$ is pulsed; however, to force tagging of right hand side values, atc will also be clocked if $insert$ is pulsed and the OR gate arrangement into at_n will set the tag bit. Thus, the tag is set on the following cycle. The corresponding logic is required for B, but with $\neg x$.

The control equations for the one step control logic are defined by (assuming that the flip-flops are negative edge-triggered, i.e. they latch on the falling clock edge):

let x = control for crossbars:

$$(l_{out}, r_{out}) = (x \Rightarrow (l_{in}, r_{in}) \mid (r_{in}, l_{in}))$$

$oldx$ = x latched on the falling edge of $insert \vee extract$

$A_n.k$ = key part of record in latch A_n

$A_n.t$ = tag part of A_n

$B_n.k$ = key part of record in latch B_n

$B_n.t$ = tag part of B_n

$insert$ = insertion control signal

$extract$ = extraction control signal

N.B. $insert \wedge extract$ = false

$$\begin{aligned} x = (&(oldx \wedge (A_n.k = B_n.k)) \\ &\vee (A_n.k > B_n.k) \\ &\vee (\neg oldx \wedge B_n.t) \\ &) \wedge \neg(oldx \wedge A_n.t) \end{aligned} \qquad (1)$$

$$ac = (x \wedge insert) \vee (\neg x \wedge extract) \qquad (2)$$

$$bc = (\neg x \wedge insert) \vee (x \wedge extract) \qquad (3)$$

$$atc = insert \vee (\neg x \wedge extract) \qquad (4)$$

$$btc = insert \vee (x \wedge extract) \qquad (5)$$

4.4.2 Discussion of the operation of the single cycle implementation

First we consider the insert operation. The record to be inserted is presented at l_n (see figure 4.9), the $insert$ signal is pulsed $true$ and $extract$ remains $false$. The latch used to hold the record will depend upon the value of x which is determined by the contents of (A_n, at_n), (B_n, bt_n) and $oldx$ before the insert takes place. If, for example, we take $x = true$ (so the A_n and at_n latches are holding the left record) and perform an insert (so $extract = false$) then the control equations (2) through (5) become:

$ac = atc = btc = insert$

$bc = false$

Thus, since $x = true$ the new record (at l_n) will be placed on the inputs of A_n and at_n which will be latched into place on the falling edge of the $insert$ signal by ac and atc (the original record in (A_n, at_n) being propagated to the next sorting element). We can also see that latch B_n is not clocked because bc remains $false$ but that the tag bit is set by the OR gate arrangement into bt_n and the clocking signal btc.

On the falling edge of $insert$ the current value of $x = true$ is transferred to the variable $oldx$ and the next value of x is calculated from (1):

$x = ((A_n.k = B_n.k) \vee (A_n.k > B_n.k)) \wedge \neg A_n.t$

Thus, the crossbar only causes a swap (x goes from $true$ to $false$) if $(A_n.k < B_n.k) \vee at_n$ which conforms with the algorithm in section 4.3.2. Furthermore, it should be noted that if a swap has occurred then (B_n, bt_n) has been remapped from the right output (r_n) to the left output (l_{n+1}) and that this record has been correctly tagged. Like wise, if we had started with $x = false$ then similar conformation would be obtained.

Now consider the extract operation. If we start with $x = false$ (so the A_n and at_n latches are holding the right record and r_{n+1} is the input) then equations (2) through (5) become:

$ac = atc = extract$

$bc = btc = false$

Thus, r_{n+1} will be latched into A_n, and the tag bit at_n will be set, on the falling edge of $extract$. At this point the value of $oldx$ will be set to $false$ resulting in the following calculation of the next value for x:

$x = (A_n.k > B_n.k) \vee \neg B_n.t$

It can be seen that the conditions for x to change from $false$ to $true$, thereby causing a swap, correspond with the specification in section 4.3.2 (remembering that $oldx = false$ so B_n was mapped onto the left and A_n onto the right). Furthermore, the value shifted in has correctly had its tag set.

4.5 Conclusions

A tagged up/down sorter has been proposed as a suitable mechanism to support EDF or FP scheduling policies. The algorithm requires just $\frac{n}{2}$ comparators in order to sort n records. The objectives of single cycle *insert* and *extract* operations has been met. Furthermore, *extract* always removes the record with the least key, and in the case of repeated keys FIFO ordering is maintained.

Implementation strategies were also outlined since this is a novel algorithm. The resulting clocked implementation is a regular one which scales reasonably well, although it is limited by the transmission of the *insert* and *extract* signals. It is anticipated that a self-timed version would scale indefinitely.

A formal specification of the algorithm and a proof of correctness has been undertaken by a colleague, B. T. Graham, using the HOL system [29]. An outline of the proof is presented in appendix A.

Chapter 5

Memory structure

5.1 Introduction

Historically the implementation of functions and control structures has differed from those of memory. In the early days of stored program electronic computers [69] valves (or *vacuum tubes*) were employed to implement functions and control structures but valve based memory would have been very large and probably too unreliable. Consequently memory required other technology: first ultrasonic pulses in mercury tanks, then phosphor persistence on a cathode ray tube (the *Williams tube*) and later on magnetic core memory. Today, main memory (which is still sometimes referred to as core memory) is constructed using silicon based technology as are the functions and control structures for processors. However, the large size and regularity of memory allows different optimisations to be made.

Since processors and memory are typically constructed using similar technology it might appear, at first sight, that technology improvements would enable both faster memories and faster processors. However, faster processors are used to tackle larger problems which typically require a larger memory. Section 5.2 elaborates this problem. Solutions for control-flow processors are discussed in section 5.3 and an alternative structure, which maintains access frequency at the expense of latency, is presented in section 5.4. Then scalable virtual addressing (section 5.5) and memory protection (section 5.6) schemes are presented. Finally the memory interconnect is discussed (section 5.7).

5.2 Memory performance

Both static and dynamic memories are constructed as a grid of storage bits. Reading from the memory is performed by requesting that one row of storage passes its contents down the columns to the bottom where the columns which are required are selected. For a particular chip density it can be seen that the length of row and column signals are dependent upon $\sqrt{(chip\ area)} \propto \sqrt{(storage\ capacity)}$. From this approximation we can see that larger memories have longer row and column lines which take longer to access data unless the density is increased to reduce the distances and capacitances. However, improvements in density allow faster processors which are used to tackle larger jobs which tend to require higher capacity memories. Unfortunately it is then impossible to scale processor performance with single memory performance since signal speed is limited by the speed of light (see section 2.3.1).[1] In practice several slightly smaller memories are used connected together by a bus. However, this merely makes the interconnect topology more complex rather than solving the access speed problem.

5.3 Memory hierarchy for control-flow processors

Today's RISC based control-flow processors make extensive use of pipelining to overlap steps in instruction execution. Typically one stage of the pipeline performs memory accesses. Whilst it is possible for `store` instructions to be posted to memory without stalling, `loads` soon stall due to data dependencies. Thus, control-flow processors are dependent upon memory access latency. There are two commonly used hardware solutions to this problem: caching and scoreboarding.

5.3.1 Caching

The processor can be fitted with a small local memory, or *cache*, to store recently used results thereby reducing the number of accesses to the large main memory. However, the cache must be fast enough to keep up with the processor so it cannot be too big. If there is a large disparity between the performance of the cache and main memory then further levels of cache may be deployed, thereby forming a pyramid structure, both spatially and temporally. Unfortunately there is then a huge range of memory accesses speeds, from the fast access time of the first level cache to an increasingly large access time for the main memory (which can be $> 100 \times$ first level cache latency).

Analysing the temporal characteristics of a cache must take into account memory access patterns. Whilst this is practical for simple single-threaded stand-alone applications, it is problematic in a reactive and/or multithreaded applications where memory access patterns vary wildly with input data and scheduling decisions. It is possible to

[1]This is backed up by trends in DRAM technology which shows every sign of continuing a four-fold improvement in density every three years whilst performance improves by only 7% and processor performance improves by 365% [54].

partition a cache into isolated sections to separate thread interactions[39]. However, this severely reduces the cache performance, particularly if a large number of threads are used.

Predictive mechanisms may be used to attempt to prefetch data and code into a cache [11]. Mechanisms may use either algorithms embedded into the hardware or `fetch` instructions inserted into the code. Hardware mechanisms tend to be *reactive* where decisions are based upon historical memory reference patterns. Conversely, software mechanisms are usually *proactive* — code is statically or dynamically analysed to assess memory access patterns and `fetch` instructions are then inserted at appropriate points. However, predictive mechanisms tend only to work well in a sequential environment. Once scheduling takes place, particularly if at a reasonably high frequency with sporadic tasks, then the predictive algorithms are crippled because it is too difficult to determine what will be in the cache when.

5.3.2 Scoreboarding

The impact of a cache miss may be reduced by avoiding stalling the pipeline until the data from a `load` is required. This may be performed by marking a *scoreboard* to indicate which registers are awaiting a loaded value and only stalling the pipeline if the register is used before the value has returned from the memory subsystem. With careful compiler design, `load`s may be performed a few instructions before the result is needed. However, this limited use of parallelism only softens the impact of cache misses upon performance rather than removing the need for caches.

5.4 Maintaining memory access frequency

An alternative to caches is to use lots of small high speed memories with a scalable interconnect rather than a bus which does not scale well. For example, a tree could be used (see figure 5.1) which forms a pipeline where each router and memory is a stage in the pipeline. Although this allows the frequency of memory accesses to be maintained, the latency will be dependent upon the length of the pipeline. However, it should be noted that the latency only grows logarithmically with the memory size:

$$\begin{aligned} total\ latency = 1+ & \\ & 2\times(maximum\ pipeline\ stage\ time) \\ & \times \left\lceil \log_{(fan\ out)} \left\lceil \frac{total\ memory\ size}{memory\ module\ size} \right\rceil \right\rceil \end{aligned}$$

Thus, for example, a 2^{64} byte memory (which is probably more memory than has ever been built to date) made up of 2^{48} tiny 64 kbyte memories in a tree with a fan out of

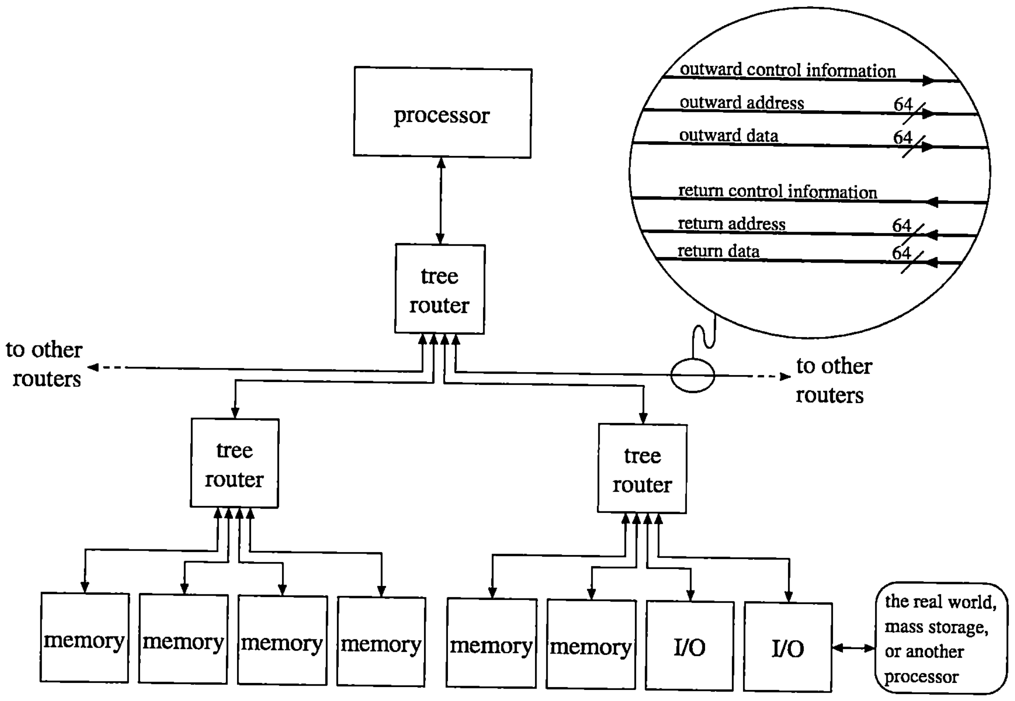


Figure 5.1: Memory tree structure

8 would have a total latency of only 33 machine cycles. Of course, the memory modules could be bigger and slightly slower than the processor provided memory modules are interleaved and there is a reasonable spread of address values.

5.5 Virtual addressing for the memory tree

Virtual addressing is very useful for memory allocation but the use of paging is too temporally unpredictable for many hard real-time systems. The traditional approach to virtual addressing is to store the translation information in main memory and provide a cache, or *translation look-aside buffer* (TLB), to store recently used translations [54]. Translation is performed by looking up the pair $\langle process\ id, page\ number\rangle$ in the TLB. The TLB is usually fully associative and, because of space and performance limitations, is often around 64 to 1024 entries. This is likely to be inadequate to map all of the memory so TLB misses must be expected. Unfortunately, like data and program caches, the temporal nature of TLBs is hard to predict.

Alternatively the virtual to physical mapping could be performed by a fully associative table on each memory module. This table would represent some small fraction (around 2% to 4%)[2] of the memory module provided that there are no many to one mappings between $\langle \textit{process id}, \textit{virtual page number}\rangle$ and $\langle \textit{physical page number}\rangle$.[3] Meeting this requirement raises problems with providing protection for shared pages but this is not insurmountable (see section 5.6).

5.6 Scalable memory protection

During execution, protection mechanisms are required, at least to provide memory protection to guard against unwanted inter-application interference. There are two orthogonal approaches:

1. Each area of memory has a list of processes and the operations which they are allowed to perform (e.g. read and/or write). This structure is called an *access list*. A translation lookaside buffer may be thought of as an access list cache.

2. Each area of memory has an access *capability* for each possible operation which must be presented like a key to unlock the operation.

In the former a memory area has a list of process privileges and in the latter a process has a list of memory privileges. Either approach could potentially be used to specify the same level of protection and at any granularity [70]. However, only page-level protection will be considered here since modern high level language compilers are able to provide a reasonable level of intra-page protection often without run-time overheads.

Page-level protection allows fire walls to be set up to isolate threads. However, some areas of shared memory are often desirable to allow message passing, in a controlled manner, between threads. Thus, for example, for message passing one thread may have read only access to a page of memory whilst another has write only access. If an access list is used then its size will vary according to the number of threads which can access each page. This dynamic growth is undesirable if protection is to be combined with virtual address mapping at each memory module. However, if capabilities are used then each memory module only needs to know what the access capability is for each of its pages.

[2]If 8 kbyte pages are assumed and that approximately 20 bytes (8 bytes of virtual address to 4 bytes of physical address and an 8 byte capability) of fully associative memory is required per page which takes up approximately 10 times the area per bit then approximately 2.5% of the memory module would be translation table.

[3]This restriction should not be confused with a subset of this problem called *aliasing*, where many virtual page numbers map onto one physical page number, since if there is no aliasing (a one to one mapping between all virtual and physical pages) this does not guarantee that the stricter requirement is met.

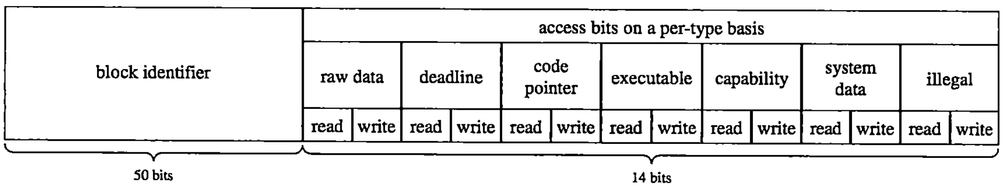


Figure 5.2: Example capability format

Capabilities for a page-based system are allocated in *domains* (or *strides* or *blocks*) where each domain requires two capabilities: a read and a write. This delegates the job of dynamic list management to each thread which must present the correct capability to perform each memory access. Nevertheless, a thread is likely to only access a few memory domains and thus the list will be short and manageable.

Protection of executable images (sometimes referred to as *text*) is an interesting problem, particularly if shared libraries are required. An elegant and simple approach is to allow any application to execute anything and only protect data reads and writes [56]. This allows free sharing of executables albeit at the risk of making the location of bugs harder to identify. The underlying mechanism is simple — just allow the instruction fetch unit to access any area of memory by allowing it to use the supervisor capability. Safety is assured because memory modification protection is a property of the capabilities an application has rather than the code which it is executing.

Some finer grained protection may be added by introducing a few basic types which, together with the capability, indicate to the processor and memory system what may be read, written and modified. For example:

0 –	raw data	4 –	user capability
1 –	deadline	5 –	system data
2 –	code pointer	6 –	illegal
3 –	executable	7 –	system capability

Most of these types have associated read and write bits in the capability to indicate how the data may be used (see figure 5.2). Further details of the meaning of these types can be found in section 6.6 and an example of their use in section 7.7.

5.7 Tree routers

Messages going down the memory tree from the processor to the memory need to be broadcast because the virtual address gives no indication of which memory module will accept the message. Each memory module receives the request and attempts to perform a virtual→physical translation followed by a capability check.

Inputs			Outputs
invalid	invalid	$\rightarrow$	invalid
invalid	valid	$\rightarrow$	valid
valid	invalid	$\rightarrow$	valid
valid	valid	$\rightarrow$	invalid

Figure 5.3: Tree mapping of messages going up the tree from the memory modules to the processor

For `stores`, if the translation and capability check pass then the `store` is performed, otherwise the `store` is ignored. Little else can be done without passing additional information about the sending microthread because the microthread which issued the `store` has probably terminated at this point in time. However, the simple approach does not impinge upon security, it merely makes debugging of programs more difficult. Of course testing to see if the correct value is as at an address may be performed by issuing a `load` immediately after a `store`.

For `loads`, if translation and capability checks pass then the valid data and destination address are passed back up the tree, otherwise null data with type *invalid* is sent. Thus, routing of signals going up the tree from the memory to the processor compares the two inputs and performs the mapping in figure 5.3.

Broadcasting messages down the tree consumes power which increases exponentially with depth, assuming that each router uses the same amount of power to route a message. By placing the address translation and capability checking a few level up the tree would allow a physical address to be generated sooner, thereby allowing message to be routed directly to the appropriate memory module without further broadcasting.

5.8 Summary

A *tree memory*, consisting of many small memory modules and a tree topology interconnect, was proposed. This structure has the property that memory access frequency may be maintained with increasing processor performance whilst memory access latency increases by only $O(\log(size))$.

Single address space virtual address mapping is supported at the memory module level. This provides a total address map which grows with the number of memory modules.

Coarse grained protection is also provided at the memory module level by providing read and write capabilities per page. Additional fine grained protection is achieved using types which control how the processor can use the data.

Chapter 6

Anaconda — a multithreaded processor

6.1 Introduction

The motivations in chapter 2 and the review of past processor models in chapter 3 suggest that a multithreaded processor may meet the requirements for high performance real-time and best-effort computing. This chapter describes a multithreaded processor, Anaconda, which was designed with real-time and best-effort requirements in mind. The underlying model is based upon data driven microthreads (further details in the next section). A microthread, in this context, is a nonpreemptable control-flow unit of code which accepts up to 16 input parameters, in a data-flow manner, before execution commences. During execution, data is passed to other microthreads and messages are posted to the memory system.

Matching parameters to microthreads is provided by a direct mapped matching store (section 6.3) not dissimilar to that used by tagged data-flow machines (see section 3.3.3). A hardware scheduler is required to support this model. The hardware priority queue, proposed in chapter 4, is used (section 6.4). Memory (section 6.5) and a protection system (section 6.6) is provided by the memory tree structure, capability and type system presented in chapter 5. Microthread instructions (section 6.7) are based upon the Digital Alpha instruction set.

Finally, further details are presented starting with the matching store, code preloading and context switching mechanisms (section 6.8), and moving on to operating system support (section 6.9), I/O devices (section 6.10) and the execution pipeline (section 6.11).

Figure 6.1 illustrates the interconnectivity of the basic blocks which are discussed in the following sections. Simulation and evaluation appear in the next chapter.

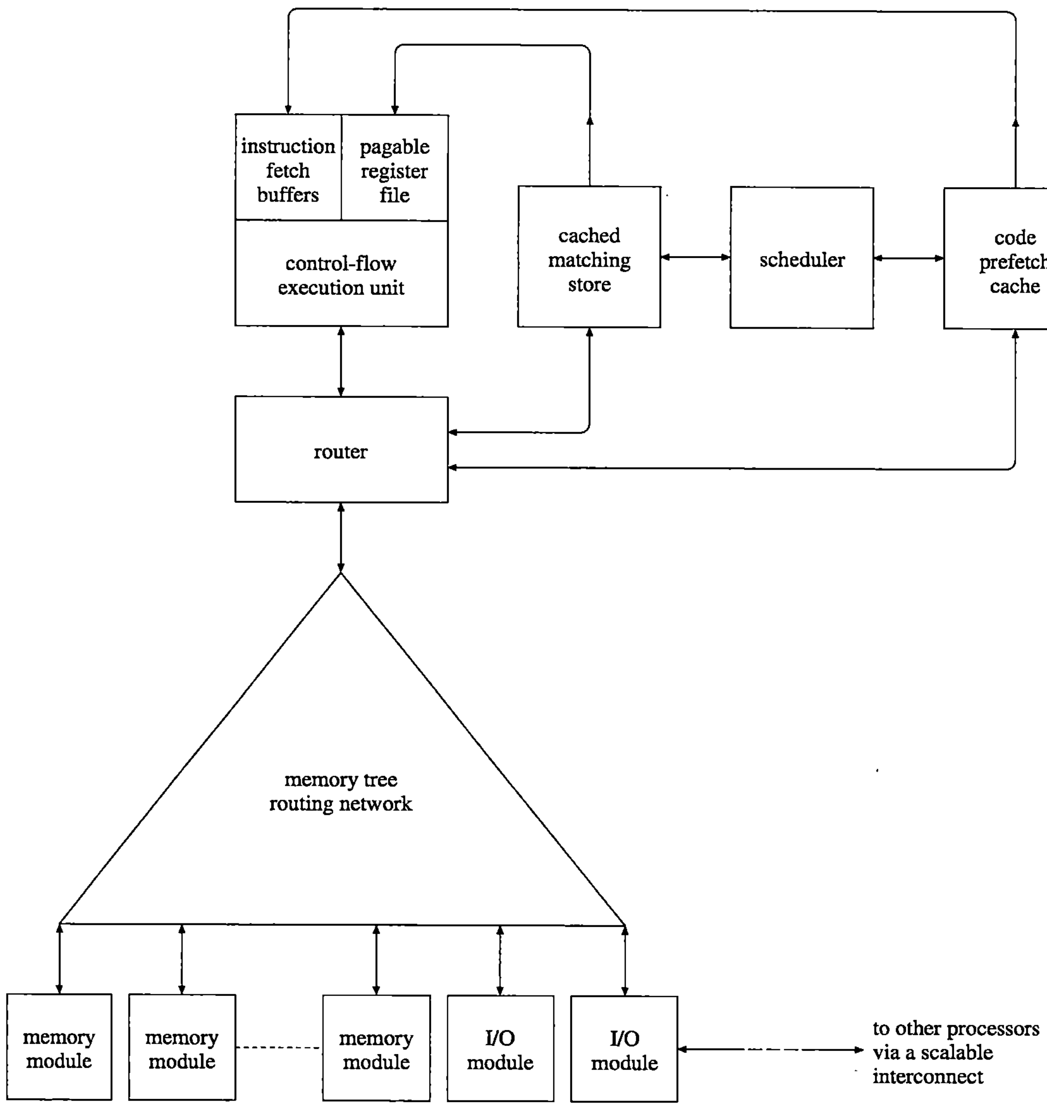


Figure 6.1: The overall structure of Anaconda

6.2 Data driven microthreads

Anaconda is designed to execute data driven microthreads, which can also be thought of as large grained data-flow where each data-flow node is a microthread. Each microthread consists of a control-flow routine with between 8 and 32 instructions. Microthreads have up to 16 input parameters which must be presented before execution can commence. An instance of a microthread stores its parameters in an activation frame. Thus, there is a similar relationship between microthreads and activation frames as there is between functions and stack frames on a control-flow processor. A matching store is provided for joining parameters to microthreads by writing them to the appropriate microthread's activation frame and recording which parameters have been written (see section 6.3). When an activation frame is full it is scheduled by the hardware (see section 6.4).

