Performance Implications of Context Switches on Misses to DRAM

Lance Pompe v Meerdervoort

A research report submitted to the Faculty of Science,
University of the Witwatersrand, Johannesburg, in partial fulfillment
of the requirements for the degree of Master of Science in Computer Science.

December, 1998

Declaration

I declare that this research report is my own, unaided work except where otherwise acknowledged. It is being submitted in partial fulfillment for the degree of Master of Science in Computer Science in the University of the Witwatersrand, Johannesburg. It has not been submitted before for any degree or examination in any other university.

_____

Lance Pompe v Meerdervoort

This_____day of_____1999

Abstract

Advances in microprocessor technology have resulted in the situation where CPU performance is improving faster than DRAM main memory performance. While it is true that current differences between CPU and DRAM speeds are not yet at the point where DRAM should be considered a slow peripheral, it is worthwhile considering the possibility of current trends continuing long enough that DRAM does resemble a slow peripheral. Eventually, a cache miss to DRAM could resemble a main memory page fault to disk - an expensive delay being experienced while the miss is handled. The logical approach may be to handle the miss as a page fault to disk is typically handled – to perform a context switch.

A basic requirement for context switching on DRAM accesses is a fast context switch. The RAMpage memory hierarchy, which handles the L2 cache as a paged memory and main memory as secondary storage, is the ideal platform for evaluating this idea. The reasons for this are twofold. Firstly, the RAMpage model, in treating the secondary cache as main memory, allows context-switching code to be pinned in SRAM, meaning that this code cannot cause references to DRAM once in SRAM. This ensures a faster context switch on average and avoids the complications of recursion in the miss handler when switching on a miss causes a miss itself. Secondly, although the RAMpage hierarchy has been shown to substantially reduce the number of references to DRAM, this benefit is offset by an increase in the time needed to handle a DRAM access. Thus there exists good potential for improved CPU utilization and therefore performance improvement through context switching. This is particularly true for large page sizes, in which fewer but more expensive misses occur.

The objective of this study is to investigate the feasibility of context switching as an approach to dealing with the increasing gap between CPU and DRAM performance and to evaluate the scalability of this approach as the speed gap widens. This objective is achieved by using trace-driven simulation to model the performance of the RAMpage hierarchy with context switching on page faults to DRAM. Overall results show context switching on page faults improves RAMpage performance from a previous maximum of 17% over the standard hierarchy, to 25%. A secondary result shows that pinning code used to perform context switches in SRAM under the RAMpage hierarchy substantially improves average context switching time, particularly when large pages are used.

Although a more detailed study would be needed to verify the merits of this approach fully, this work demonstrates that context switching on a DRAM access is an approach worthy of consideration, particularly when looked at in the context of the RAMpage model.

Acknowledgements

My supervisor, Dr. Philip Machanick was instrumental to the success of this project. Thank you Philip for your patience and for providing guidance, assistance and feedback during the course of this research work.

I am indebted to Pierre Salverda for assistance with RAMpage hierarchy details and for providing numerous suggestions. Dr. Borislav Roussev assisted with context switching details. Dr. Yinong Chen, Dr. Scott Hazelhurst, Dr. Conrad Mueller and Prof. Robert Baber assisted with suggestions and comments. Vusi Magagula, Shane Morton and Mogi Reddi helped with proof-reading. Thank you to all of you.

I'd like to express my appreciation to my parents and family for their love and support during this work.

Finally, I'd like to thank the Foundation for Research and Development and the University of the Witwatersrand for providing the financial support that allowed me to complete this degree, and Yttrium Technologies for providing computer equipment used for the simulations.

Contents

List of Figures

List of Tables

Chapter 1 – Introduction

1.1 Motivation

Designing a computer system involves looking at cost/performance tradeoffs and deciding where the optimal design point lies for the system in question. This depends on a number of technology trends which allow the computer architect to anticipate future problems that could arise if current methods continue to be used, and to suggest better ways of coping with a problem before it becomes too severe. A new architectural approach can then be evaluated cost effectively using simulation.

A significant trend in evidence today is the increasing speed gap between the CPU and main memory [Hennessy and Jouppi 1991, Hennessy and Patterson 1996]. This disparity became apparent in the 1960s and has grown steadily since then, threatening to produce a bottleneck in overall system performance. In response, caches were developed as a way of masking the lower performance of main memory. However, the effectiveness of caches is ultimately limited as the CPU-DRAM speed gap widens further, with threats of a looming "memory wall" – a barrier to further performance improvement imposed by the memory system [Wulf and McKee 1995, Wilkes 1995, Johnson 1995].