A large sequential routine may be broken up into a number of sequentially ordered microthreads to form one logical thread of control (see figure 6.2). Only one activation frame is required when executing a single sequence of microthreads because only one microthread is active at any one time so the activation frame may be reused. Communication with memory and I/O is supported by posting messages. Stores simply post a write request to the memory system. Loads do not stall but instead are performed split phase: one microthread posts a memory request and specifies a destination microthread using an address into its activation frame. Thus, the data loaded is sent as the input parameter to an awaiting microthread. Furthermore, the microthread which initiates the

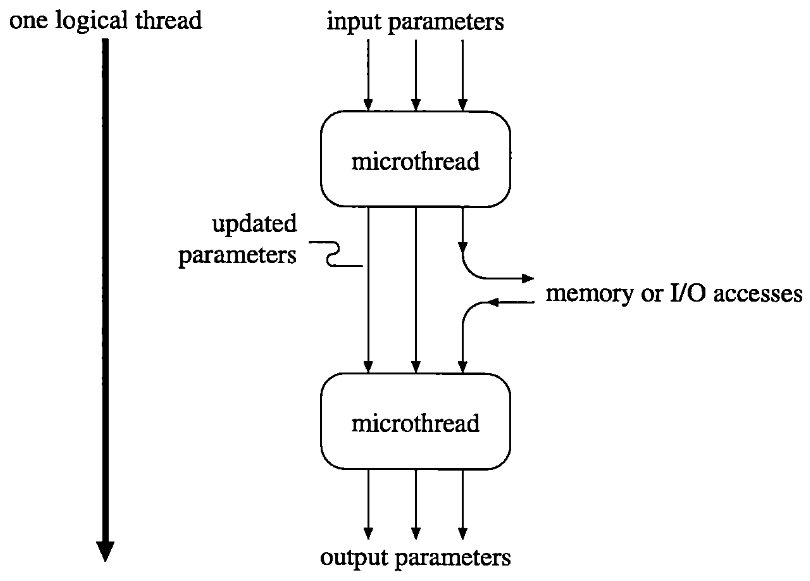


Figure 6.2: A single logical thread constructed from a sequence of microthreads

transaction does not need to stall awaiting the memory response.

Conventional multithreaded programs may be constructed from microthreads. For example, figure 6.3 demonstrates the microthread structure for one thread spawning (*forking*) two more threads and then waiting for them to complete before proceeding (*joining*). In this instance, just three activation frames are used, one for each thread.

A more data-flow oriented style may also be supported. For example, figure 6.4 illustrates a data-flow styled bubble sort constructed from `min/max` microthreads which accept 10 parameters as input, and output the lowest 5 parameters to the left and the 5 highest parameters to the right. Further programming examples appear in the evaluation (see chapter 7).

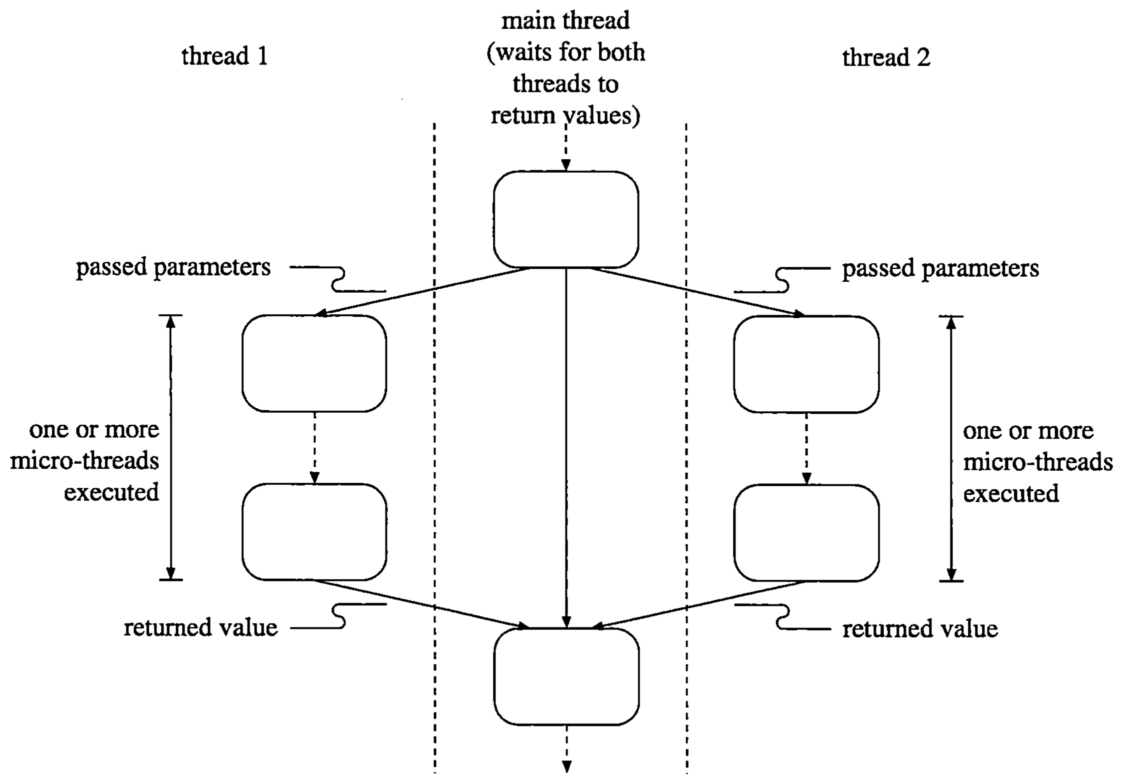

thread 1
main thread (waits for both threads to return values)
thread 2
passed parameters
passed parameters
one or more micro-threads executed
one or more micro-threads executed
returned value
returned value

Figure 6.3: Example microthread structure for forks and joins

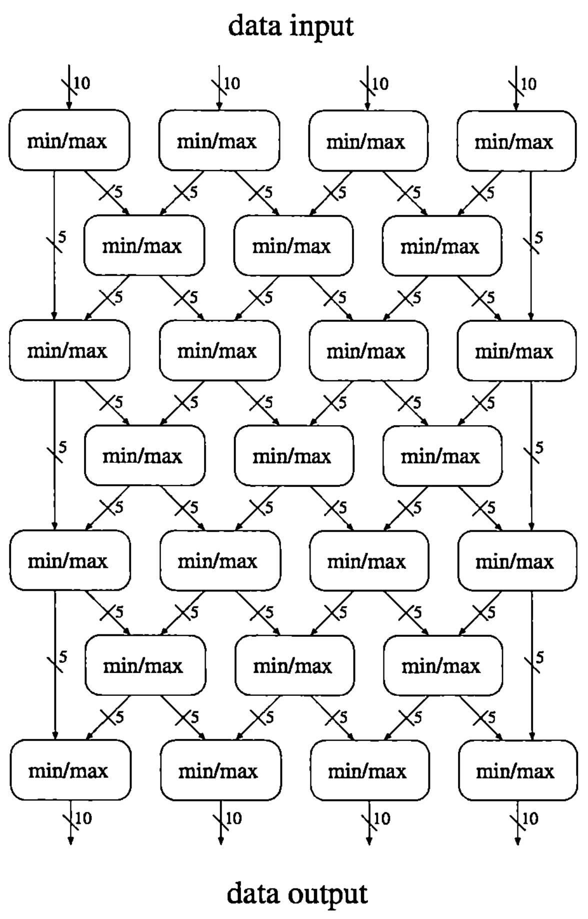


Figure 6.4: Data-flow styled bubble sort

6.3 Matching

A matching store is required to join up to 16 input parameters (each an 8 byte quadword long) to each microthread. Storage for a microthread's parameters is called an *activation frame* (see figure 6.5). Activation frames are allocated in memory, like any other data, but are usually held in a fully associative local cache, one cache line per activation frame, so the base address of the activation frame must be cache aligned. Caching ensures that intermediate results are localised and allows rapid transfer of context because it is practical to have very wide busses if they are short.

The matching process uses presence bits, one for each quadword of data. Presence bits are provided in the upper 16 bits of the first quadword of the activation frame (the lower 48 bits holding the deadline — see the next section). Each presence bit indicates whether a corresponding quadword is full (1) or empty (0); bit 48 is the flag for quadword 0, bit 49 for quadword 1, etc.

Storing presence bits as data allows them to be set and cleared in a single write. However, this approach does mean that conventional memory stores and stores to the matching store must be treated differently. This is achieved by providing two forms of store instructions: conventional data stores and stores to the matching store which have the side effect of setting the appropriate presence bit. This also allows the differentiation between what should be cached in the matching store and what should be written through to the main memory.

The activation frame also contains a pointer to the microthread's code as well as a home and a temporary capability. These will be explained in the following sections.

When an activation's presence flags are all set, it is issued to the scheduler. At this point, the presence flags are also cleared to prevent the odd spurious message (e.g. from a rogue processes — see section 7.7) causing the microthread to be scheduled twice.

Before a microthread is executed, the activation frame parameters and activation frame address are preloaded into the register file (see figure 6.6). See section 6.8 for further details.

dl	cp	hc	tc	p0		p11
16 presence bits 48 deadline bits	micro-thread code pointer	home capability	temporary capability or data	data quadword 0		data quadword 11

Figure 6.5: Activation frame format

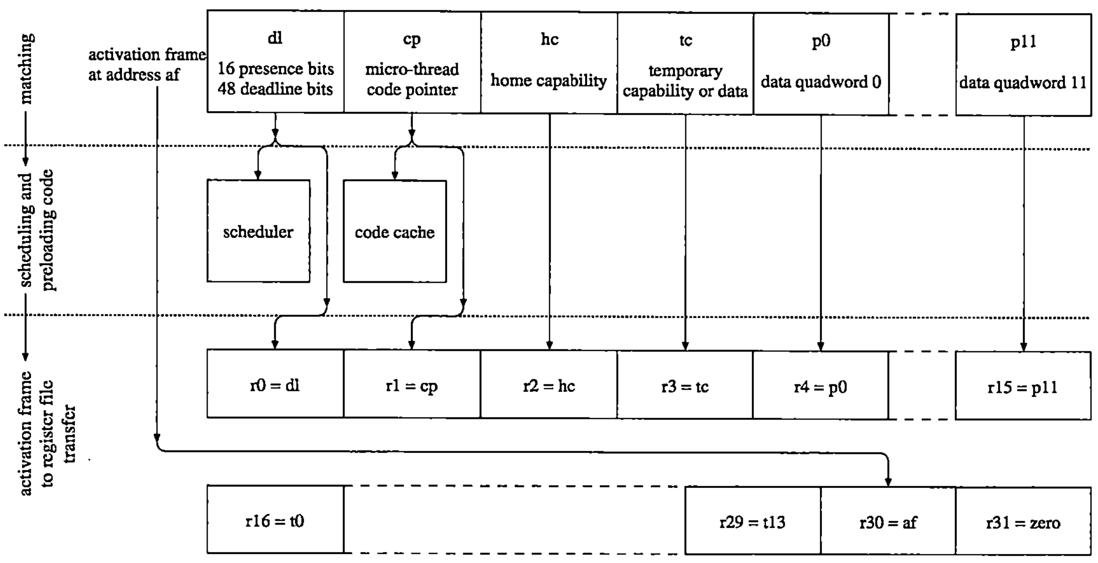


Figure 6.6: Preloading an activation frame and prefetching code

6.4 Scheduling

Microthreads need to be scheduled so that the processor resource is allocated in a timely manner. Earliest deadline first and fixed priority scheduling are provided using the hardware priority queue presented in chapter 4. Obviously, this mechanism must be provided with a deadline or priority for each executable microthread. On Anaconda this is contained in the lower 48 bits of each microthread's first parameter. Deadlines are measured in $1\mu s$ intervals so the 48 bits allows deadlines for up to 8.9 years before roll-over. It is assumed that the computer will be rebooted before roll-over occurs.

The scheduler is also closely linked with preloading data and code before execution — see section 6.8 for details.

6.5 Memory structure

Anaconda uses the memory tree structure presented in chapter 5. Anaconda's data driven microthreads model allows concurrency to be used to tolerate the latency of the memory tree.

The memory tree uses a page based capability protection system. To perform memory accesses, the appropriate capabilities must be sourced: one for a store and two for a load request (one for the data read and one for the write to the matching store). Anaconda sources capabilities from one of two slightly specialised registers: home and temporary (`hc` and `tc` respectively — see figure 6.6).

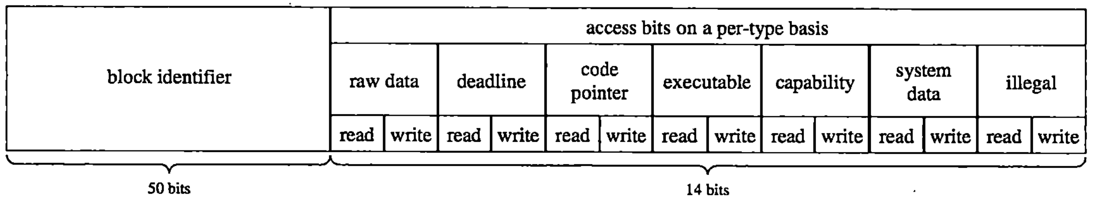


Figure 6.7: Anaconda capability format

Capabilities consist of a *block identifier* which can be considered as the key for accessing a block of pages (see figure 6.7). There are also load and store enable bits for the various data types. The next section discusses how types are used.

6.6 Exceptions and types

Anaconda supports user and supervisor modes. Most code, including much of the operating system, can be executed in user mode. Supervisor mode is provided to allow a trusted software nanokernel (see section 6.9) to support capability generation, low level exception handling and other basic house keeping functions, which require protection mechanisms to be turned off. Entering supervisor mode is achieved either when an exception occurs or if the home capability of the executing microthread has the *system capability* type.

Anaconda has 8 basic data types (see figure 6.8). A 3 bit tag is added to every quadword — in the memory, matching store and register file — to identify its type. In supervisor mode, any type of data may be loaded, stored and modified. When in user mode, however, some operations are totally prevented and others are conditional upon having the correct capability including the correct load/store bits for that type. It should be noted that user capabilities and system data may be loaded and stored but may not be modified. This inhibits forging of capabilities and system data.

The type of a quadword may be set using the `type` instruction (further details in the next section). The result type from a dyadic operation is the type of the input operand with the most *significance* (i.e. the one with the highest number — see figure 6.8). However, if a computation error occurs (e.g. due to a division by zero) then the *illegal* type is returned. Operations which attempt to use an *illegal* datum cause an exception at the register fetch stage. Thus, all exceptions can be indirectly attributed to a particular instruction (the exception is said to be *precise*[1]) but operations which cause er-

[1]Imprecise exceptions cannot be attributed to a particular instruction. For example, on the Alpha 21064, if a floating point divide is issued followed by a floating point multiply, then the divide can cause an exception after the multiply has completed because divides take much longer. Unravelling out of order completion is complex, so instead, an imprecise exception is raised.

type		user privileges			supervisor privileges		
significance	name	load	store	modify	load	store	modify
0	raw data	β	β	α	α	α	α
1	deadline	β	β	α	α	α	α
2	code pointer	β	β	α	α	α	α
3	executable	β	β	α	α	α	α
4	user capability	β	β	–	α	α	α
5	system data	β	β	–	α	α	α
6	illegal	β	β	–	α	α	α
7	system capability	–	–	–	α	α	α

where	α	=	can always perform
	β	=	can perform given the correct capability
	–	=	can never perform

Figure 6.8: Anaconda types and their privileges

rors are only detected when the result is used, i.e. the error is detected late. Whilst this mechanism does provide slightly more information than imprecise exceptions, similar software techniques are still required to respond to an exception in a meaningful manner. However, this approach does allow very long pipelines to be used (e.g. for floating point divides) with the knowledge that once the instruction has been issued, it will not cause an exception although it may produce an *illegal* result. As will be explained fully in the next section, many instructions can write their results through to the matching store. Thus, for example, a floating point divide, which writes its result to the matching store, could be issued down a very long pipeline and a new microthread could be scheduled before it has finished. If a conventional imprecise exception mechanism were used (e.g. that used by the Alpha) then errors detected later on in the pipeline could occur after several context switches have taken place. This would make sensible exception handling very complex.

When operating on floating-point numbers, Alphas cause exceptions to deal with unusual floating-point problems rather than making the hardware more complex. This is particularly true with respect to some of the more esoteric aspects of the IEEE floating-point standard, e.g. handling infinity. On Anaconda, such cases may simply be dealt with by issuing the floating-point instruction followed by a branch which is conditional on the result being invalid (`bil` — see the next section) and the appropriate test of the floating point condition codes register to determine what went wrong.

Parts of the processor expect certain types and cause exceptions if the types are not correct. When a microthread is preloaded, its deadline and code pointer are checked to ensure they are of the correct type. Capability registers are type checked when used. Checking the ability to store data of a particular type, given a particular capability, is performed by the processor. However, checking that loading data of a particular type is allowed is performed by the memory modules but instead of directly causing an exception, an *illegal* type is returned so that an exception occurs indirectly. An example use for types and capabilities is presented in section 7.7.

6.7 Instructions

Anaconda instructions are based upon the Alpha instruction set [61] because:

1. choosing a commercial instruction set facilitates comparisons[2];
2. it is a modern RISC instruction set;
3. there are sufficient gaps in the instruction set to allow additional Anaconda instructions.

The only disadvantage with Alpha is that the integer and floating point register files are separate. This does not fit well with the matching mechanism because activation frames would have to be partitioned into floating point and integer parts to map onto the appropriate register files. In practice this would result in the size of the activation frame being increased to ensure that sufficient integer parameters could be passed between microthreads (this is born out by the results in chapter 7). However, a larger activation frame would make poorer use of the match cache and increase the amount of work to perform a context switch. Therefore, Anaconda has a unified register file.

6.7.1 Load, store and write through instructions

Load and store instructions each come in two basic forms: memory or matching store; and two data lengths: longword or quadword (see figure 6.9).

The mapping between activation frames and the register file means that registers `r0...r15` have their equivalent position in an activation frame. To assist the transfer of intermediate results between microthreads, all integer and floating point modification operations are given an alternate form which writes the result through to the matching store using the current activation frame address[3] and the destination register number as an offset. The home (default) capability is automatically used. For example, the *write through* to the matching store version of the bitwise `or` instruction:

[2]As will be seen later on, to facilitate comparison, not only is the Alpha instruction set used but also the Alpha 21064's pipeline [20].

[3]The current activation frame is the address in the `af` register — see figure 6.6.

Instruction		Meaning
`ld`*x*	`r,off(addr),cap`	Load from address `off+[addr]` using source capability `cap`, wait for a memory response and place the result directly into the destination register `r`. N.B. *because this instruction stalls, it is only available in supervisor mode for system debugging.*
`ld`*x*`_req_m`	`r,off(addr),cap`	Post a load request from source address `off+[addr]` using source capability `cap`. The destination address is set to the matching store at the current activation frame address with the destination register indicating the offset within the frame (i.e. `r*8+[af]`). The home capability, `[hc]`, is used as the destination capability.
`st`*x*	`r,off(addr),cap`	Store to memory the data in register `r` at address `off+[addr]` using capability `cap`.
`st`*x*`_m`	`r,off(addr),cap`	Store to the matching store cache the data in register `r` at address `off+[addr]` using capability `cap`.

where	[*r*]	indicates a dereference of register *r*
	x	is either `l` or `q` to indicate longword or quadword accesses respectively

Figure 6.9: Load and store formats

`or_m r3,r4,r5`	writes the result of (`r3` OR `r4`) to register `r5` and also to the equivalent position in the matching store — the address being calculated from the current activation frame base (`af`) and the quadword offset of 5 to map to register `r5`, i.e. address `5*8+[af]`

6.7.2 Integer and bitwise instructions

Anaconda supports all Alpha integer and bitwise instructions (see [61]). However, on Anaconda, all of them have an alternate form which write the result through to the matching store.

There is one additional integer operation, `ldap`, to allow presence bits to be set. This extends the load address (`lda`) group of instructions (see figure 6.10). Of course each of these three instructions can also be augmented with `_m` to indicate a write through to the matching store.

Instruction	Meaning
`lda rd,off(rs)`	`[rd] = off + [rs]`
`ldah rd,off(rs)`	`[rd] = (off << 16) + [rs]`
`ldap rd,off(rs)`	`[rd] = (off << 48) + ([rs] & ((1<<48) - 1))`

where	[*r*]	indicates a dereference of register *r*
	`a << b`	means that `a` is shifted left by `b` places
	`a & b`	means the bitwise operation `a` AND `b`

Figure 6.10: Load address group of instructions

6.7.3 Floating-point instructions

Only the T-type (64 bit IEEE) floating point (see [61]) is currently supported because the register format corresponds to the memory format so an integer load (`ldq_req_m` — see below) may also be used to load T-type numbers. The other three Alpha floating point formats could be supported but would require some method for converting between the compact memory format and the verbose register format. This could be supported by appropriate conversion instructions or by extending the type system (see section 6.6) so that the conversion happens automatically.