1.2 Proposed Solution

As lagging DRAM speed continues to degrade system performance, methods used by operating systems in dealing with slow disk drive devices start to look attractive. The delay caused by a disk access may run into millions of CPU cycles, prompting the use of context switching. Here, the overhead of a context switch is justified by the high cost of the disk access. Using the same rationale, a delay in accessing DRAM, although not yet nearly as high as a disk access, may possibly be similarly masked by switching the CPU to another task.

To enable a context switch to be performed on a cache miss to DRAM, the cache needs to be software-controlled. Additionally, it is desirable that context switching code be locked in the cache to avoid the situation where it calls itself recursively when part of its code causes a cache miss. One way to achieve this is to manage the lowest level of cache as a paged memory. The *RAMpage memory hierarchy* [Salverda 1997, Machanick *et al.* 1998] proposes that main memory be moved up to the lowest level of SRAM cache while DRAM is managed as a first level paging device, and the disk becomes a second level paging device. This arrangement

allows the necessary modifications to be made to support context switching on DRAM accesses: SRAM misses to DRAM are handled in software and context switching code can be pinned in SRAM.

RAMpage simulation results [Salverda 1997] indicate performance improvements over the standard memory hierarchy. Moreover, these improvements are shown to scale with the expected future CPU-DRAM speed gap increases. Improvements come mainly from a lower SRAM miss rate, particularly when large page sizes are used. However, the benefit of a reduced miss rate as the page size is increased starts to decrease as the worsening cost of larger pages takes effect. The result is that optimal performance does not occur where it should, where miss rates are optimal; i.e. optimal performance occurs with non-optimal miss rates on smaller pages.

Now, a high miss cost thus benefits context switching by creating more room for context switches. A context switch will obviously only be worthwhile if its overhead is less than the time the CPU would spend being idle if the switch was not taken. Thus, it can be expected that context switching will be more beneficial on large pages. Likewise, as the speed of a CPU increases, so the amount of work that can be done by it in a unit of time increases. Thus, it can be expected that for large page sizes, context switching on a page fault will be more beneficial. Since the RAMpage hierarchy has the lowest miss rates with the largest page size as well as the ability to scale well with increasing CPU speed, context switching on page faults should be the most beneficial where RAMpage should be optimal, i.e. with large page sizes. This means performance should scale well with increasing CPU speeds.

The aim of this research is to investigate the feasibility and model the performance of context switching on page faults under the RAMpage hierarchy. This represents the tradeoff between increased CPU utilization on the one hand, and the context switch overhead and a possible increased SRAM miss rate on the other. Another goal is to evaluate the scalability of context-switching-enabled RAMpage with the CPU-DRAM speed gap.

1.3 Document Structure

The remainder of this report is organized as follows. The next chapter provides a background to the problem and introduces related concepts. Chapter 3 looks at technology trends influencing memory system and CPU performance. Chapter 4 summarizes previous work to improve memory systems and introduces the RAMpage memory hierarchy. Issues relating to context switching in the RAMpage hierarchy are dealt with in chapter 5. The simulated

systems are described in Chapter 6, followed by the performance results in Chapter 7. In conclusion, overall findings and future work are discussed in Chapter 8.

Chapter 2 – Background

2.1 Introduction

This chapter serves as an overview of concepts and terminology relevant to the CPU-DRAM performance problem. Firstly, the design of memory systems is looked at to illustrate the tradeoffs and problems involved in building a memory system. Basic cache concepts are then presented as a basis for a more detailed explanation of the CPU-DRAM performance problem, followed by a section on resource utilization principles.

2.2 Memory System Design

The memory system determines the speed with which data can be transferred to and from the processor. Regardless of how fast a processor executes instructions, if the memory system is not able to keep up, it will limit performance.

There are three conflicting goals in designing a memory system [Hennessey and Patterson 1996]:

- High capacity.
- High performance.
- Low cost.

Satisfying these requirements has led to the design of hierarchical memory systems composed of a mix of memory devices that range in performance, cost and capacity. The design goal of a hierarchical memory is to give the illusion that the entire memory system is composed of the fastest devices in the structure, while its cost is dominated by the slower, cheaper, higher capacity devices. Figure 1 shows a typical memory hierarchy organization.