As with integer operations, floating-point instructions may also be suffixed with '_m' so that they write through their result to the current activation frame. As will be seen in section 6.11, this is particularly useful for long latency operations, like floating-point, because it allows microthreads to be descheduled before the instruction has completed. Without this facility the pipeline would have to stall to wait for the operation to finish before a store instruction could write the result to the matching store.

6.7.4 Branch, jump and descheduling instructions

Branch and jump instructions take the usual Alpha form but cause an exception if the microthread code boundary is breached. Intermicrothread branches, jumps and function calls are performed in a more indirect manner by setting the appropriate code pointer value for the next microthread in the sequence. In the case of a jump, this may be performed by writing the jump address to the code pointer in the appropriate microthread's activation frame using a store (`stq_m`) instruction (see figure 6.9).

Descheduling a microthread may be performed using the `next` instruction. This is actually a specialisation of `xr r,offset`[4] which writes the `[PC]+offset*4` address value[5] through to the current activation frame at the `r` position. Thus, the `xr` instruction effectively supports intermicrothread PC relative branches. There are other instructions in the `xr` family to support the intermicrothread conditional branches (see figure 6.11) and `xsr` which writes the branch address through to the activation frame but, unlike `xr`, does not cause a reschedule. Thus, one possible intermicrothread equivalent of a subroutine call is:

```
xsr  ra,2                    # write through the return address
xr   cp,routine_offset       # write the jump address through...
                             # ...to the next microthread
```

To allow the detection of data with the `illegal` type, the branch if illegal (`bil` and `xil` and branch if legal (`bll` and `xll`) instructions are added. This replaces the usual exception mechanism found on the Alpha (see section 6.6).

6.7.5 Type modification instruction

To allow explicit type modification a `type` instruction is added. Like other operate instructions, `type` has two source operands, one of which may be a constant. The source

[4]`next` ≡ `xr zero,0`

[5]PC offsets are given in instruction units and since instructions are 4 bytes long the offset must be multiplied by 4 to give a byte address.

Instruction		Meaning
xsr	r,off	Write the address [PC]+off*4 through to the current activation frame with quadword offset r.
xr	r,off	As xsr but also results in a reschedule.
x*cc*	r,off	Integer conditional version of xr which tests [r] using condition *cc* and if true writes [PC]+off*4 through to the current activation frame at the code pointer position cp.
fx*cc*	r,off	Floating-point version of x*cc* although *cc* cannot be BC or BS.

where [*r*] indicates a dereference of register *r*

cc is a condition:

EQ	equal to zero	floating point conditions	integer conditions
LT	less than zero	floating point conditions	integer conditions
LE	less than or equal to zero	floating point conditions	integer conditions
GT	greater than zero	floating point conditions	integer conditions
GE	greater than or equal to zero	floating point conditions	integer conditions
IL	data type is illegal	floating point conditions	integer conditions
LL	data type is legal	floating point conditions	integer conditions
BC	bit 0 is cleared		integer conditions
BS	bit 1 is set		integer conditions

Figure 6.11: Intermicrothread branch instructions

parameters are OR'd together and the lower 3 bits are used to set the type of the destination register. An exception is raised if in user protection mode and an attempt is made to convert the type of a register to one which is only available in supervisor mode (see section 6.6).

6.7.6 Instruction formats

The instruction formats and opcode summary are in appendix B.

6.8 Cache control and preloading context

The matching store cache is fully associative to ensure maximal use. Full associativity also enables easy prediction of the number of available entries, thus permitting static analysis of usage patterns so that real-time guarantees can be met. Whilst a fully associative cache is more complex than a direct mapped one, it need not be slow if it is pipelined (see figure 6.12). In this figure the associative lookup is separated from the Cache RAM. The associative lookup could be divided into further pipeline stages if required.

Stores into the match cache (see figure 6.12) can come from two sources: the processor and the memory system. These requests are FIFO buffered and are handled in round-robin order. In the unlikely event that either of the FIFOs become nearly full then the processor and memory are stalled. It is important that a stall is issued before a FIFO is completely full because it will take several cycles before the stall takes effect (see section 6.11 for details about the execution pipeline). However, it should be noted that stalling the processor is very serious for hard real-time systems and should be considered as a real-time error. Fortunately, in practice it is very difficult to overflow the FIFOs and code which could cause overflow can easily be identified statically.

Addresses of stores to the matching store are looked up in the associative memory and if a miss occurs then the cache miss logic is activated which fetches the appropriate activation frame from the memory. The cache victim is selected according to the following rules:

- if an empty cache line exists then use it;
- failing that, select a cache line which has not been issued to the scheduler on a *not last used* basis[6];

[6]A *not last used* cache miss policy is simple to implement and typically performs better than *random* replacement [54]. A *least recently used* policy would be advantageous but it is hard to implement [54].

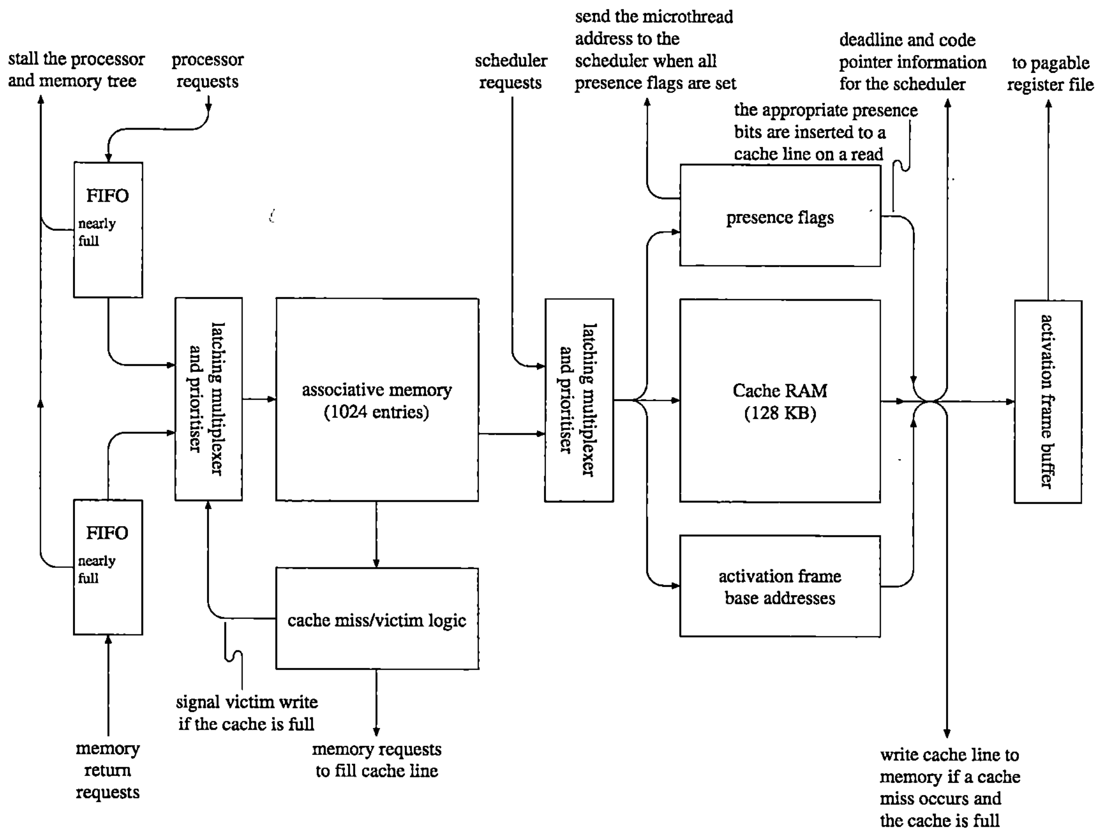

stall the processor and memory tree
processor requests
scheduler requests
send the microthread address to the scheduler when all presence flags are set
deadline and code pointer information for the scheduler
to pagable register file
the appropriate presence bits are inserted to a cache line on a read
FIFO
nearly full
presence flags
latching multiplexer and prioritiser
associative memory (1024 entries)
latching multiplexer and prioritiser
Cache RAM (128 KB)
activation frame buffer
FIFO
nearly full
activation frame base addresses
cache miss/victim logic
signal victim write if the cache is full
memory return requests
memory requests to fill cache line
write cache line to memory if a cache miss occurs and the cache is full

Figure 6.12: Overview of the Anaconda matching store

- if all the cache lines have been scheduled then an exception is generated and the nanokernel removes activations from the scheduler and deallocates them from the match cache (see section 6.9).

Under normal operation, over filling the cache should be avoided since it will be detrimental to the real-time characteristics. In order to keep the cache clean, activations may be explicitly deallocated under software control by writing all 1's to an activation frame's deadline/presence bits. This sets the deadline to infinity[7] and sets all the presence bits to full. More importantly, this information can easily be passed to the cache miss logic so that the appropriate cache line may be queued for reuse.

A hit in the associative store produces a cache line number which is then passed onto the next stage (see figure 6.12). The data is written to the cache RAM and the appropriate presence bit is set. In the case of a store to the deadline/presence bits then the presence bits are written into the presence flags store. The base address for the activation frame[8] is also passed to the next stage and stored so that later on the base address

[7]The scheduler hardware assumes that a deadline of all 1's is infinity.

[8]The base address can be computed by masking off the bottom 7 bits of the store address.

may be determined given a cache line number.

When all the presence bits are set (i.e. the activation frame is full) the scheduler is notified of the appropriate cache line, requests the deadline and code pointer information. If the deadline is infinity (all 1's) then the activation frame has been deallocated so the microthread is not scheduled. Assuming the deadline is not infinity then the code prefetch is invoked. Each cache line in the match cache has a partner in the code prefetch cache so the same cache line number may be used as an index. Messages sent to the memory to fetch the code use the cache line number and offset as the return address, and use a system capability with a zero block identifier to indicate that the data is to be consumed by the code cache. This capability also has a *system* type which turns off memory protection (see section 6.6).

A completed code prefetch informs the scheduler which reads the deadline information and inserts it into the hardware priority queue along with the cache line number. Scheduling a thread for execution involves reading the appropriate activation frame and code cache lines, in parallel, into two buffers. The activation frame is then transferred into one page of the register file and the code into an execution buffer. Thus, when execution commences all the data and code are available locally.

6.9 Nanokernel support

The nanokernel is much smaller than a traditional microkernel because it only intercepts exceptions and initiates threads, which make up the microkernel, to actually handle exceptions. The following basic exceptions must be supported:

1. activation frames which do not have correctly typed deadline and code pointers when they are scheduled;
2. exceptions caused by user level microthreads violating type mechanisms;
3. illegal instructions;
4. direct branches beyond a microthread boundary;
5. any *out of time* signals from a watchdog timer which ensures that microthreads do not exceed their allocation of 32 processor cycles;
6. software traps;
7. cache overflow because all the match cache lines have been scheduled.

The first five exceptions simply cause an operating system thread to be initiated to clean up the mess.

Software traps (`call _pal` on the Alpha) are not normally required because user level calls to the operating system may be performed using an interdomain remote procedure call (see section 7.7 for an example). However, `call _pal` is still supported for reporting serious failures (e.g. the inability to send a message to the operating system).

Cache overflow, due to all the matching store cache lines being scheduled, is a drastic situation which arises when there are too many runnable microthreads. This should be avoided by careful code analysis. However, to prevent the system locking up the nanokernel can deallocate some of the cache lines. For example, the last two elements in the hardware priority queue (those with a distant deadline) could be made accessible to the nanokernel code. These elements are then removed from the priority queue and the corresponding activation frames are deallocated from the matching store cache. In their place, one microthread is generated and pointers to the removed microthreads are stored in its activation frame. The signal microthread is given the least deadline of the deallocated microthreads and when it eventually executes it will reallocate the microthreads. Thus, two microthreads are replaced by one. This principle could easily be extended to allow, say, eight microthreads to be deallocated with one replacement.[9]

A small part of the instruction prefetch and activation frame caches may be allocated to support the nanokernel to ensure that the required data and code are available locally.

6.10 Input, output and timers

Input and output (I/O) devices are memory mapped and physically positioned at the leaves of the memory tree at the same level as the memory modules. Memory protection is also applied to I/O devices to prevent unauthorised accesses.

Input may be mapped directly to an awaiting microthread by sending messages up the memory tree to the appropriate activation frame in the matching store. Figure 6.13 illustrates the basic sequence of data transfers between the processor and input device. An output device may be controlled in a similar manner — output device control and data being written directly to the memory mapped device with completion signals being sent from the output device to an awaiting activation frame.

This approach to handling I/O ensures that all messages to the processor (input data and output signals) are synchronised to the appropriate microthread by the matching

[9]The maximum number of microthreads which could be deallocated is 13 because their activation frame addresses must be stored in an activation frame and 3 quadwords of the activation frame must be used to store the deadline, code pointer and system capability.

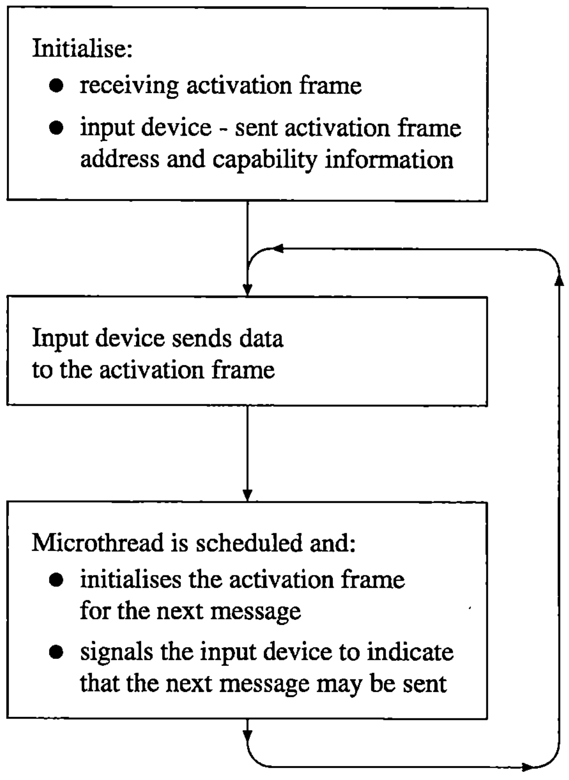

Initialise:
• receiving activation frame
• input device - sent activation frame address and capability information
Input device sends data to the activation frame
Microthread is scheduled and:
• initialises the activation frame for the next message
• signals the input device to indicate that the next message may be sent

Figure 6.13: Data transfer sequence to an input device

store and prioritised using the hardware scheduler. Note that no interrupts are required and the execution unit is not bothered by low priority I/O when dealing with high priority I/O. Furthermore, if the input protocol in figure 6.13 is used then when the processor is overloaded, low priority I/O will be delayed and potentially low priority input will be discarded when the input device's buffer overflows. This is desirable behaviour because it allows high priority messages to get through and prevents low priority messages from needlessly consuming memory tree bandwidth.

Timers are also required to ensure that output is produced on time and not early. If very tight timing is required then this may be supported by a particular output device. However, most timing is supported by a timer device which makes use of the hardware priority queue (see chapter 4). Timer requests consist of a time and an address in the matching store which is inserted into the priority queue. The first element of the priority queue is compared with the current time and at the appropriate moment it is removed from the queue and a message is sent to the matching store.

A processor watchdog timer is also required to ensure that each microthread does not consume more than 32 cycles (see section 6.9).

6.11 Execution unit pipeline

Anaconda's execution unit pipeline is similar to Alpha's (see figure 6.14 and [20]). To assist evaluation, the Anaconda pipeline has exactly the same temporal characteristics as the Alpha 21064 [20] for integer, floating-point and branch operations. Therefore, all the data feed-forwarding that Alpha performs is also performed by Anaconda although this is not shown in figure 6.14. The rest of this section primarily explains the Anaconda modifications to the pipeline and, due to space and copyright limitations, presents little

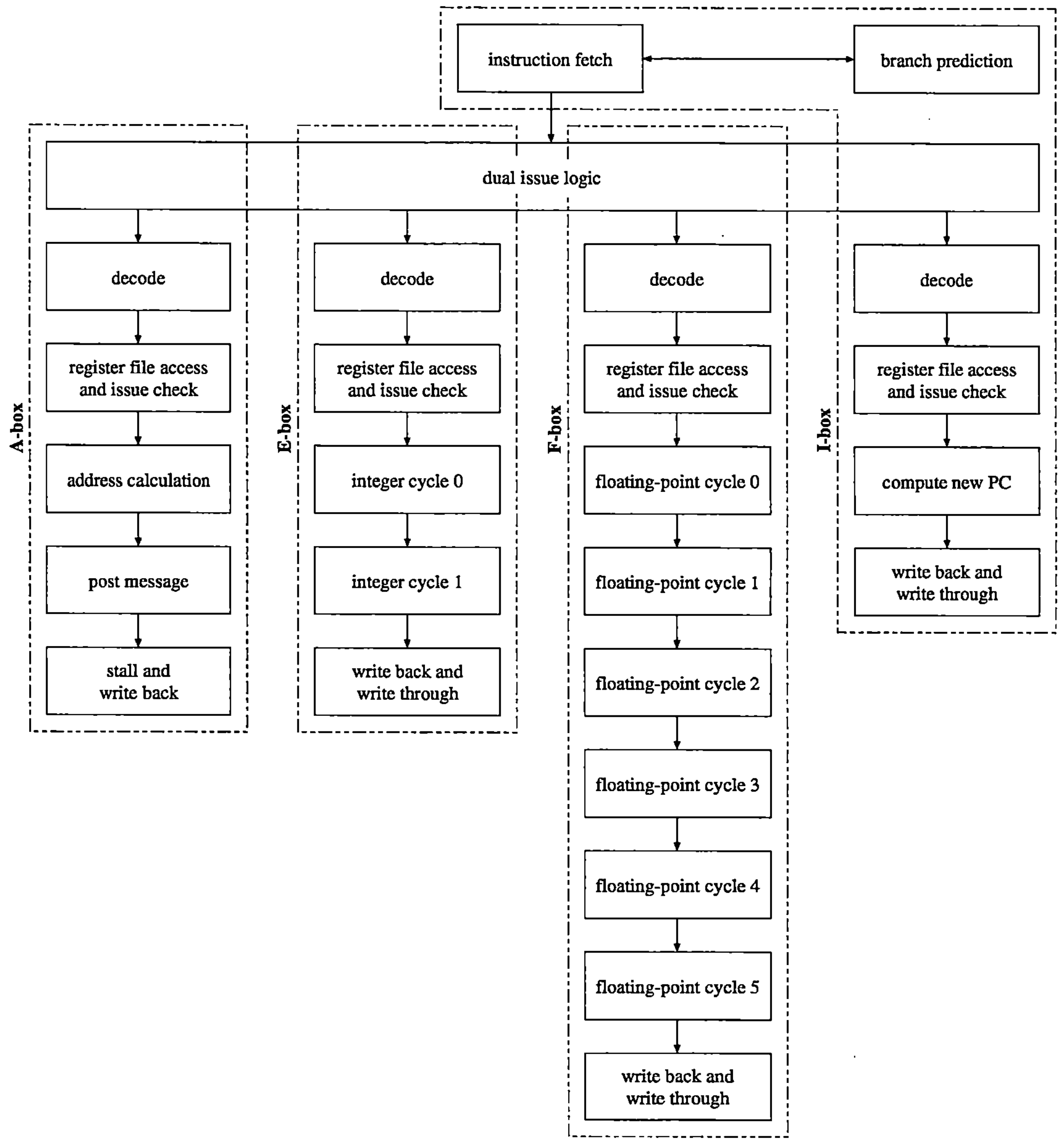


Figure 6.14: Overview of Anaconda's execution unit pipeline

detail on the Alpha (see [20] for details).

The execution unit consists of four pipelines: address calculation (A-box), integer execution (E-box), floating-point execution (F-box) and instruction fetch (I-box). Two quadword aligned instructions are fetched at once and, where possible, the dual-issue logic attempts to issue both instructions down separate pipelines. If only one instruction can be issued then the next instruction will also be singly issued.

The branch prediction logic is provided but for conditional branches only static branch prediction is supported on Anaconda to assure predictable behaviour. Thus, as for Alpha, conditional branches which branch forwards are assumed not to be taken and those going backwards are assumed to be taken. Of course nonconditional branches and jumps must always be taken. Further down the instruction pipeline it is determined whether the branch has been correctly predicted, and if not then incorrectly fetched instructions are removed from the pipelines and instruction fetch is restarted at the correct position.

As with Alpha, stages of the pipelines up to and including the issue check are all static and may be stalled. However, once an instruction passes the issue check it proceeds without stalling until it completes. Anaconda makes one exception to this for `ld`x instructions (see figure 6.9) which may stall at the *write back* stage whilst waiting for a memory response. This instruction may only be used in supervisor mode and then only rarely. Most load and store requests are simply posted to the memory without waiting for a response.

The *write back* stage in the E,F and I-boxes has been extended so that data may be written through to the matching store. This has important implications for the efficiency of the pipelines during a context switch. When a `next` instruction reaches the *issue check* stage it is stalled until all instructions preceeding it have been issued. Then the write backs for the instructions in the lower part of the pipelines may be disabled because there are no further instructions belonging to the currently executing microthread which may read the register file again. Feed-forward paths which pass information back up the pipeline must also be disabled. Thus, the remaining instructions must store their results through to the matching store. Furthermore, the register bank for the next microthread may now be switched in and the first instructions may pass the decode stage.

6.12 Summary

Anaconda is a multithreaded processor based upon a data-driven microthread model. The primary components are (with reference to figure 6.1):

- *interprocessor communication* — using message passing via one or more I/O links situated at the leaves of the memory tree.
- *memory tree* — constructed from many small memory modules and I/O units with a scalable tree interconnect. Single address space virtual addressing and page based capability protection is provided by the memory modules. Thus, translation and protection resources are automatically added with more memory.
- *cached direct-mapped matching store* — each active microthread has an activation frame in the cached direct-mapped matching store for receiving parameters. Presence bits for each parameter are stored as part of the first parameter so may be set in a single store.
- *code prefetch cache* — for storing code for runnable microthreads.
- *scheduler* — provided by a hardware priority queue to support either fixed priority or earliest deadline first policies.
- *pageable register file* — which allows activation frames to be transferred from the cached matching store before a microthread is executed.
- *instruction fetch buffer* — which buffers code coming from the preload cache.
- *control-flow execution unit* — an Alpha 21064 styled pipeline for executing instructions.

The next chapter evaluates the Anaconda design.

Chapter 7

Evaluation

7.1 Introduction

An Anaconda assembler and simulator were constructed to allow programs to be written, executed and timed (sections 7.2 and 7.3). A memory copying routine and Livermore loop 7 were written to evaluate the code efficiency (sections 7.4 and 7.5). Mutual exclusion, signalling and remote procedure calls are then demonstrated (sections 7.6 and 7.7). Finally, conclusions are drawn (section 7.8).

7.2 Assembler

A simple assembler was built on top of C++ [64]. Thus, assembler programs are written as a sequence of C++ function calls (see, for example, the memory copying routine in appendix C). All parsing, expression evaluation and type checking is performed by the C++ compiler. Furthermore, this allows a variety of code expansion techniques using simple C++ loops or function calls.

To generate Anaconda code the assembler C++ program is compiled and when executed produces the assembler output. More interestingly, instead of a one for one replacement of the assembler instruction by the appropriate binary machine code, it is also possible to generate a sequence of native Alpha instructions to simulate each Anaconda instruction. This proves to be a very fast way of simulating an Anaconda. However, in order to assess the temporal properties of Anaconda a more detailed simulation is required.

7.3 Simulator

A massively detailed discrete event simulation could have been constructed of the implementation outlined in chapter 6. However, it was anticipated that this would be very slow. Instead, advantage was taken of the programmer's model which hides detailed timing. It is the detailed control structures of a particular implementation which affect timing rather than the functional execution. Therefore instruction execution can be separated from details of the instruction execution pipeline. So a simple instruction interpreter was constructed which feeds a stream of completed instructions into a pipeline simulation. So, the pipeline simulation only needs to model the control structures and does not need to consider data movement. For example, feed forwarding of data does not need to be performed although instruction issuing checks need to be performed to ensure that the data could have been fed forward.

The only instruction which does not execute correctly using this scheme is `rpcc` since it reads the cycle counter which is dependent upon the time at which the instruction reaches a particular stage in the address calculation unit (A-box). Rather than resolve the problem a simple debugging message is emitted by the pipeline simulation with the correct value but the code being simulated must not expect a correct value. Thus, `rpcc` may be used for timing but the answers must be read from the debugging messages. In fact the simulator has a "silent" mode where only the time between two `rpcc`s is output which is useful for producing data from repeated runs.

The model of the memory tree, matching store, instruction prefetch cache and scheduler primarily concentrate on control information. However, some address and data information is required for setting presence bits in the matching store. Capability protection and virtual address translation were not explicitly simulated although the additional latency to perform these calculations was taken into account.

Figure 7.1 presents an overview of the simulator. Various parts of the simulation may be removed and replaced by a simpler model. For example, the memory tree and instruction prefetch cache many be replaced by a simple single cycle memory simulation. This allows Alpha code to be executed as though it was running from cache. Using this simplified model the pipeline was tested using many sequences of code to ensure that the timing information obtained was identical to a real Alpha 21064 processor.

7.4 Memory copy test

A parallel memory copying routine, operating on quadword aligned data, was written to test memory bandwidth, the ability to tolerate memory latency and the overheads in managing multiple threads. The code is in appendix C and is depicted in figure 7.2. For efficiency reasons it is important that each microthread copies as large a chunk as

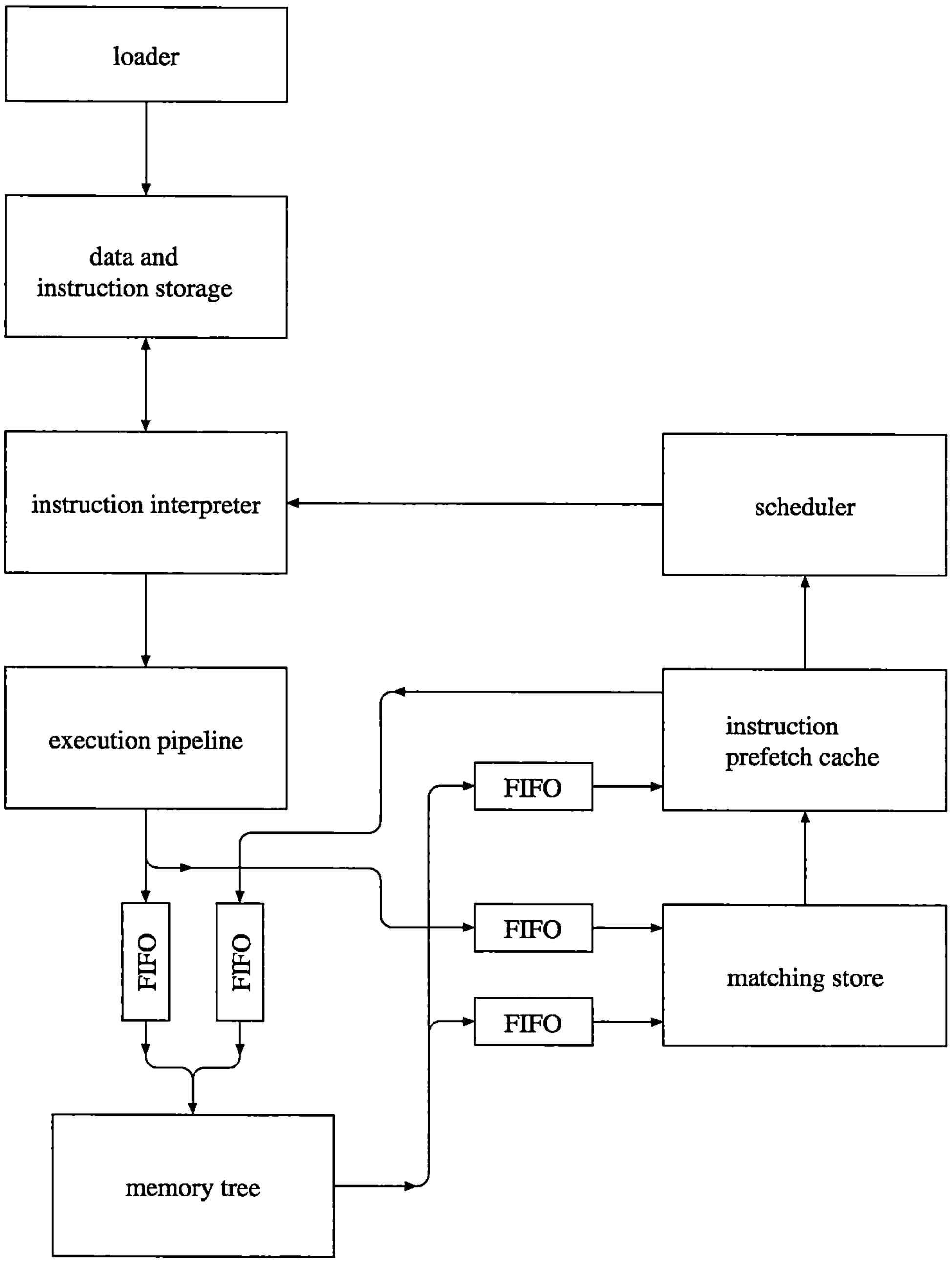


Figure 7.1: Overview of the simulator

possible. This is limited by the activation frame size and the other parameters which a copy microthread needs, in this case: a deadline, code pointer, capability, loop counter, source address and destination address. Thus, from the original 16 quadwords, 9 are available in the activation frame to perform copying so the copy loop is unfolded 9 times. Code is also required to cope with copying block sizes which are not divisible by 9; in this case a microthread for copying 3 items and another for copying one at a time. Thus, at the end of the copy up to 2 blocks of 3 quadwords and 2 single quadwords may need to be copied. This division of work is an optimal trade off between being able to make use of cached code, the number of microthreads needed to be executed and total code size. Whilst this may seem overly complex for a simple copy operation it should be remembered that an efficient implementation on an Alpha requires similar loop unfolding (typically 4 times when using the Digital C compiler).

A 64 kbyte memory copy was performed using between 1 and 10 threads with intelligent instruction fetch caching turned on or off (see figures 7.3 and 7.4). Without intelligent instruction caching, code is fetched regardless of whether it is still in the prefetch cache from the previous invocation of the microthread. Thus, additional time is spent fetching code which results in more threads being needed in order to provide sufficient work to be able to tolerate a particular memory latency (see figure 7.3). However, with intelligent caching, fewer threads are required (see figure 7.4). Figure 7.4 also demonstrates that there is little overhead in running additional threads. If there were then the execution time would begin to rise again as the number of threads were increased, so 10 threads would take longer than 6 threads.

A typical local memory latency would be around 30 cycles long (see section 5.4). Assuming that the intelligent instruction prefetch cache is used, then only 3 parallel threads are required for the copying process to become CPU, rather than memory latency, limited. At this point, 64 kbytes are copied in approximately 27 kcycles which equates to 3.3 cycles per quadword copy. This compares favourably with an Alpha 21064 processor with a fast memory system[1] which takes approximately 90 kcycles (10 cycles/quadword) the first time the copy is performed, due to TLB misses, and 64 kcycles (7.8 cycles/quadword) if the copy is repeated. The poor Alpha performance is due to cache misses. Techniques, like giving hints to the cache using the Alpha `fetch` instruction,[2] could be used but are unlikely to improve much upon Anaconda's 3.3 cycles/quadword. Furthermore, there are many problems in using hints like `fetch` — see section 5.3.1.

[1]To be precise a DEC3000/500s executing code generated by the Digital C compiler with all optimisations turned on.

[2]The `fetch` instruction has no effect on a DEC3000/500s but will probably be supported in future Digital products

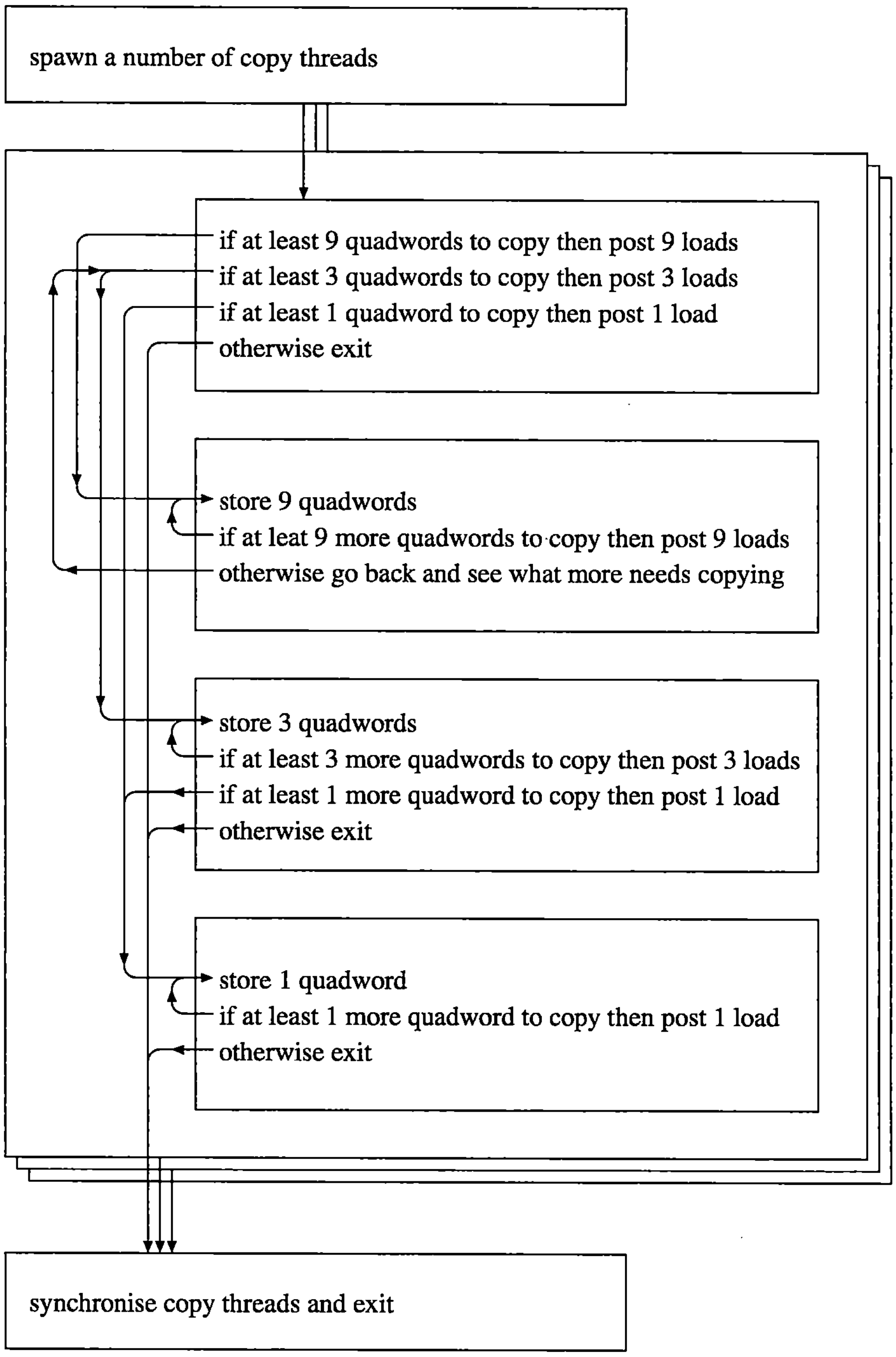

spawn a number of copy threads
if at least 9 quadwords to copy then post 9 loads
if at least 3 quadwords to copy then post 3 loads
if at least 1 quadword to copy then post 1 load
otherwise exit
store 9 quadwords
if at leat 9 more quadwords to copy then post 9 loads
otherwise go back and see what more needs copying
store 3 quadwords
if at least 3 more quadwords to copy then post 3 loads
if at least 1 more quadword to copy then post 1 load
otherwise exit
store 1 quadword
if at least 1 more quadword to copy then post 1 load
otherwise exit
synchronise copy threads and exit

Figure 7.2: Overview of the parallel memory copy routine

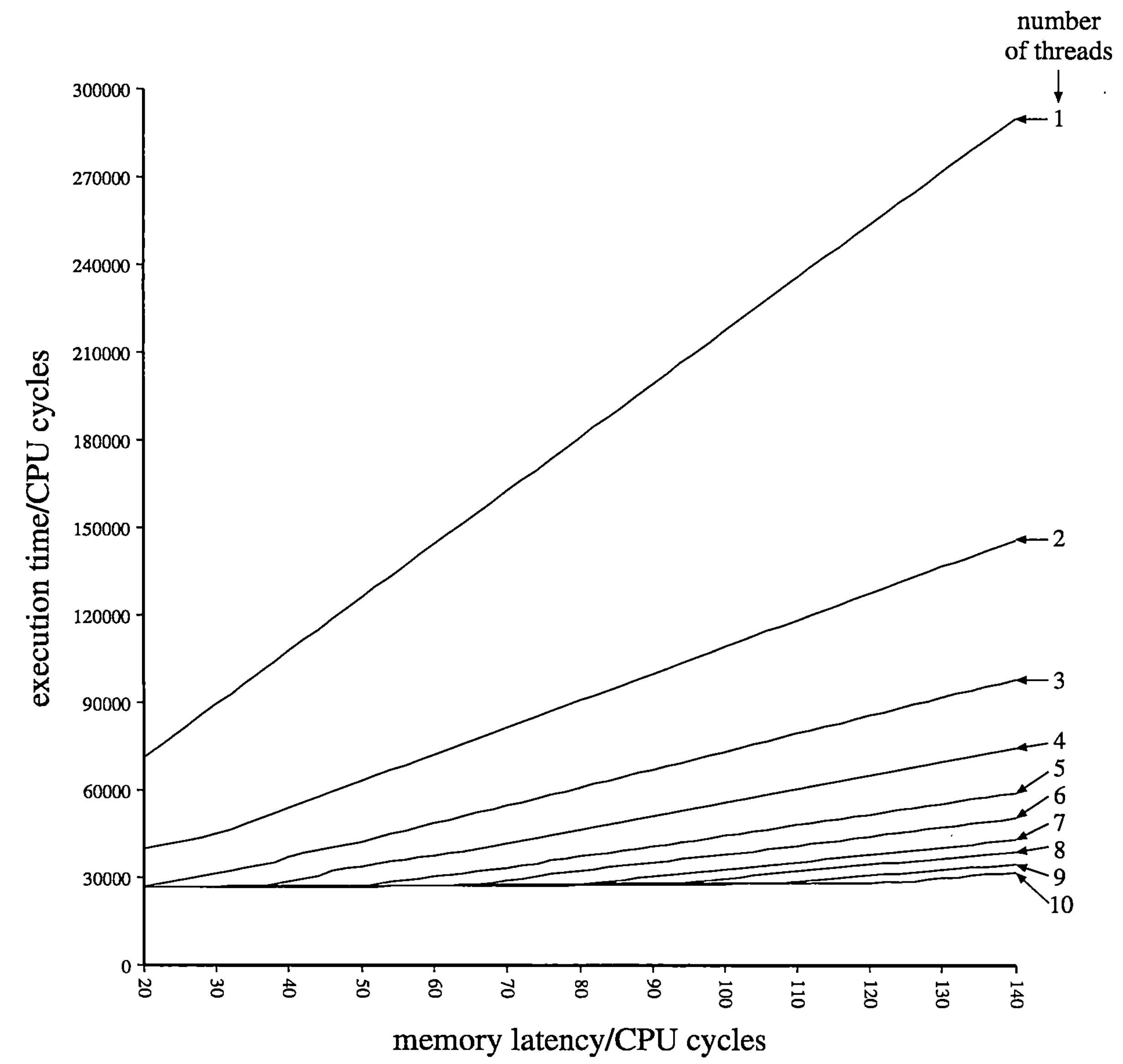


Figure 7.3: Memory copy without intelligent instruction caching

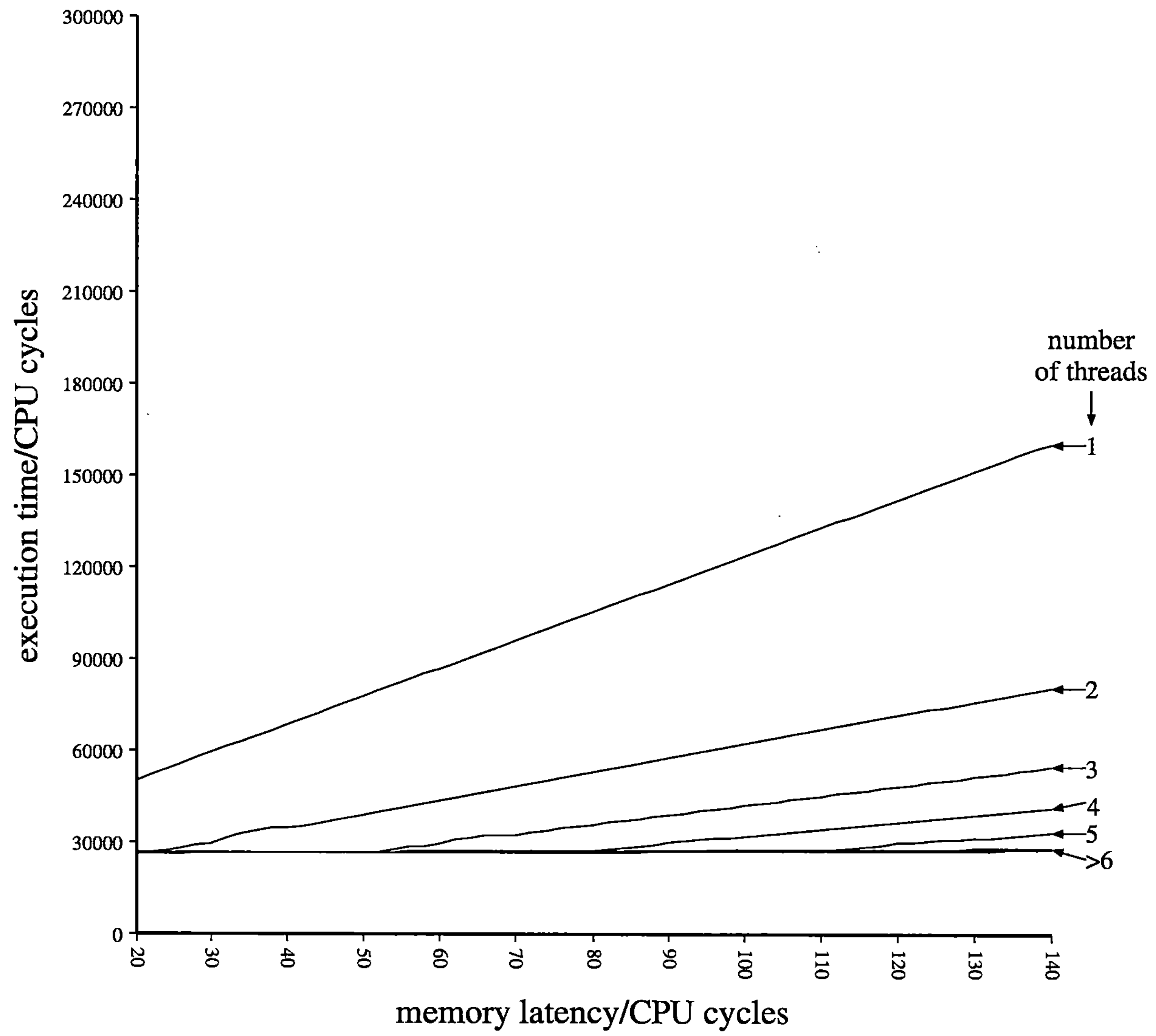


Figure 7.4: Memory copy with intelligent instruction caching

7.5 Livermore loop 7 test

Livermore loop 7 (see the code in figure 7.5) was chosen to demonstrate how an inner-loop may be unrolled to obtain sufficient concurrency to tolerate memory latency. Coding began by performing data-flow analysis of the code (see figure 7.6). From this it can be seen that each iteration of the loop shares some constants but also variables `U[k+1]`...`U[k+6]` may be used in the next iteration as `U[k']`...`U[k'+5]` where `k'` is the value of `k` for the next iteration. Unrolling the loop assists in reusing the values of array `U` and allows a larger block of code to be optimised which facilitates instruction reordering to make best use of performance. For example, a floating point add or multiply may be issued every clock cycle but takes 6 cycles before the answer is available. Therefore, it is advantageous if instructions can be ordered so that data dependencies do not stall the pipeline. Without loop unrolling the Livermore loop contains too many interdependencies to avoid stalling the pipeline. However, after loop unrolling there is sufficient freedom to avoid stalls.