The key to the hierarchy working is the decreasing frequency of access of each memory device as one moves down the hierarchy. The basis for this is a principle known as *locality of reference* [Denning 1968]. Memory references made by a program consist of data and instructions. During execution, both data and instruction references tend to be clustered in two ways: in time, and in position in memory, known as *temporal* and *spatial* locality respectively [Smith 1982]. The nature of most programs allows this locality to be exploited – programs

typically contain a number of loops and operations on tables and arrays, causing repeated references to a small set of data and instructions [Hennessy and Patterson 1996].

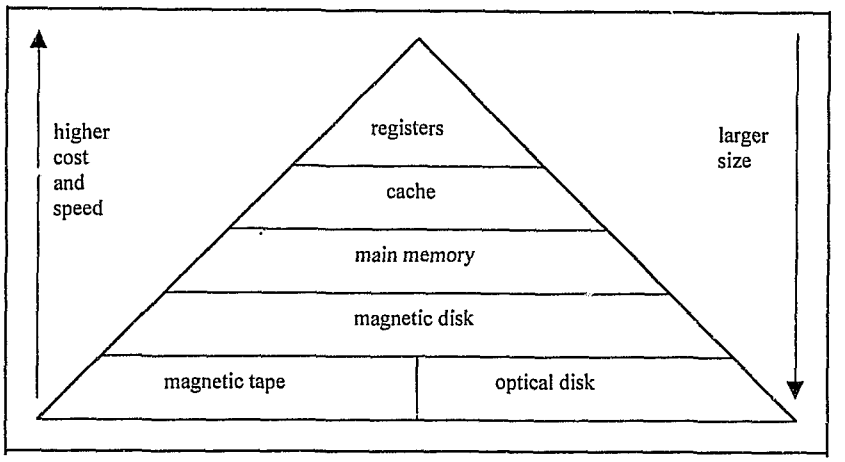

***Figure 1*: A memory hierarchy.**

2.3 Caches and Cache Design

2.3.1 Introduction

The portion of the memory hierarchy most significant in terms of performance and complexity is the cache. A cache is a small, fast buffer in which parts of the contents of a larger slower memory which are likely to be needed soon by the processor are kept. The purpose of a cache is to hide the latency of the slower memory by providing an average memory access time closer to that of the cache than that of the slower memory.

Apart from keeping production costs to a minimum, there are two main conflicting goals in cache design:

- Keeping the cycle time (minimum time between requests) to a minimum.
- Keeping the miss rate as low as possible.

These desirable properties conflict because modifying design parameters to improve one usually results in the other being negatively affected. Tradeoffs among cache design parameters are discussed below, but first a few important concepts are introduced.

2.3.2 Basic concepts

Cache or memory *access time* is the time between when the CPU requests a word, and the desired word arrives. A cache contains a portion of main memory and is organized into *blocks*

(sometimes also called *lines*). Each block contains a number of words and is the smallest unit of storage in a cache which is transferred between the cache and memory during a cache miss. Each block is uniquely addressed using a *tag*, *index* and *block-offset*. The block-offset selects the desired data from the block, the index field is used to select the *set* (group of two or more blocks in the cache) and the tag field is used for the comparison to determine whether a hit occurs [Hennessy and Patterson 1996]. The objective behind organizing caches into blocks is to exploit the spatial locality of references – there is a high probability that future references will occur within the same block.

The *working set* of a process is the set of instructions and data that the process frequently accesses over a specific period. This idea was developed by Coffman and Denning [1973] and applied to virtual memory but the principle also holds for caches. The ideal is to have the working set of a process resident in the cache.

When the CPU needs to access a memory reference, a check is first made to see whether the reference is in the cache. If so, a cache *hit* occurs and the reference is transferred to or from the cache, if not, a cache miss occurs, resulting in a block containing the reference being transferred from main memory to the cache.

When *paging* is used, the CPU uses *virtual* addresses that need to be translated to *physical* addresses before an access to main memory occurs. The *page table* provides the virtual to physical mapping and is stored on disk with parts resident in memory. A *translation lookaside buffer* (TLB) is used to cache recently used address translations, reducing the need for time-consuming memory and disk accesses.

Since the cache is much smaller than the virtual or physical address space, many blocks may potentially be assigned to the same position in the cache, resulting in an incoming block displacing an existing block. Determining which block is replaced depends on how the cache is organized, particularly with respect to the number of locations a block is able to be placed into, or the *associativity* of the cache. Caches can be divided into three groups according to their degree of associativity [Smith 1982]:

- *Direct mapped.* A block is always placed in one location in the cache.
- *Set associative.* An *n-way* set associative cache can place a block in any of *n* locations.
- *Fully associative.* A block can be placed anywhere in the cache. The associativity of a cache has important implications for the replacement strategy. Replacement in a set associative and fully associative cache involves choosing between a number of blocks, the

higher the associativity, the more sophisticated this selection process can be to avoid replacing the wrong block.