For Anaconda code, a data-flow approach, using a number of concurrent microthreads, may be used. This takes advantage of the matching store for intermediate results from a large number of loop unrolls and facilitates intelligent instruction cache. In this case the Livermore loop was unrolled six times and the resulting data-flow graph was regrouped as nine intercommunicating microthreads (see figure 7.7) in a static data-flow manner (i.e. with data going forward and completion signals going backward).

During execution several iterations of the loop tend to run in parallel (see figure 7.8). However, according to the memory latency and the scheduling policy used, different orderings appear. Furthermore, the ordering which emerges affects the amount of available work to tolerate memory latency. If there is insufficient work then lower priority (later deadline) threads may execute. To test this the program was executed with varying scheduling policies, different sizes of lower priority dummy microthreads and a range of memory latencies.

In figure 7.9, all the Livermore loop microthreads were given the same priority. The effects of background microthreads being scheduled was determined by varying their length for repeated simulations. The resultant range of execution times is depicted as

```
1007 DO 7 k= 1,n
         X(k)=      U(k   ) + R*( Z(k   ) + R*Y(k   )) +
    1          T*( U(k+3) + R*( U(k+2) + R*U(k+1)) +
    2          T*( U(k+6) + Q*( U(k+5) + Q*U(k+4)))
    7 CONTINUE
```

Figure 7.5: Fortran code for the Livermore loop 7 kernel

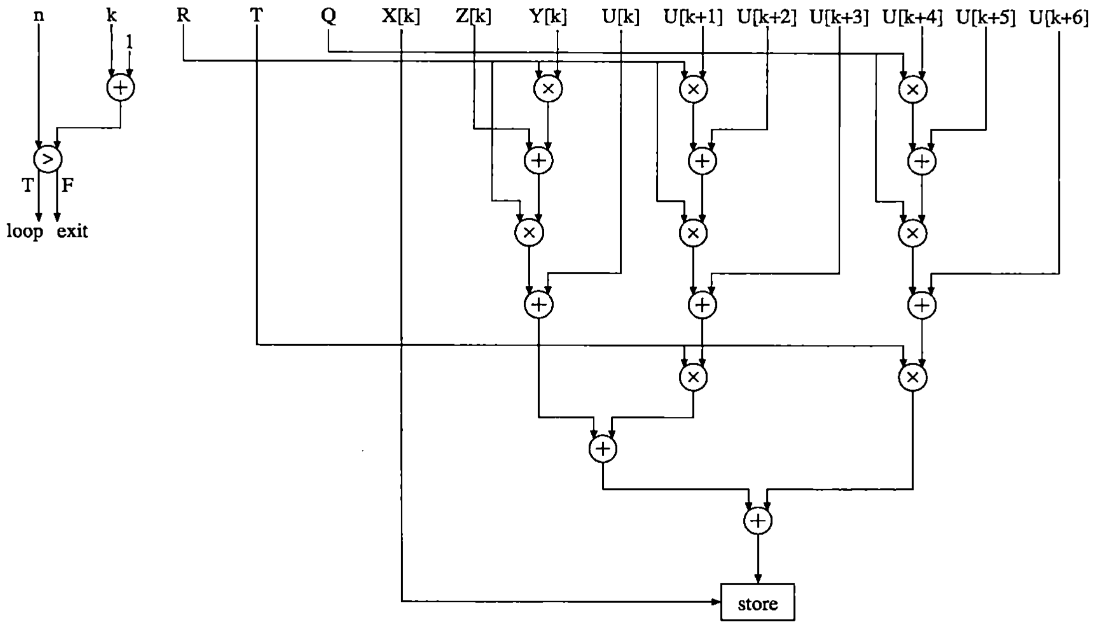


Figure 7.6: Data-flow analysis of Livermore loop 7

a gray area with the effects from dummy microthreads of length 8 and 32 cycles also being plotted as lines.

For reasonable memory latencies (less than 35 cycles) performance was CPU limited except during startup and termination hence the slight variance in execution time where dummy microthreads get chance to execute. When the memory latency reaches around 40 cycles an unfortunate ordering occurs where microthreads 1 and 2, which fetch data, do not get scheduled early enough which reduces the amount of latency which can be tolerated. Interestingly this corrects itself at around a memory latency of 50 cycles and then becomes progressively worse.

To encourage loop iterations to start as early as possible, the deadline time was reduced slightly each time around the loop. As can be seen in figure 7.10, this has little effect upon memory latency tolerance and adds overhead due to dynamically adjusting the deadlines.

Then a policy of reducing the deadline time for microthreads involved with initiating load posting was tried; in this case microthreads 0, 1 and 2. This improves memory latency tolerance considerably (see figure 7.11 and because the deadline allocation is static it has no overhead. For loops which exhibit less concurrency than Livermore 7, this technique would be valuable even if the main memory latency was short.

If an intelligent instruction cache is not used then the program exhibits poor performance (see figure 7.12). However, regardless of whether an instruction cache is used, worst case performance can still be determined, and more importantly, is typi-

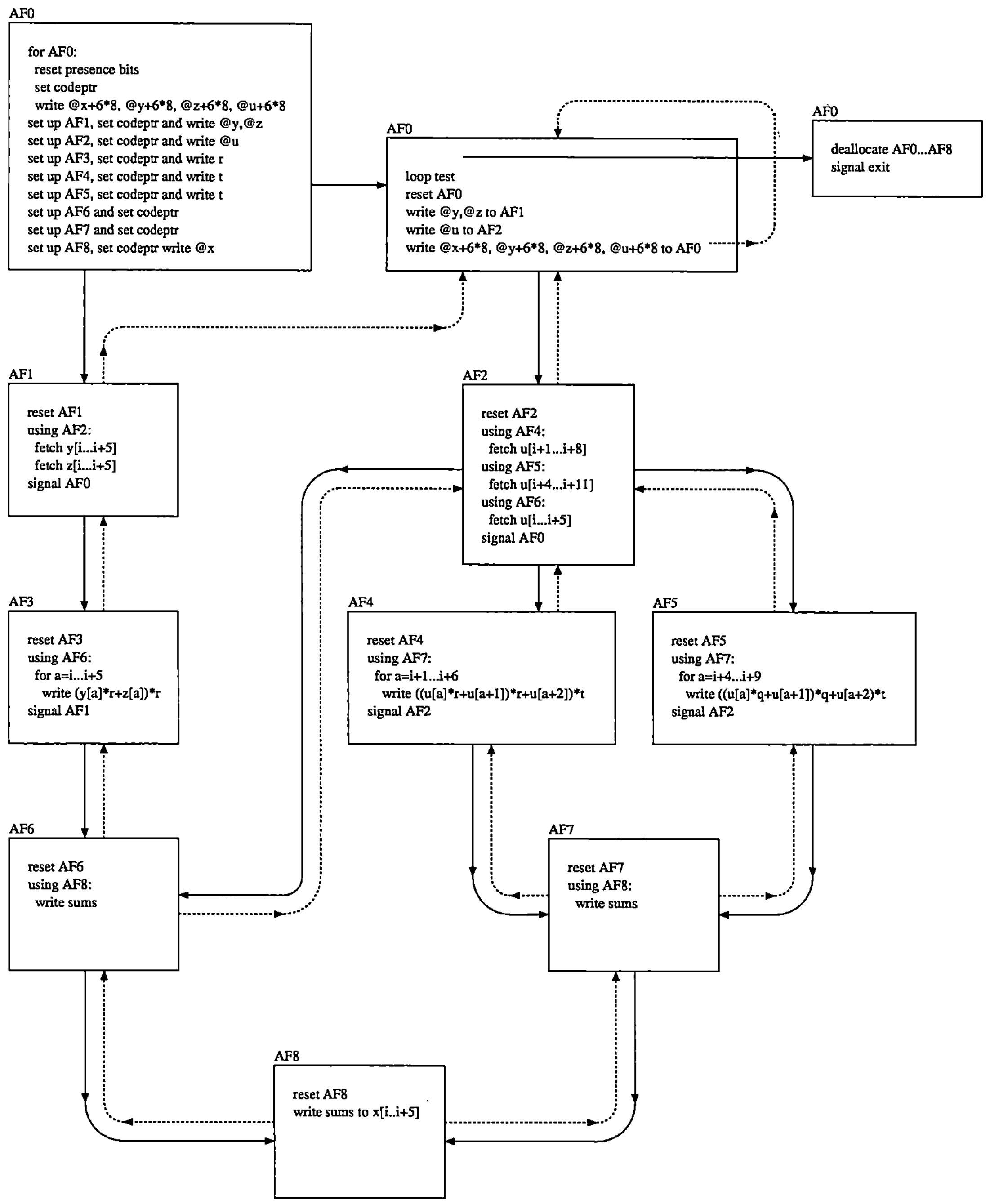


Figure 7.7: Overview of the Anaconda routine to perform Livermore loop 7

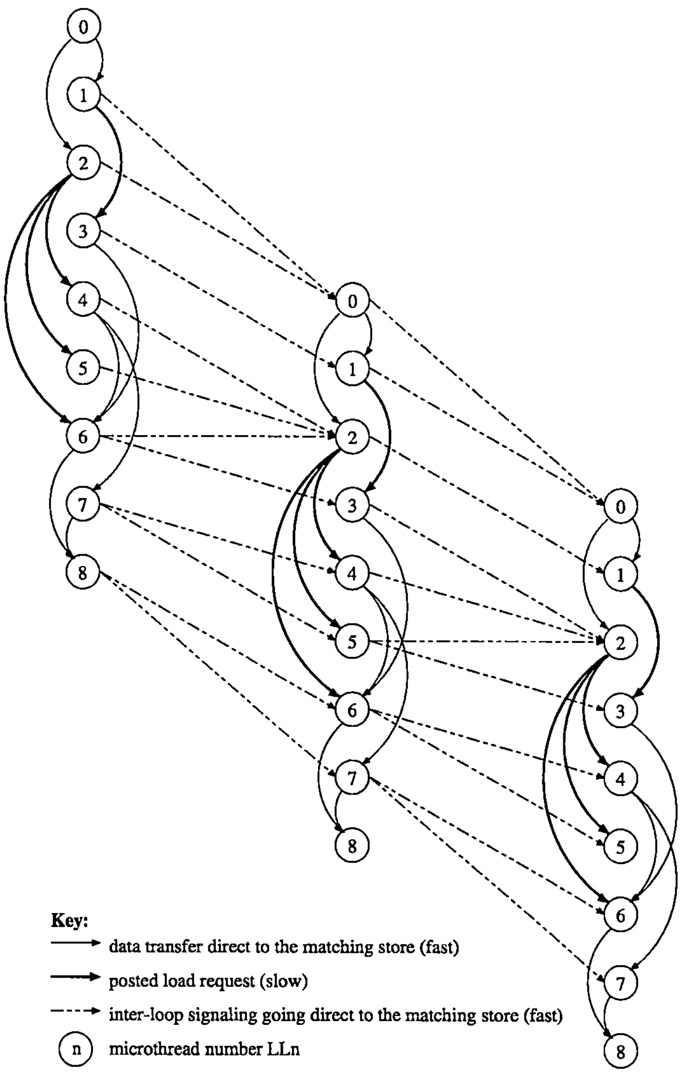

Key:
data transfer direct to the matching store (fast)
posted load request (slow)
inter-loop signaling going direct to the matching store (fast)
n microthread number LLn
0
1
2
3
4
5
6
7
8

Figure 7.8: Interdependencies between microthreads during several iterations

cally within 11% of the best case performance when executing the Livermore loop (see figures 7.11 and 7.12). Furthermore, the variance is due to other work being performed rather than the processor stalling. This contrasts with an Alpha where the cost of performing a read can be from 1 (no cache miss) to around 35 cycles (a secondary cache miss) and if a TLB miss occurs then it can take thousands of cycles. During a cache miss or TLB miss, little useful work is performed.

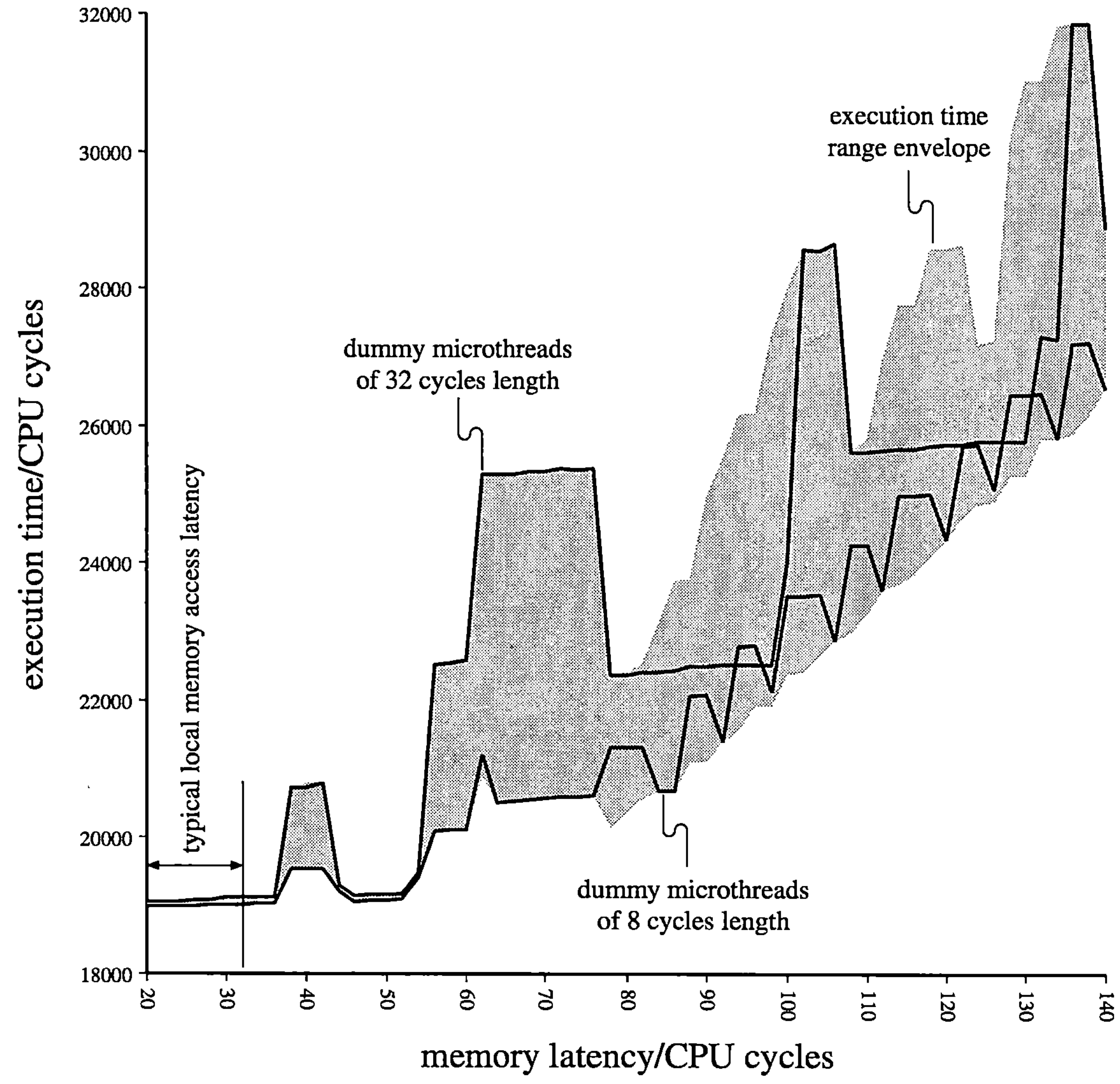


Figure 7.9: Livermore loop 7 with all 9 microthreads having the same deadline

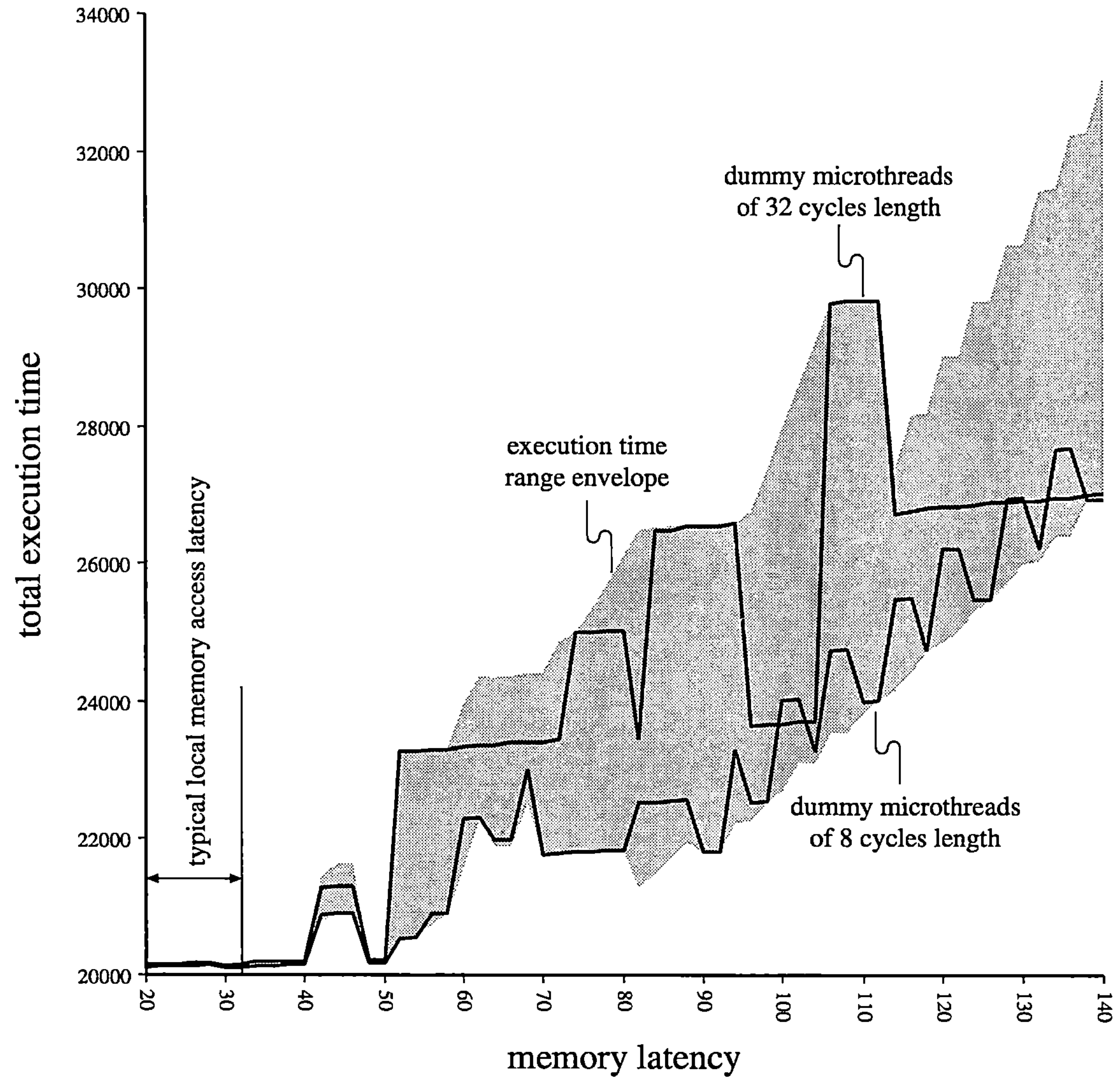


Figure 7.10: Livermore loop 7 with a decreasing deadline for each iteration

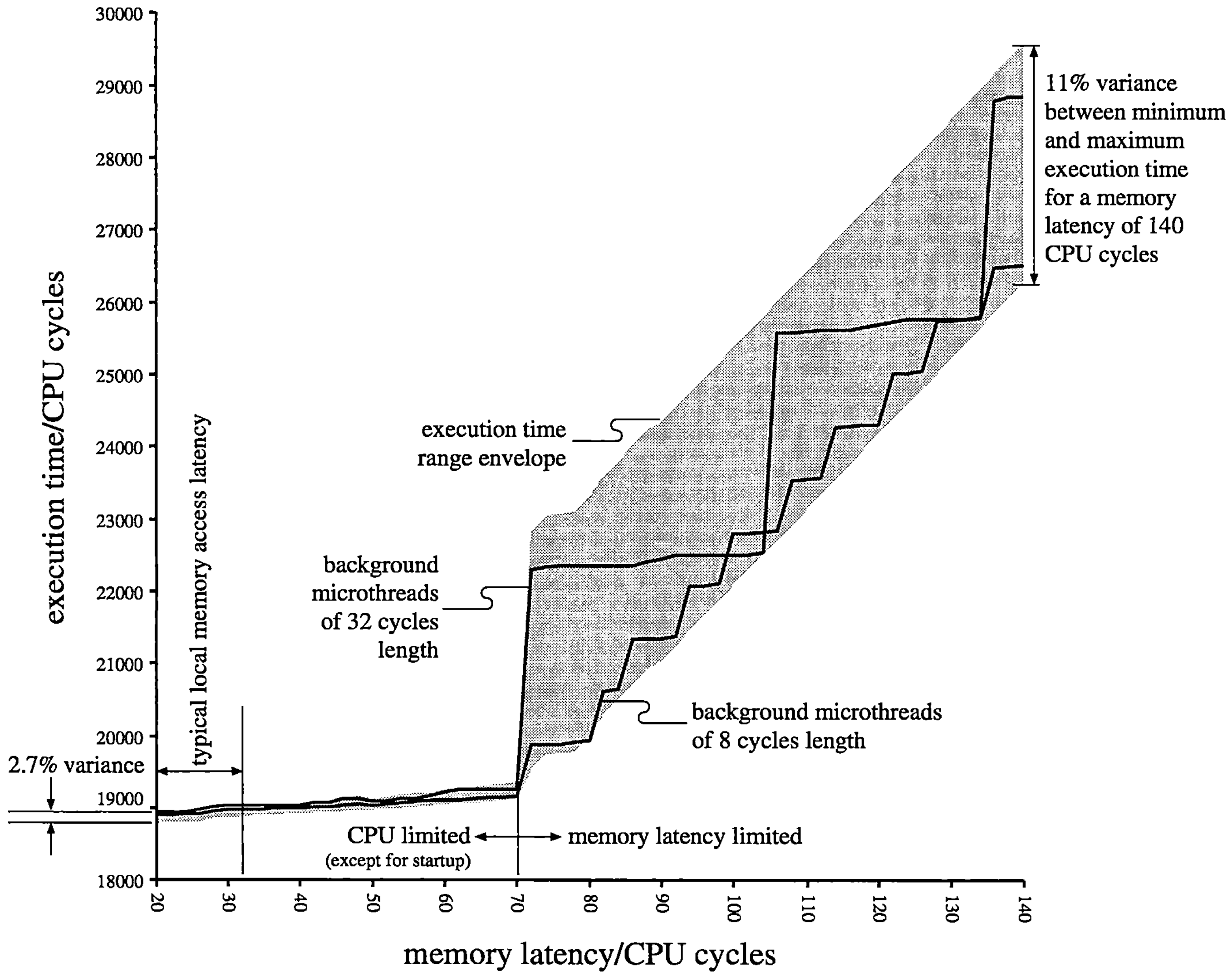


Figure 7.11: Livermore loop 7 with microthreads 0, 1 and 2 having a slightly earlier deadline

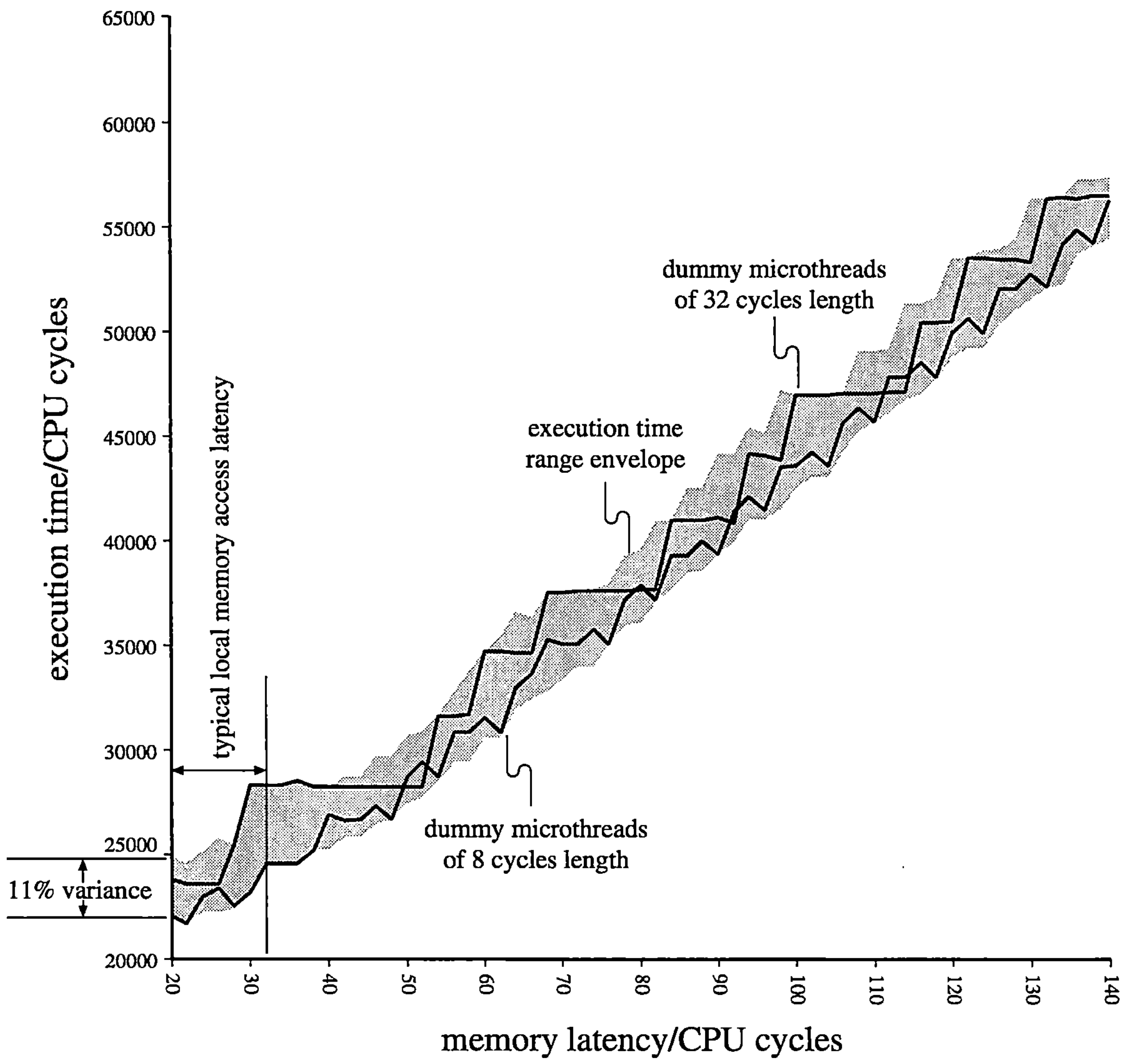


Figure 7.12: Livermore loop 7 with the same parameters as figure 7.11 but with intelligent instruction caching turned off

7.6 Signalling and mutual exclusion

Signalling a microthread is as simple as storing a value. The store, of course, is to the microthread's activation frame at some predetermined address. However, unlike a control-flow machine, there is no further software overhead since the hardware matching store actually takes care of the synchronisation and the hardware scheduler does the rest.

Whist microthreads are executed nonpreemptively, spin locks cannot be implemented by simply reading a flag, testing it and writing the new value back, because loads operate split phase, so there has to be a reschedule between the read and test. However, it should be noted that load and store requests are forced to be FIFO ordered. Thus, it is possible to request a load from a particular flag and immediately set it. The microthread which receives the load, tests to see if the flag had already been set and if it has, re-attempts to set the flag for its self.

Dijkstra's semaphores [21] may be implemented using a spin lock to provide mutual exclusion on a semaphore value. The `P` operation (wait) requires two microthreads, and the `V` operation (signal) requires three microthreads (see figure 7.13).

Monotonically increasing event counts and sequences [55] may be implemented in a similar manner to semaphores and allow a variety of synchronisation systems to be constructed, including that used by Posix threads [10]. Five primitives are required, two for sequences and three for events:

`read_sq(s)` returns the value of sequence `s`. This is just a memory read.

`ticket(s)` returns the next monotonically increasing number in the sequence and guarantees that any subsequent calls to either `ticket` or `read_sq` will return a higher value. This may be implemented using a spin lock to provide mutual exclusion whilst increasing the sequence number.

`read_ev(e)` as `read_sq` but for events.

`await(e,v)` wait until the event count e reaches or exceeds the value `v`. This may be implemented in a similar manner to the `P` semaphore operation with a circular buffer being used for the wait queue.

`advance(e,n)` increases the value of the event count `e` by `n` and signals any threads waiting for `e` to become this large. Implementation involves taking a spin lock out whilst updating `e` and scanning the waiting queue for processes to signal.

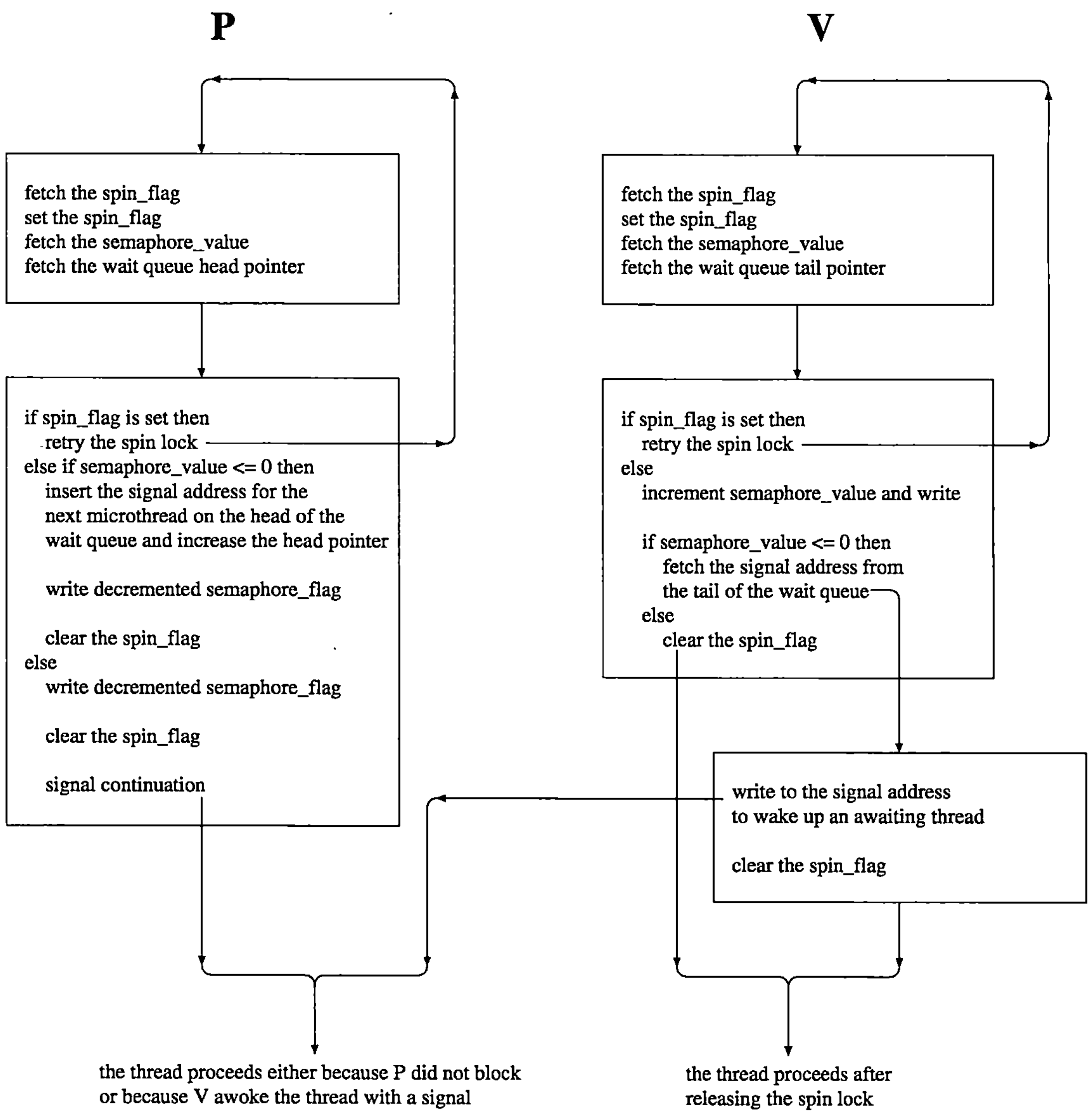

P
V
fetch the spin_flag
set the spin_flag
fetch the semaphore_value
fetch the wait queue head pointer
if spin_flag is set then
retry the spin lock
else if semaphore_value <= 0 then
insert the signal address for the next microthread on the head of the wait queue and increase the head pointer
write decremented semaphore_flag
clear the spin_flag
else
write decremented semaphore_flag
clear the spin_flag
signal continuation
fetch the spin_flag
set the spin_flag
fetch the semaphore_value
fetch the wait queue tail pointer
if spin_flag is set then
retry the spin lock
else
increment semaphore_value and write
if semaphore_value <= 0 then
fetch the signal address from the tail of the wait queue
else
clear the spin_flag
write to the signal address to wake up an awaiting thread
clear the spin_flag
the thread proceeds either because P did not block or because V awoke the thread with a signal
the thread proceeds after releasing the spin lock

Figure 7.13: An implementation of Dijkstra's semaphores

7.7 Interdomain remote procedure calls

The remote procedure call (RPC) [8] provides a convenient model for interdomain communication. One approach to implementing RPCs on Anaconda is to set up two interdomain communication channels: one for the call and the other for results. An interdomain channel is constructed from an activation frame in a page of memory which is principally owned by the receiver. The receiver initialises the activation frame with the appropriate code pointer and capability. Then the sender is given a capability which only allows it to write data with a *raw data* type. This prevents it from acquiring the activation frame's capability and code pointer, and from writing a new capability or code pointer. However, the capability or code pointer may be overwritten with raw data but this will cause an exception when the thread is scheduled. If the data to be communicated is too large for an activation frame it may be stored in the same page as the activation frame and a pointer to the data may be passed instead. Obviously, further pages may be allocated if communicating large blocks of data.

Setting up two channels to perform an RPC is the principle cost. When an application is loaded it may be given an RPC interface to the operating system (OS). The OS, or some other third party, must be involved in establishing a new RPC interface.

1. the client requests a new page and two capabilities (a general purpose one and a write-raw-data-only one) from the OS

2. the client sends a request to the OS for a connection to the server

3. the OS signals the server to set up a RPC interface and gives it the write-raw-data-only capability to the client

4. the server requests a new page and two capabilities (a general purpose one and a write-raw-data-only one) from the OS

5. the server returns the write-raw-data-only capability to the client via the OS or by using the channel to the client which has already been set up.

Once the RPC interface has been established, the only communication cost is marshalling and unmarshalling data. The capability protection mechanism, matching store and hardware scheduler perform the rest of the work without the software overhead which would be incurred on a control-flow machine. No further calls to the OS are required unless failures occur.

7.8 Conclusions

A memory copy routine was simulated which revealed that Anaconda performs poorly when executing only a single thread. However, once three threads are active, performance is very good even though a data cache is not used. This is because the matching store allows a huge number of outstanding loads between a number of concurrent threads. Even when memory latency is over 100 cycles, only a few threads are required to keep the processor busy. This has interesting implications for applications which use shared memory or message passing which have a long data latency.

The memory copy also demonstrated the low overhead in spawning and running threads. Also, intelligent caching of instructions does help and has predictable behaviour. However, without instruction caching, similar performance may be obtained provided more threads are executable. This is important when executing long sequences of code because the instruction caching will not be effective.

Livermore loop 7 was used to demonstrate loop unrolling which is essential to elicit sufficient concurrency in order to hide memory latency. However, it was noted that there was sufficient freedom, in the order in which microthreads could be executed, to allow orderings which do not hide memory latency well. A workable solution to this is to give a slightly earlier deadline to microthreads which are responsible for requesting loads. Thus, loads are requested as quickly as possible leaving sufficient work to be performed when waiting for memory responses.

Techniques, for enforcing mutual exclusion and providing communication, were discussed. Anaconda's capability protection mechanism proves to be more than adequate. Furthermore, the matching store and hardware scheduler remove many software overheads which would be incurred on a control-flow machine.

Chapter 8

Conclusions

8.1 Review

This dissertation began by presenting a commentary on current computer designs, their motivations and future directions. The vast majority of today's computers are oriented towards applications which require best effort performance. Increasingly, this is to the detriment of hard real-time and multimedia applications which require a computer to deliver temporally predictable performance. Since hard real-time and multimedia applications are becoming more common, the agenda was set to design a processor to meet their requirements. The result is a multithreaded processor design, called *Anaconda*, which not only fulfils these requirements, but also performs well for best effort applications.

The following sections present a more detailed review of the background issues and the Anaconda design.

8.1.1 Background

Control-flow and data-flow processor designs were reviewed. They represent two extremes in the design space: control-flow offers sequential execution whilst data-flow supports instruction level concurrency. Their radical differences exhibit advantages and disadvantages which often oppose each other: where one has an advantage the other is often disadvantaged.

Control-flow's strict sequential nature allows a precise ordering to be placed upon data movement. This allows efficient use of closely coupled storage (e.g. a register file). Furthermore, multiple assignment to the same address is practical which simplifies memory reuse and interaction with input and output (I/O) devices. However, such tight sequential behaviour provides little concurrency to hide the effects of long latency operations. Memory access latency is a particular thorn in the side of control-flow designs because faster processors tend to require larger memory which has a longer latency. Whilst caches are used to reduce the average access latency, it then becomes difficult to predict the temporal characteristics. Temporal predictability is also not helped by the need to provide rapid address translation and memory protection. This inevitably results in another cache (a TLB) in an attempt to localise some of the address translation and protection information. The final problem is the use of interrupts to synchronise external events to appropriate software handler code, which disrupts the execution pipeline and causes significant and unpredictable overhead.

The data-flow model supports some form of matching store to synchronise data to instructions. This is a boon to message passing both within and between processors. Furthermore, the matching mechanism allows far more concurrency to be supported by the hardware. More concurrency results in more work being available whilst waiting for long latency operations. This facilitates the use of pipelining techniques to provide a scalable memory structure at the expense of increasing access latency. However, instruction level concurrency prohibits a sensible scheduling policy and makes multiple assignment, and thus I/O, problematic.

Where the control-flow model has strengths, the data-flow model tends to have weaknesses and vice versa. This has prompted research into various amalgams of the two models in an attempt to produce more rounded designs with fewer disadvantages. These designs are broadly referred to as multithreaded processors.

One view of the design space may be obtained by plotting the amount of available concurrency supported by the hardware against the size of the nonpreemptable executable unit (see figure 8.1). Whilst this is a highly inexact representation, it does present a view of the control-flow/data-flow void and some of the processor designs which attempt to fill it. Some multithreaded designs have added a little concurrency to control-flow (e.g. $\star$T, Alewife, Transputer, D-RISC and P-RISC) and others have introduced some sequentiality to data-flow (e.g. tagged-token data flow machines like EM4 and Monsoon). However, these small additions only solve a few problems. Other designs (e.g. HEP, Tera and MDFA) are more radical and offer both control-flow and data-flow features. However, to date these data-flow/control-flow hybrids schedule very small groups of instructions and often on a per-instruction basis. This leaves too little time to perform a sensible hardware scheduling decision. Furthermore, poor use is made of a register file for intermediate results.

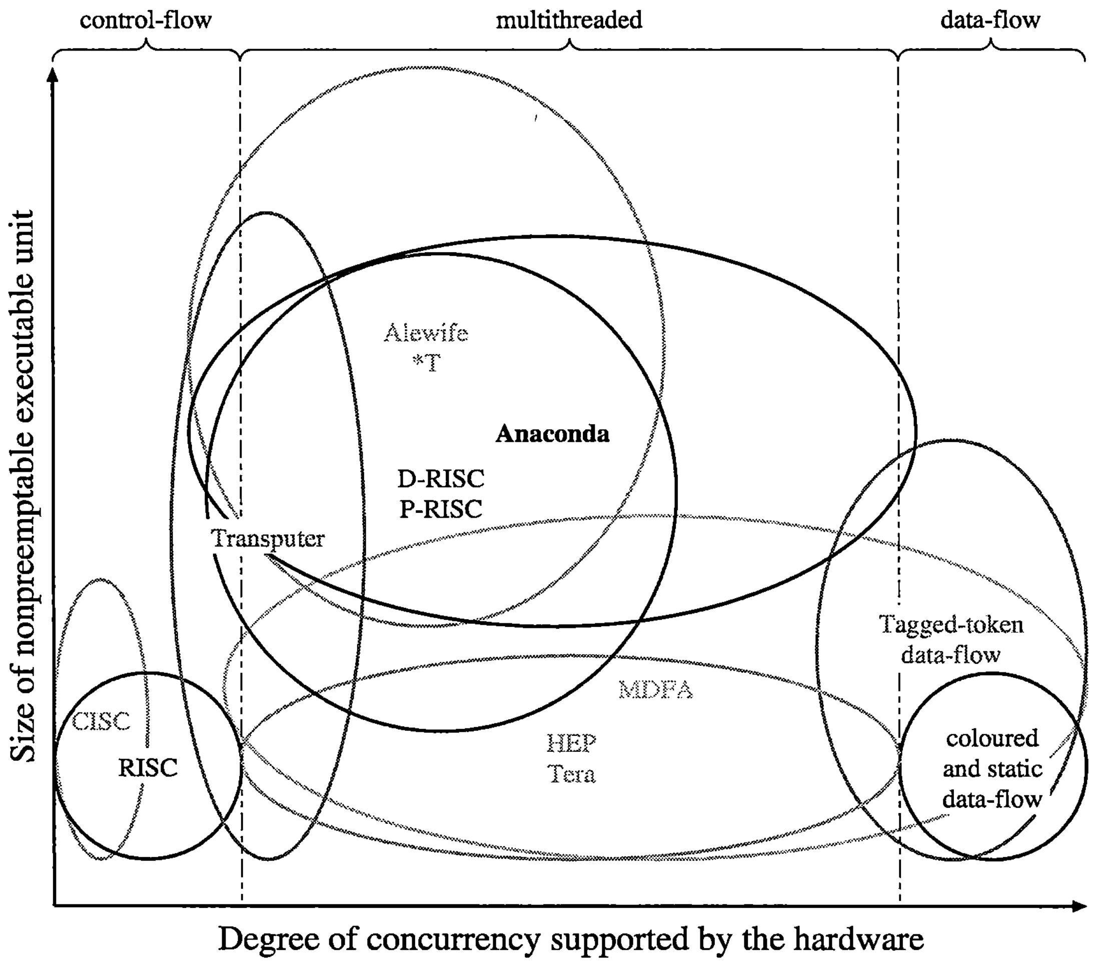

control-flow
multithreaded
data-flow
Size of nonpreemptable executable unit
Alewife
*T
Anaconda
D-RISC
P-RISC
Transputer
CISC
RISC
MDFA
HEP
Tera
Tagged-token data-flow
coloured and static data-flow
Degree of concurrency supported by the hardware

Figure 8.1: Filling the control-flow/data-flow void

8.1.2 Anaconda

With real-time requirements in mind, a multithreaded processor, called *Anaconda*, was designed. The underlying model is based upon data driven microthreads where a microthread is a nonpreemptable control-flow unit which takes between 8 and 32 clock cycles to execute. This supports coarse grained data-flow and control-flow execution and thus covers a large area in the design space depicted in figure 8.1. Moreover, microthreads are sufficiently large to allow sensible scheduling decisions to be made.

Anaconda's microthreads have up to 16 input parameters which must be presented, in a data-flow manner, before execution commences in a control-flow manner. A direct mapped matching store is provided for synchronising input parameters to microthreads. Scheduling is supported by a hardware priority queue for either fixed priority or earliest deadline first policies. Context switching is facilitated by a pageable register file and control-flow execution uses an Alpha 21064 styled pipeline. A code prefetch cache is also used so that when execution commences, all the required code is available locally as well as the data being preloaded into the register file.

A memory tree structure is used because it can maintain access frequency regardless of size, although the latency increases $O(\log(size))$. Virtual address translation is distributed amongst the memory modules which make up the memory tree, so adding more memory adds more address translation resource. Memory protection is provided by page based capabilities. This also allows the protection mechanism to be distributed amongst the memory modules of the memory tree. Finer grained protection is achieved using a typing system. Of particular note is the *illegal* type which facilitates detection of errors when the result data is used rather than flagging an exception when the error occurs. This method makes long pipelines, and remote memory protection, viable.