2.3.3 Cache design tradeoffs

There are three basic parameters available to cache designers, namely, *cache size*, *degree of associativity* and *block size*. These parameters are related to the three types of misses that can occur [Hennessy and Patterson 1996]:

- *Compulsory misses.* The compulsory miss rate is related to the block size. These misses occur when a process calls for new references that have not been previously loaded in the cache. There is no way to avoid these misses entirely. Since it is relatively easy to increase the bandwidth of memory, thereby increasing the number of words transferred per unit of time, a large block size will increase overall throughput. Increasing the block size also serves to reduce compulsory misses, being in effect a *prefetch* mechanism. However, at a certain point the compulsory misses saved are at the expense of increased conflict or capacity misses.
- *Conflict misses.* The conflict miss rate is related to the degree of associativity of the cache. A conflict miss occurs when a block that is present in the cache is displaced to make room for an incoming block, only to be referenced again and displace the block that displaced it. This happens when two or more blocks map to the same cache location with the one block being replaced with the other when it should have remained in the cache, being part of the working set. A direct-mapped cache will experience the most conflict misses because each block maps to exactly one location. Increasing the associativity will decrease this type of miss, but increased associativity comes at the price of an increase in access time (hit time) and a more complex implementation.
- *Capacity misses.* Capacity misses are related to the cache size. A cache that is too small to hold the working set of a process will experience capacity misses. Increasing the cache size will reduce capacity misses, but also tends to increase implementation costs and cycle time.

Thus, there is a tradeoff between reducing each type of miss by increasing its related design parameter and an increase in the cycle time and complexity or negatively affecting another miss type.

2.3.4 Cache performance

System performance is a decreasing function of the cache miss rate, the cache access time (hit time) and the time taken to service a cache miss. Let T_c and T_m represent the cache access time

and average time to service a cache miss respectively, p and $(1-p)$ the probability of a cache hit and cache miss respectively. Then T_{ave}, the average memory access time is given by:

$$T_{ave} = pT_c + (1-p)T_m$$

As processors reach greater speeds in relation to DRAM, T_m and to a lesser extent T_c become relatively more expensive. Since T_m is mostly a function of main memory speed and bus architecture, efforts to improve cache performance usually try to reduce the miss rate $(1-p)$, and the cache access time T_c. Simultaneously minimizing both cache access time and miss rate is almost impossible, necessitating a tradeoff between them [Agarwal *et al.* 1988].

2.3.5 TLB performance

The performance of the TLB is critical, since TLB misses can have a significant impact on system performance. At best, the portion of the page table needed is in the L1 cache, and at worst, a disk access is needed to look up a page table entry. Code or data references that are scattered through memory, such as object-oriented and operating system code tend to have the worst TLB performance [Cheriton *et al.* 1993, Nagle *et al.* 1993].

The next section reviews operating system principles relevant to context switching.

2.4 Processes and Resource Utilization Principles

2.4.1 Introduction

Ensuring that the resources in a system are optimally used is a major goal of operating system design. It is desirable that the time the CPU is idle is kept to a minimum. This section reviews operating system concepts and terminology as a means to better understand the issues involved in the experimental work in this research, i.e. performing context switches under RAMpage. A description of the *process* concept is given first, followed by an overview of *context switching*. *Threads* are then introduced as having the benefit of a less costly context switch than processes. An overview is given of the *scheduler* and *dispatcher*, both of which are involved in performing a context switch. The mechanism used to *schedule* the order of process switching is then described. Lastly, *asynchronous I/O* is briefly discussed as the principle used to improve response time by relegating slow I/O operations to a peripheral, while the CPU performs useful work on another process.

2.4.2 Processes

Informally, a process is a program in execution. In addition to the program code, a process includes the current activity represented by the value of the *program counter* and the contents

of the processor's registers. A process also includes the process stack, which contains temporary data such as local variables, and a data section containing global variables.

As a process executes, it changes state. Each process may be in one of the following states [Tanenbaum 1987]:

- *New* – the process is being created.
- *Running* – instructions are being executed.
- *Blocked* – the process is waiting for some event, such as an I/O operation.
- *Ready* – the process is waiting to be assigned to a processor.
- *Terminated* – execution is complete.