A detailed simulation of Anaconda was undertaken. Example applications demonstrated that both multithreaded control-flow and coarse grained data-flow could be efficiently executed. Whilst simple single threaded control-flow code runs inefficiently, loops may be unfolded to introduce more concurrency, thereby increasing efficiency. Furthermore, since concurrency is demonstrably virtually overhead free, it is anticipated that programmers writing specifically for multithreaded processors, like Anaconda, will naturally program with concurrency in mind.

Only a little concurrency is required (often around four active microthreads) in order to tolerate the latency of the memory tree. This is because microthreads are sufficiently large. For example, with four active microthreads there will be three microthreads worth of work (between 24 and 96 cycles); in which time a microthread issuing a memory read request can wait for a response and be rescheduled. Furthermore, microthreads may issue several memory requests so the memory may be kept busy. The long latency of deeply pipelined operations (e.g. floating point division) and interprocessor communication may also be tolerated in a similar manner.

Temporal predictability is assisted by eliminating memory caching by using the memory tree and tolerating its bounded latency. The matching store and instruction prefetch do both use caches but in a predictable and controllable manner. Even when other threads are executing, bounds may be placed upon the execution time, and, moreover, the worst case may be close to the best case if suitable use of the hardware scheduler is made.

Good use is made of the register file and matching store for intermediate results. This localises the transmission of much data which reduces time and power costs. The memory tree structure also removes the need for long busses with large fanout which facilitates rapid data transfer. The net result is that all transmission lines are short which makes the design amenable for implementation using self-timed circuits or some form of localised clocking scheme.

To conclude, Anaconda has the following desirable characteristics:

- functional primitives:
  - supports all the usual arithmetic, logical and conditional operations
  - can efficiently use operations requiring a long pipeline
  - hardware synchronisation primitive (the matching store) for concurrency control
- memory structure and access mechanisms:
  - scalable memory structure, virtual addressing and protection
  - good use of local storage
- concurrency:
  - simple flexible matching store for synchronisation (no interrupts are required)
  - hardware scheduler which supports a sensible scheduling policy
- parallelism:
  - the matching store assists interprocessor message passing and the hardware scheduler prioritises messages
- temporally predictable
- all signals only travel over relatively short distances

8.2 Future work

8.2.1 An implementation

This dissertation presents three man years of research into multithreaded processor design, culminating with a detailed design study. Whilst the results look promising, an implementation must now be considered. As one progresses with such a project, more people power and funding is required. Quite how or who should conduct an implementation is unclear.

8.2.2 Policing

Whilst the earliest deadline first or fixed priority scheduler is sufficient to schedule hard real-time tasks, this can only succeed if the tasks cooperate with one another. In order to police the execution pipeline, matching store and hardware scheduler usage, to place fire walls between applications, some method for ensuring a given quality of service is required. To avoid a rogue process consuming 100% of the execution pipeline resource, some mechanism would be required to ensure that only a given percentage may be used by a particular application. This may be achieved by traditional time slicing although some method of resource usage accounting would be required. Similarly, the matching store may be partitioned amongst applications. The relationship between the size of the matching store and the size of the scheduler, would ensure that the scheduler is not overloaded.

8.2.3 Control-flow core

Anaconda's control-flow execution pipeline was based upon the Alpha 21064 to assist performance comparisons. However, the Alpha 21064 is designed with different criteria. For example, all Anaconda instructions are preloaded into the instruction buffer before execution commences. Whilst loading instructions into the I-buffer, decoding, branch addresses computed and even some issue checks could be performed.

8.2.4 Language support

High level language issues need to be addressed to ensure that the programmer may make good use of the low overhead concurrency and good synchronisation offered by Anaconda. A declarative language would be one approach, but it is anticipated that an multithreaded imperative language would be more efficient because it more closely fits Anaconda's execution model.

8.2.5 Static code analysis

Anaconda is sufficiently temporally predictable to allow fixed and narrow bounds to be placed upon execution time. This is obviously advantageous for hard real-time applications. However, suitable static code analysis tools are required to assist the programmer.

8.2.6 Operating system support

Anaconda provides explicit hardware support for a nanokernel to perform basic housekeeping functions. On top of this, a microkernel may be constructed. Since Anaconda supports low overhead concurrency and synchronisation, this should have profound effects upon the structure of an efficient microkernel.

8.2.7 Caching for power reduction

Memory tree accesses are likely to consume significant power. A data and capability cache could be placed between the processor and the memory tree to reduce overheads and improve upon best effort performance. However, to ensure good temporal characteristics, this must not upset the constant access rate offered by the memory tree.

Bibliography

[1] D. Abramson and Egan G. The RMIT data-flow computer: A hybrid architecture. *The Computer Journal*, 33(3), 1990.

[2] A. Agarwal and et al. Sparcle: an evolutionary processor design for large-scale multiprocessors. *IEEE Micro*, pages 48–61, June 1993.

[3] B. Ahn and J.M. Murray. A pipelined, expandable VLSI sorting engine implemented in CMOS technology. *IEEE International Conference on Circuits and Systems*, 3:134–137, 1989.

[4] R. Alverson and et al. The Tera computer system. *Computer Architecture News*, 18(3), September 1990. Also published as ASM Supercomputing'90.

[5] P.V. Argade and et al. Hobbit — a high-performance, low-power microprocessor. In *CompCon spring 93*. 38th annual IEEE Computer Society International Computer Conference (CompCon spring 93) San Francisco, CA D930222—26, 1993.

[6] K. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computer Conference 32*, pages 307–314, 1968.

[7] A.D. Berenbaum, B.W. Colbry, D.R. Ditzel, R.D. Freeman, H.R. McLellan, K.J. Oconnor, and M. Shoji. Crisp — a pipelined 32 bit microprocessor with 13 kbits of cache memory. *IEEE journal of solid—state circuits*, 22:776–782, 1987.

[8] A.D. Birrell and B.J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, 1984.

[9] D. Bitton, D.J. DeWitt, D.K. Hsiao, and J. Menon. A taxonomy of parallel sorting. *Computing Surveys*, 16(3):287–318, 1984.

[10] R.J. Black. *Explicit Network Scheduling*. PhD thesis, University of Cambridge, Due to be submitted in 1994.

[11] K. Boland and A. Dollas. Predicting and precluding problems with memory latency. *IEEE Micro*, 14(8):59–67, 1994.

[12] A. Burns and A. Wellings. *Real-time systems and their programming languages.* Addison Wesley, 1989.

[13] T.J.W. Clarke. General theory relating to the implementation of concurrent symbolic computation. Technical Report 174, University of Cambridge, Computer Laboratory, New Museums Site, Pembroke Street, Cambridge, CB2 3QG, England, August 1989.

[14] E.G. Coffman and P.J. Denning. *Operating systems theory.* Prentice-Hall, 1973.

[15] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms.* MIT Press, 1991.

[16] H. Corporaal. Evaluating transport-triggered-architectures for scalar applications. *Microprocessing and Microprogramming*, 38:45–52, 1993.

[17] H. Corporaal and P. Vanderarend. Move32int, a sea-of-gates realization of a high-performance transport triggered architecture. *Microprocessing and Microprogramming*, 38:53–60, 1993.

[18] I. David, R. Ginosar, and M. Yoeli. An efficient implementation of boolean functions as self-timed circuits. *IEEE Transactions on Computers*, 41(1):2–11, January 1992.

[19] J.B. Dennis. Data flow supercomputers. *IEEE Computer*, 13(11), 1980.

[20] Digital. *DECchip*$^{\text{tm}}$ *21064-AA Microprocessor — hardware reference manual*, 1992. available via http://www.digital.com/info/forms/search.html.

[21] E.W. Dijkstra. The structure of THE operating system. *Communications of the ACM*, 11(5), 1968.

[22] E.W. Dijkstra. A heuristic explanation of Batcher's baffler. *Science of Computer Programming*, 9:213–220, 1987.

[23] K.E. Drexler. *Nanosystems — molecular machinery, manufacturing and computation.* John Wiley & Sons, 1992.

[24] M. Farmwald and D. Mooring. A fast path to one memory. *IEEE Spectrum*, October 1992.

[25] M.J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE,* 54(12), 1966.

[26] T.J. Fountain, K.N. Matthews, and M.J.B. Duff. The CLIP7A image processor. *IEEE transactions on pattern analysis and machine intelligence*, 10:310–319, 1988.

[27] S.B. Furber and I.E. Sutherland. Computing without clocks — asynchronous microprocessor design. Sun annual lecture in Computer Science at the University of Manchester, 1994.

[28] G.R. Gao. An efficient hybrid dataflow architecture. *Journal of Parallel and Distributed Computing*, 19:293–307, 1993.

[29] M.J.C. Gordon and T.F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher-order Logic.* Cambridge University Press, 1993.

[30] J.R. Gurd and et al. The Manchester prototype dataflow computer. *Communications of the ACM*, 28(1), January 1985.

[31] S. Hasuo and T. Imamura. Digital logic circuits. *Proceedings of the IEEE,* 77(8), August 1989.

[32] J.L. Hennessy and D.A. Patterson. *Computer architecture — a quantitative approach.* Morgan Kaufmann, 1990.

[33] W.D. Hillis. *The Connection Machine.* The MIT Press, 1985.

[34] K. Hwang and F.A. Briggs. *Computer Architecture and Parallel Processing.* McGraw-Hill, 1988.

[35] E.A. Hyden. *Operating system support for quality of service.* PhD thesis, University of Cambridge, England, 1994. Also available as technical report 340 from the University of Cambridge, Computer Laboratory.

[36] INMOS. *Transputer Reference Manual.* Prentice Hall, 1988.

[37] J.S. Kowalik (editor). *Parallel MIMD computation : the HEP supercomputer and its applications.* MIT Press, 1985.

[38] M. Kakumu and M. Kinugawa. Power-supply voltage impact on circuit performance for half and lower submicron CMOS LSI. *IEEE Transactions on Electron Devices*, 37(8), August 1990.

[39] D.B. Kirk. SMART (Strategic Memory Allocation for Real-Time) cache design. In *10th annual real-time system symposium*, pages 229–237, 1989.

[40] H.C. Lauer and R.M. Needham. On the duality of operating system structures. Technical report, Xerox PARC, 3408 Hillview Avenue, Palo Alto, California 94304, USA., 1978.

[41] D.T. Lee, H.S.U. Chang, and C.K. Wong. An on-chip compare/steer bubble sorter. *IEEE Transactions on Computers*, C-30(6), June 1981.

[42] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. St evens, A. Gupta, and J. Hennessy. The Dash prototype — logic overhead and performance. *IEEE transactions on parallel and distributed systems*, 4:41–61, 1993.

[43] T.E. Leonard, editor. *VAX architecture reference manual.* Digital Press, 1987.

[44] S. Levi and A.K. Agrawala. *Real-time system design.* McGraw-Hill International Editions, 1990.

[45] Yen-Chun Lin. On balancing sorting on a linear array. *IEEE Transactions on Parallel and Distributed Systems*, 4(5), May 1993.

[46] INMOS Ltd. *Occam 2 reference manual.* International series in computer science. Prentice-Hall, 1988.

[47] C. Mead and L. Conway. *Introduction to VLSI systems.* Addison-Wesley, 1980.

[48] S.W. Moore and G. Morgan. The recursive MOVE machine: r-move. In *RISC architectures and applications, IEE colloquium 1991/163*, 1991.

[49] Motorola. *MC88100 — RISC Microprocessor user's manual.* Prentice Hall, 1989.

[50] R.S. Nikhil and Arvind. Can dataflow subsume von Neumann computing? In *16th Annual International Symposium on Computer Architecture*, pages 262–272, 1989.

[51] G.M. Papadopoulos. *Implementation of a general-purpose dataflow multiprocessor.* Research Monographs in Parallel and Distributed Computing. MIT Press, 1991.

[52] G.M. Papadopoulos, G.A. Boughton, R.G. Greiner, and M.J. Beckerle. *T: Integrated building blocks for parallel computing. Technical report, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, Massachusetts 02139, USA., 1993. To appear in Supercomputing 1993.

[53] D. Parkinson and J. Litt, editors. *Massively Parallel Computing with the DAP.* Pitman, 1990.

[54] D.A. Patterson and J.L. Hennessy. *Computer organisation and design — the hardware/software interface.* Morgan Kaufmann, 1993.

[55] D. Reed and R. Kanodia. Synchronization with eventcounts and sequencers. Technical report, MIT Laboratory for Computer Science, 1977.

[56] T. Roscoe. Private communication, January 1994.

[57] S. Sakai and et al. Prototype implementation of a highly parallel dataflow machine EM-4. In *5th International Parallel Processing Symposium.* IEEE, 1991.

[58] J.H. Saltzer. Protection and the control of information sharing in Multics. *Communications of the ACM*, 17(7):388–402, 1974.

[59] M.A. Shutte and J.P. Shen. Instruction-level experimental evaluation of the Multiflow TRACE 14/300 VLIW computer. *The Journal of Supercomputing*, 7:249–271, 1993.

[60] D.P. Siewiorek, C.G. Bell, and A. Newell. *Computer Structures: Principles and Examples.* McGraw-Hill, 1982.

[61] R.L. Sites, editor. *Alpha Architecture Reference Manual.* Digital Press, 1992.

[62] J.E. Smith and S. Weiss. PowerPC 601 and Alpha 21064: a tale of two RISCs. *Computer*, 27(6):46–58, 1994.

[63] J.A. Stankovic and K. Ramamritham. *Hard real-time systems: tutorial.* IEEE (document collection EH0276–6), 1988.

[64] B. Stroustrup. *The C++ programming language.* Reading, Mass & Wokingham, 1986.

[65] I.E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989.

[66] P.C. Treleaven, R.P. Hopkins, and P.W. Rautenbach. Combining data flow and control flow computing. *The Computer Journal*, 25(2), 1982.

[67] N. Weiderman. Hartstone: synthetic benchmark requirements for hard real-time applications. Technical Report CMU/SEI-89-TR-23, Carnegie Mellon University, 1989.

[68] D.J. Wheeler. *Automatic Computing with the EDSAC.* PhD thesis, University of Cambridge, August 1951.

[69] M.V. Wilkes. *Memoirs of a computer pioneer.* MIT Press, 1985.

[70] M.V. Wilkes and R.M. Needham. *The Cambridge CAP computer and its operating system.* Operating and programming systems series. North Holland, 1979.

[71] T.E. Williams and M.A. Horowitz. A zero-overhead self-timed 160-ns 54-b CMOS divider. *IEEE Journal of Solid-State Circuits*, 26(11):1651–1661, November 1991.

Appendix A

Proof of correctness for the tagged up/down sorter

By B. T. Graham

A.1 Introduction

This appendix presents a formal specification of the tagged up/down sorter algorithm on page 37. The informal requirements on page 31 are also formalised and a sketch of the proof of correctness is presented. The formal proof was conducted with the assistance of the HOL system [29], which is a high integrity machine-implementation of a classical higher order logic.

A.2 Formal specification

We begin by defining data structures to represent the state of the queue. First is a record datatype.

```
⊢ ∀ (r:(*)record). (∃(n:num) (x:*). r = TAGGED n x) ∨
                   (∃ n x. r = UNTAGGED n x)
```

This theorem (theorems are identified by the ⊢ symbol) shows that objects of type `record` come in two forms: tagged and untagged. All such objects have two fields, a key field of type `:num` (natural numbers) and a data field which is polymorphic (i.e. it can be of any type, hence the type variable $*$). The sorting function of the algorithm is not dependent on the data field, hence leaving it polymorphic maintains full generality.

Extractor functions key and data, a predicate is_tagged, and a tagging function tag are defined, all with the obvious meanings.

The state of the queue is represented by a pair of finite lists of records. These lists represent only the elements in the queue which hold an inserted value, and not those elements that contain infinity. A simple informal proof by induction on the number of records in an infinite queue satisfies that the inserted records always cluster at the front of the queue, with no intervening empty elements, and thus the chosen representation is suitable.

The definitions of several functions required to manipulate list pairs follow. (Note that definitions are theorems in the HOL system. Recursive definitions may use pattern matching for arguments, as below where the empty list argument [] appears. Parts of the recursive definitions are conjoined with the ∧ operator.)

```
⊢ ∀ a b al bl.
    CONS_PAIR (x,y) (xs,yx) = (CONS x xs,CONS y ys)

⊢ (∀ rr f g. MAP_2_2 f g ([],rr) = g rr) ∧
  (∀ ll f g. MAP_2_2 f g (ll,[]) = g ll) ∧
  (∀ l ll r rr f g. MAP_2_2 f g (CONS l ll,CONS r rr) =
    CONS_PAIR (f (l,r)) (MAP_2_2 f g (ll,rr)))

⊢ (∀ rr f g. MAP_2_1 f g ([],rr) = g rr) ∧
  (∀ ll f g. MAP_2_1 f g (ll,[]) = g ll) ∧
  (∀ l ll r rr f g. MAP_2_1 f g (CONS l ll,CONS r rr) =
    CONS (f (l,r)) (MAP_2_1 f g (ll,rr)))

⊢ (∀ rr f g. MAPP_2_1 f g ([],rr) = g rr) ∧
  (∀ ll f g. MAPP_2_1 f g (ll,[]) = g ll) ∧
  (∀ l ll r rr f g. MAPP_2_1 f g (CONS l ll,CONS r rr) =
    CONS (f (l,r) (ll,rr)) (MAPP_2_1 f g (ll,rr)))
```

The CONS_PAIR function adds new head elements to both list arguments. Three versions of MAP functions which can be applied to pairs of lists are needed. MAP_2_2 applies a function f to pairs of elements from each list, and builds a pair of lists as a result. MAP_2_1 is similar but returns instead a single list. MAPP_2_1 applies a function f which takes as arguments not only the head elements at each level in the list, but the rest of the lists as well. All three functions have an additional function-valued argument g which is applied to the remaining part of the longer list, should the list arguments be unequal in length.

The MAP functions will often be used with functions that are predicates. To determine that the predicate holds at all levels the following functions are supplied to examine the returned lists.

```
⊢ (∀ f z. Foldl f z [] = z) ∧
   (∀ f z x xs. Foldl f z (CONS x xs) =
                Foldl f (f z x) xs)

⊢ ALL = Foldl ∧ T
```

Foldl is the standard *fold left* operation of functional programming. Applied to the arguments ∧ and T (the HOL constant for *true*), the resulting function ALL tests that every element in a list is T.

The algorithm consists of either an insertion or an extraction, followed by a compare and swap mapped down the pair of lists representing the state of the queue. The compare and swap operation is defined by the following two functions, the first of which is the comparison itself.

```
⊢ ∀ a b. below (a,b) = (key a < key b) ∨ (is_tagged a)

⊢ Compare_and_swap =
   MAP_2_2 (λ(a,b).(below (a,b)) => (b,tag a) | (a,b))
           (λ al.([],MAP tag al))
```

Note that only the key field and tag are considered by the algorithm. The relation below (a,b) holds when either the a record has a lower key value than the b record, or the a record is tagged. The Compare_and_swap function maps down the pair of lists, swapping the records (and tagging the right ones) as required so the record on the right is below the record on the left for every pair of records. If the lists are of unequal length, the remaining elements are moved to the right list and tagged.[1]

Insert adds a new record to the left list, while Extract removes the head element of the right list, thus both cause the two lists to shift one position relative to the other, always shifting the left side down relative to the right.

```
⊢ ∀ k ll rr. Insert k (ll,rr) =
     Compare_and_swap (CONS k ll,rr)

⊢ ∀ ll rr r. Extract (ll,CONS r rr) =
     (r,Compare_and_swap (ll,rr))
```

The definitions of both operations combine the required shifting of the lists and the application of the Compare_and_swap function. Extract returns both the extracted record as well as the diminished queue. The record types are polymorphic in both definitions.

[1]The λ symbol identifies a local function whose arguments end with a ".". The conditional expression of the form (c => a | b) can be read as IF c THEN a ELSE b. Compare_and_swap is an example of the use of higher order functions, where the list pair argument is not shown in the definition.

In order to define the invariant, we must be able to order records with identical key values. To assist the proof we use the data field to hold information about the order of insertion of records with identical keys, rather than adding an extra field to the record for this purpose. Thus the proof of correctness will be performed on `:num record` instances, and the data field will be higher for records inserted later. This does not restrict the generality of the results since the `data` field is ignored by the algorithm. The information we hold therein is strictly for use in distinguishing the insertion order of records with identical keys. An ordering relation which reflects this is defined as follows.

```
⊢ ∀ a b. BELOW (a,b) =
    (key a < key b) ∨
    ((key a = key b) ∧ (data a < data b))
```

This relation holds when the left record has a lower key value than the right record, or if the key values are identical, when the left record has a lower data value, indicating it was inserted before the right record,

The invariant consists of six components. First we require that each pair of records at the same depth in the queue is ordered, with the one in the right list BELOW the one in the left. We can ignore the last records in the longer list.[2]

```
⊢ ∀ ll rr. pair_ORDERED (ll,rr) =
    ALL (MAP_2_1 (λ(a,b). BELOW (b,a)) (K []) (ll,rr))
```

Second, the right list is either the same length as or one longer than the left list.

```
⊢ ∀ ll rr. Lengths (ll,rr) =
    (LENGTH ll = LENGTH rr) ∨
    (LENGTH ll + 1 = LENGTH rr)
```

Third, the records in the right list are ordered, with the least at the head. The predicate compares successive records in the list by using two copies of it, offset by one position.

```
⊢ ∀ rr. rt_ORDERED rr =
    ALL (MAP_2_1 BELOW (K []) (rr,TL rr))
```

Fourth, if a record in the left side is tagged, then it is BELOW the record in the right list which is at one level deeper in the queue. This captures the property, referred to on in section 4.3.2, that once a record arrives on the right it is sorted with respect to the keys of other records on the right.

[2]Note the expression `(K [])` is a function which returns the empty list `[]` when applied to any list argument.

```
⊢ ∀ ll rr. lt_Tagged (ll,rr) =
    ALL (MAP_2_1 (λ(a,b). is_tagged a ⊃ BELOW (a,b))
             (K []) (ll,TL rr))
```

Fifth, every untagged record in the left list that has the same key value as a record lower in either list must have a higher data value. In effect, this requires that records inserted later have this fact recorded in the data field.

```
⊢ ∀ ll rr. lt_UnTagged (ll,rr) =
    ALL (MAPP_2_1
         (λ(a,b) (aa,bb).
          ¬(is_tagged a) ⊃
           ALL (MAP (λ c. (key a = key c) ⊃
                          data c < data a) aa) ∧
           ALL (MAP (λ c. (key a = key c) ⊃
                          data c < data a) bb))
         (K [])
         (ll,rr))
```

Finally, every record in the right list is tagged.

```
⊢ ∀ rr. rt_Tagged rr = ALL (MAP is_tagged rr)
```

The invariant is the conjunction of the six conditions.

```
⊢ ∀ ll rr. Invariant (ll,rr) =
    pair_ORDERED (ll,rr)   ∧ Lengths (ll,rr)     ∧
    rt_ORDERED rr          ∧ lt_Tagged (ll,rr)   ∧
    lt_UnTagged (ll,rr)    ∧ rt_Tagged rr
```

The final two definitions below describe a constraint on the records being loaded and a relation that holds between the least element and the rest of the records in the queue.

```
⊢ ∀ new ll rr. load_constraint new (ll,rr) =
    ¬(is_tagged new) ∧
    ALL (MAP (λ c. (key new = key c) ⊃
                   data c < data new) ll) ∧
    ALL (MAP (λ c. (key new = key c) ⊃
                   data c < data new) rr)
```

```
⊢ ∀ a ll rr. LEAST a (ll,rr) =
    ALL (MAP (λ c. BELOW (a,c)) ll) ∧
    ALL (MAP (λ c. BELOW (a,c)) rr)
```

The load_constraint requires that new records are not tagged, and that they have the appropriate ordering information in the data field. The LEAST predicate will assure that the correct record is extracted from the queue.

A.3 Proof sketch

We present a proof sketch, rather than a detailed examination of the HOL machine proof. The first theorem shows the invariant holds on the empty queue. The proof consists of rewriting with the constraint definitions.

```
⊢ Invariant ([],[])                                                  (1)
```

In order to split the proof of later theorems into simpler parts, we derive an intermediate theorem which unfolds the invariant in the case when both lists are nonempty, expressing the invariant as equal to a series of conditions on the head elements plus the invariant holding on the pair of tails of the lists.

```
⊢ ∀ l ll r rr.                                                       (2)
    Invariant (CONS l ll,CONS r rr) =
    Invariant (ll,rr) ∧                                      (I0)
    BELOW (r,l) ∧                                            (I1)
    (¬(NULL rr) ⊃ BELOW (r,HD rr)) ∧                         (I2)
    (is_tagged l ⊃                                           (I3)
       ¬(NULL rr) ⊃ BELOW (l,HD rr)) ∧
    (¬(NULL rr) ⊃                                            (I4)
       ¬(is_tagged l) ⊃
         ALL (MAP (λ c. (key l = key c) ⊃
                        data c < data l) ll) ∧
         ALL (MAP (λ c. (key l = key c) ⊃
                        data c < data l) rr)) ∧
    is_tagged r                                              (I5)
```

This proof consists largely of unfolding the definitions of the constraint components. The clauses for pair_ORDERED, rt_ORDERED, and rt_Tagged are solved independently, while the remaining three: Lengths, lt_Tagged and lt_UnTagged are grouped together. The pair_ORDERED and rt_Tagged clauses are solved by unfolding definitions. The rt_ORDERED clause splits into two cases, where rr is an empty or a nonempty list. The remaining clauses use the same case split; when rr is NULL the Lengths constraint demands that ll is as well, and the definitions solve the case. Otherwise, the clauses are equivalent. This theorem is used in the proofs of two of the main theorems, and we label the 6 conditions expressed by the conjuncts of the right hand side $\mathtt{I}_0$ through $\mathtt{I}_5$ for easier reference.

The next theorem shows that the invariant is maintained when the lists are shifted one position relative to each other by removing the head of the right list and applying the `Compare_and_swap` function.

```
⊢ ∀ ll rr r. Invariant (ll,CONS r rr) ⊃                              (3)
             Invariant (Compare_and_swap (ll,rr))
```

The proof is by successive list inductions, first on `ll` then on `rr`. The two base cases are solved using the `Lengths` constraint to derive that the queue is empty or has one element. Theorem (2) is used to split the proof into conjuncts corresponding to $\mathtt{I}_0$ through $\mathtt{I}_5$. The $\mathtt{I}_0$ clause is solved using the inductive hypotheses, and the remaining clauses are solved by considering separately cases where the head elements are or are not swapped.

Theorem (3) is used to prove a theorem showing that the invariant is maintained by `Insert`, provided the record is not tagged upon entry and the data field ordering value is appropriate (i.e. the `load_constraint`).

```
⊢ ∀ ll rr new.                                                       (4)
     load_constraint new (ll,rr) ⊃
     Invariant (ll,rr) ⊃
     Invariant (Insert new (ll,rr))
```

The proof proceeds by a case split on the structure of `rr`. If `NULL` then `ll` must be as well by the `Lengths` constraint, and the invariant holds for the queue with a single record. In the other case (`CONS h t`), we split the goal into requirements on the heads of the lists and the invariant on the rest, using theorem (3) to solve the latter, splitting cases based on whether `new` and `h` get swapped.

The next intermediate theorem expresses the fact that the invariant assures that the least element is the one which is extracted from a nonempty queue.

```
⊢ ∀ ll rr least.                                                     (5)
     Invariant (ll,CONS least rr) ⊃ LEAST least (ll,rr)
```

The proof is by successive list inductions over `ll` then `rr`, doing a case analysis on the heads of the list, and using the transitivity of `BELOW`. This is combined with the earlier result to give the required theorem about the correctness of the `Extract` operation.

```
⊢ ∀ ll rr k.                                                    (6)
    Invariant (ll,CONS k rr) ⊃
    (let (removed,rest) = Extract (ll,CONS k rr)
     in
     ((removed = k) ∧
      LEAST removed (ll,rr) ∧
      Invariant rest))
```

A.4 Conclusion

Three theorems: (1), (4) and (6), express the correctness of the abstract algorithm. Relating these results to the behaviour of an implementation will need finite limits on the number of records which can be inserted, and an abstraction which distinguishes locations holding inserted values from others, as well as a justification for this abstraction.

Although the proofs have been described sparely, the completion of the proofs in a high integrity proof system lends a very high assurance of their validity. We note that the use of a formal proof system has more than once caught errors and omissions in the informal proofs of correctness we developed along with the formal proof. The invariant was strengthened in response to each omission, and the final variant carried version number 5. We can only speculate on whether this reflects carelessness on the part of the person performing the proof, or is typical of informal proofs of even relatively simple systems.

Appendix B

Anaconda instruction formats

Anaconda instructions are based around the Alpha instruction set [61]. The function of the additional Anaconda specific instructions is presented in section 6.7.

Anaconda and Alpha instructions have a 6 bit primary opcode which is summarised in figure B.1 (see page C-7 of [61] for details of the Alpha instructions). Instruction formats appear in figures B.2 and B.3.

	00	08	10	18	20	28	30	38
0/8	PAL (pal)	LDA (mem)	INTA (op)	MISC (mem)	LDF (mem)	LDL[2] (mem)	BR XR[1] (br)	BLBC XLBC[1] (br)
1/9	FLTV_M[1] (op)	LDAH (mem)	INTL (op)	\PAL\	LDG (mem)	LDQ[2] (mem)	FBEQ FXEQ[1] (br)	BEQ XEQ[1] (br)
2/A	FLTI_M[1] (op)	LDAP (mem)	INTS (op)	JSR	LDS (mem)	LDL_REQ_M[1] (mem)	FBLT FXLT[1] (br)	BLT XLT[1] (br)
3/B	FLTL_M[1] (op)	LDQ_U (mem)	INTM (op)	\PAL\	LDT (mem)	LDQ_REQ_M[1] (mem)	FBLE FXLE[1] (br)	BLE XLE[1] (br)
4/C	INTA_M[1] (op)	LDA_M[1] (mem)	\PAL\	\PAL\	STF (mem)	STL (mem)	BSR XSR[1] (br)	BLBS XLBS[1] (br)
5/D	INTL_M[1] (op)	LDAH_M[1] (mem)	FLTV (op)	\PAL\	STG (mem)	STQ (mem)	FBNE FXNE[1] (br)	BNE XNE[1] (br)
6/E	INTS_M[1] (op)	LDAP_M[1] (mem)	FLTI (op)	BIL[1] XIL[1] (br)	STS (mem)	STL_M[1] (mem)	FBGE FXGE[1] (br)	BGE XGE[1] (br)
7/F	INTM_M[1] (op)	STQ_U (mem)	FLTL (op)	BLL[1] XLL[1] (br)	STT (mem)	STQ_M[1] (mem)	FBGT FXGT[1] (br)	BGT XGT[1] (br)

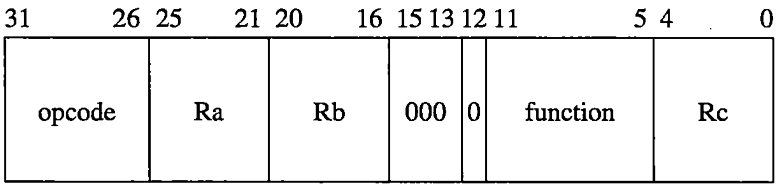
Notes:

1	an Anaconda extension
2	an Alpha opcode which may now only be used in supervisor mode
_M	the destination register is written through to the matching store
(op) (mem) (br)	} instruction format (see figures B.2 and B.3)

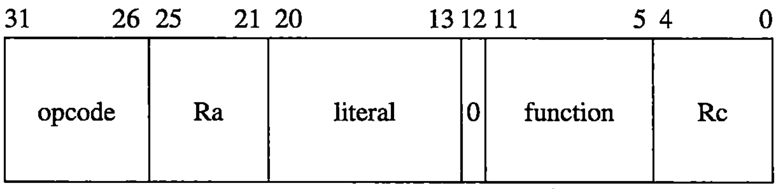
further details in [61]

Figure B.1: Anaconda additions to the Alpha opcode summary

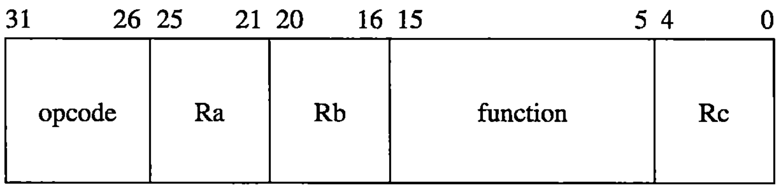
Integer operate format:
(op)



Integer operate format with literal:
(op)



Floating-point operate format:
(op)



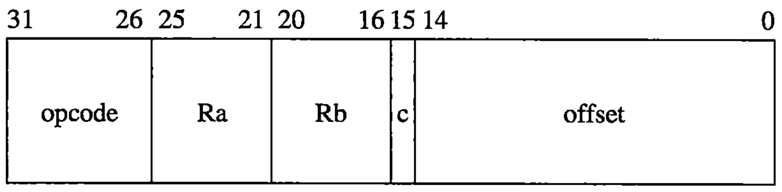
Notes:

if the opcode indicates a write through to the matching store then the default capability and activation frame base are implicitly used.

Figure B.2: Anaconda instruction formats — part 1

Memory Format:

(mem)

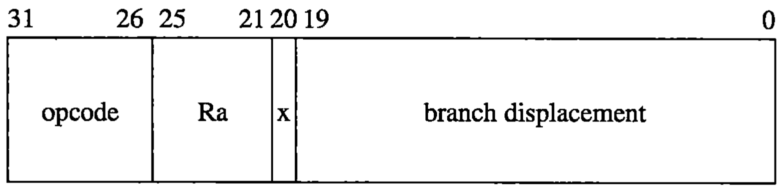


Notes:

c is the destination capability in the case of a store but the source capability for a load. The destination capability for a load is the default one and the destination address is calculated using the default activation frame base and Ra as an offset.

Branch Format:

(br)



Notes:

if x is 0 then the branch is a conventional one (in the br family)
if x is 1 then the branch is an intermicro thread branch (the xr famaily)

Figure B.3: Anaconda instruction formats — part 2

Appendix C

Anaconda memory copy program

```
/*****************************************************************************

        Anaconda test program - memory copying

*****************************************************************************/


#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "anacondageneric.h"
#include "writeaout.h"

#define RPLUS(r1,r2)    ((ireg) (((int) r1) + ((int) r2)))

#define MEMSIZE (2*3*5*7*11*9)

main(int argc, char *argv[])
{
  int i, j, segsize, numpar;

  if((argc!=2) || (sscanf(*++argv,"%d",&numpar)!=1)) {
    fprintf(stderr,"Usage: expacting one number to indicate number of parallel memory copies\n");
    exit(2);
  }
  if((numpar<1) || (numpar>10)) {
    fprintf(stderr,"ERROR: The number of parallel memory copies must be in the range 1..10\n");
    exit(2);
  }


  printf("Setting up application with %d parallel memory copies\n",numpar);

ASMBEGIN

/******* general initialisation which would normally be done by the OS *******/
    quadunalign();                      // source the global pointer
    br(gp,"findgp");
    text.quadaddaligned(getlabel("GPstart"));
label("findgp");
    ldq(gp,0,gp,h);
```

```
                                        // load the base activation frames pointer
    ldq(af,getlabel("ActivationFramesPtr")-getlabel("GPstart"),gp,h);

/******* start of the application ********************************************/

    rpcc(t0);

    // setup exit microthread
    lda(dl,1,zero);                     // set deadline
    ldap_m(dl,calcpresence(cp,hc,gp) & ~(((1<<numpar)-1)<<((int) p0)),dl);
                                        // set presence bits (cp,hc,gp and p0..p numpar)
    lda_m(gp,0,gp);                     // write through global pointer
    lda_m(hc,0,zero);                   // write through the home capability
    xsr(cp,"Exit");                     // source the code pointer

    // setup initialisation microthread
    lda(af,16*8,af);                    // next activation frame
    ldap_m(dl,calcpresence(cp,hc,in3,p3,p4,p5,gp),dl);
                                        // poast a load for a pointer to the memory ranges table
    ldq_req_m(p5,getlabel("MemRangesPtr")-getlabel("GPstart"),gp,h);
    lda_m(gp,0,gp);                     // write through the gp
    lda_m(hc,0,zero);                   // and the hc (home capability)
    lda_m(in3,(((int) p0)-16)*8,af);    // write the first sync. address for signalling exits
    lda_m(p3,16*8,af);                  // write the address of the next activation frame
    lda_m(p4,numpar,zero);              // write the counter for the number of parallel threads
    xr(cp,"Init");                      // next microthread and set code pointer to "Init"

    // Initialisation loop which spawns the microthreads to perform the copies
    //   input parameters:
    //     p3 = activation frame to allocate,  p4 = counter,
    //     p5 = pointer to memory ranges,      in3 = exit address
    cachealign();
label("Init");
    lda(t0,0,af);                       // save af
    lda(af,0,p3);                       // set af to the next free activation frame
                                        // initialise the presence bits
    ldap_m(dl,calcpresence(cp,in3,p0,p1,p2),dl);
    ldq_req_m(p0,0,p5,h);               // post load for the source address for the memory copy
    ldq_req_m(p1,8,p5,h);               // post load for the destination address for the memory copy
    ldq_req_m(p2,16,p5,h);              // post load for the number of quadwords to copy
    lda_m(in3,0,in3);                   // write the exit signal address
    xsr(cp,"MemCopyDoLoads9");          // write the code pointer

    lda(af,0,t0);                       // restore af
                                        // calculat the presence bits for another run of Init
    ldap_m(dl,calcpresence(cp,in3,p3,p4,p5),dl);
    lda_m(in3,8,in3);                   // update parameters and write them through
    lda_m(p5,3*8,p5);
    lda_m(p3,16*8,p3);
    subq_lit_m(p4,1,p4);
    xne(p4,"Init");                     // if there are more threads to spawn then run init again
    lda_m(dl,-1,dl);                    // otherwise deallocate activation frame
    NEXT();                             // and exit

    // Initiate the correct number of loads and run the appropriate microthread to store them
    //   input parameters:
    //     in3=exit signal address, p0=source address, p1=destination address, p2=counter
    cachealign();
label("MemCopyDoLoads9");
```

```
    subq_lit(p2,9,t0);                // can 9 loads be performed?...
    bge(t0,"MCDL9");
label("MemCopyDoLoads3");
    subq_lit(p2,3,t0);                // can 3 loads be performed?...
    bge(t0,"MCDL3");
    subq_lit(p2,1,t0);
    bge(t0,"MCDL1");                  // can 1 load be performed?...
    stq_m(zero,0,in3,h);              // signal exiting
    lda_m(dl,-1,zero);                // deallocate microthread
    NEXT();

label("MCDL1");                       // set up to use the 1 quadward copy microthread
    ldap_m(dl,calcpresence(cp,p0,p2,p3),dl);
    xsr(cp,"MemCopyStore1");
    subq_lit_m(p0,8*3,p0);
    br(zero,"MCDLload1");             // go and perform the 1 load etc.

label("MCDL3");                       // set up to use the 3 quadward copy microthread
    ldap_m(dl,calcpresence(cp,p0,p2,p3,p4,p5),dl);
    xsr(cp,"MemCopyStore3");
    subq_lit_m(p0,8*3,p0);
    br(zero,"MCDLload3");             // got and perform 3 loads etc.

label("MCDL9");                       // set up to use the 3 quadward copy microthread
    ldap_m(dl,calcpresence(cp,p0,p2,p3,p4,p5,p6,p7,p8,p9,p10,p11),dl);
    xsr(cp,"MemCopyStore9");
    subq_lit_m(p0,8*9,p0);

    ldq_req_m(p11,8*8,p0,h);          // post the required loads
    ldq_req_m(p10,7*8,p0,h);
    ldq_req_m(p9,6*8,p0,h);
    ldq_req_m(p8,5*8,p0,h);
    ldq_req_m(p7,4*8,p0,h);
    ldq_req_m(p6,3*8,p0,h);
label("MCDLload3");
    ldq_req_m(p5,2*8,p0,h);
    ldq_req_m(p4,1*8,p0,h);
label("MCDLload1");
    ldq_req_m(p3,0*8,p0,h);
    lda_m(p2,0,t0);                   // write through the conter
    NEXT();


    // Store 9 values and load some more
    //   input parameters:
    //     in3=exit signal address, p0=source address, p1=destination address, p2=counter
    //     p3...p11 contain data to be stored
    cachealign();
label("MemCopyStore9");               // store the 9 values
    for(i=0; i<9; i++) stq(RPLUS(p3,i),i*8,p1,h);
    subq_lit(p2,9,t0);
    bge(t0,"MCS9doloads");            // check if 9 more values can be loaded...
    ldap_m(dl,calcpresence(cp,p1),dl);  //...if not then rerun the load initialisation microthread
    lda_m(p1,8*9,p1);
    xr(cp,"MemCopyDoLoads3");
label("MCS9doloads");                 // perform 9 more loads and loop around again
    ldap_m(dl,calcpresence(p0,p1,p2,p3,p4,p5,p6,p7,p8,p9,p10,p11),dl);
    for(i=0; i<9; i++) ldq_req_m(RPLUS(p3,i),i*8,p0,h);
    lda_m(p0,9*8,p0);
    lda_m(p1,9*8,p1);
    lda_m(p2,-9,p2);
    NEXT();


    // Store 3 values and load some more
    //   input parameters:
    //     in3=exit signal address, p0=source address, p1=destination address, p2=counter
    //     p3...p5 contain data to be stored
    cachealign();
label("MemCopyStore3");                 // store the 3 values
    for(i=0; i<3; i++) stq(RPLUS(p3,i),i*8,p1,h);
    subq_lit(p2,3,t0);
    bge(t0,"MCS3doloads");              // check if 9 more values can be loaded...
                                        //...if not then initiate singe quadword copying
    ldap_m(dl,calcpresence(cp,p0,p1,p2,p3),dl);
    ldq_req_m(p3,0,p0,h);
    lda_m(p0,8,p0);
    lda_m(p1,3*8,p1);
    lda_m(p2,-1,p2);
    xr(cp,"MemCopyStore1");
label("MCS3doloads");                   //
    ldap_m(dl,calcpresence(p0,p1,p2,p3,p4,p5),dl);
    for(i=0; i<3; i++) ldq_req_m(RPLUS(p3,i),i*8,p0,h);
    lda_m(p0,3*8,p0);
    lda_m(p1,3*8,p1);
    lda_m(p2,-3,p2);
    NEXT();



    // Store 1 value and load some more
    //   input parameters:
    //     in3=exit signal address, p0=source address, p1=destination address, p2=counter
    //     p1 contains the data to be stored
    cachealign();
label("MemCopyStore1");
    stq(p3,0,p1,h);                     // store the value
    subq_lit(p2,1,t0);
    bge(t0,"MCS1doloads");              // any more to copy?...
    stq_m(zero,0,in3,h);                //...no so signal exiting
    lda_m(dl,-1,zero);                  // deallocate microthread
    NEXT();
label("MCS1doloads");                   // post another load and loop around again
    ldap_m(dl,calcpresence(p0,p1,p2,p3),dl);
    ldq_req_m(p3,0,p0,h);
    lda_m(p0,8,p0);
    lda_m(p1,8,p1);
    lda_m(p2,-1,p2);
    NEXT();



    cachealign();
label("Exit");
    rpcc(t0);
                                        // flush most of the pipeline
    for(i=0; i<8; i++) or(zero,zero,zero);
    or_lit(zero,0,r0);                  // quit simulation by OSF1 style call_pal EXIT
    call_pal(0x83);

/******************************************************************************/

    dataquadalign();
labeldata("GPstart");   printf("GP start = 0x%016lX\n",getlabel("GPstart"));
labeldata("StackStartPtr");
    data.quadaddaligned(getlabel("StackStart"));
labeldata("MemRangesPtr");
    data.quadaddaligned(getlabel("MemRanges"));

labeldata("MemRanges");
    segsize = MEMSIZE - (numpar-1)*(MEMSIZE / numpar);
    for(i=j=0; i<numpar; i++) {
      data.quadaddaligned(getlabel("SourceMemory")+j*8);
      data.quadaddaligned(getlabel("DestMemory")+j*8);
      data.quadaddaligned(segsize);
printf("segment %d, size %d\n",i,segsize);
      j += segsize;
      segsize = MEMSIZE / numpar;
    }

ASMGENERAL

labeldata("SourceMemory");
    for(i=0; i<MEMSIZE; i++) data.quadaddaligned(i);

labeldata("DestMemory");
    for(i=0; i<MEMSIZE; i++) data.quadaddaligned(0);


ASMVERYEND("/tmp/swm11/tmp.out")

  return(0);
}
```