Each process is represented in the operating system by a *process control block* (PCB). The PCB stores the following information [Silberschatz *et al.* 1994]:

- *Current process state* – whether the process is running, blocked etc.
- *Program counter* – indicates the address of the next instruction to be executed.
- *CPU registers* – depending on the computer architecture the registers vary in number and type. They include accumulators, index registers, stack pointers and general-purpose registers, as well as any condition-code information. This state information and the program counter must be saved when an interrupt occurs to allow the process to continue afterwards.
- *CPU scheduling information* – includes a priority indicator, scheduling queue pointers etc.
- *Memory-management information* – stores page table information and the base and limit register values.
- *Accounting information* – includes CPU time used, time limits, process numbers etc.
- *I/O status information* – contains the list of I/O devices allocated to this process, a list of open files etc.

Therefore, the PCB serves as a repository for any information that may vary from process to process.

2.4.3 Context switching

Switching the CPU to another process requires saving the state of the old process and loading the saved state for the new process. Context-switch time is pure overhead, because the system

does no useful work while switching. The speed of a context switch varies from machine to machine, depending on the memory speed, the number of registers which must be copied and the existence of special instructions (such as a single instruction to load or store all registers). The cost of a context switch ranged from 1 to 1000 microseconds in 1993 [Silberschatz *et al.* 1994]. Figure 2 shows the sequence of events in a context switch between processes P_0 and P_1.

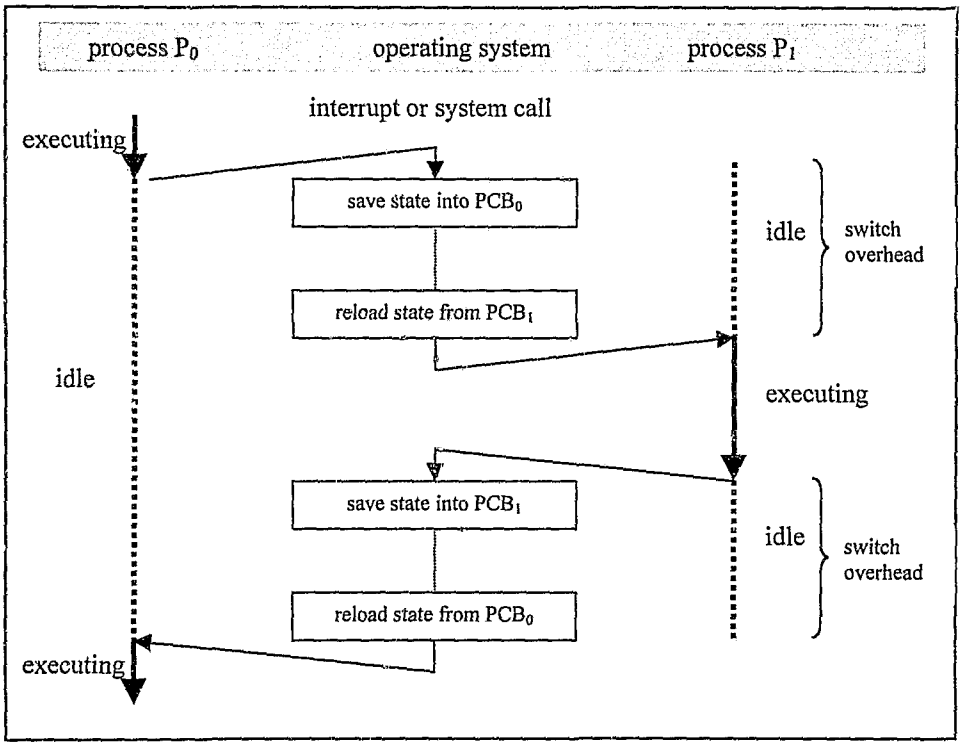

***Figure 2*: A context switch between two processes[1].**

Context switch times are highly dependent on hardware support. If multiple register sets are provided, a context switch may be achieved by simply changing the pointer to the current register set. If there are more active processes than register sets the system resorts to copying register data to and from memory as before. Also, the more complex the operating system, the more work must be done during a context switch. Advanced memory-management techniques may require extra data to be switched with each context. For example, the address state of the current process must be saved before the next process is run; the amount of work needed to do this depends on the memory-management method of the operating system.

[1] Can be generalized to more than two processes.

2.4.4 Threads

A thread can be defined as a single sequential flow of control. It is a basic unit of CPU utilization and consists of a program counter, a register set, and a stack space to store code and data. It shares with peer threads its code section, data section, and operating system resources such as open files and signals. The extensive sharing makes CPU switching among peer threads inexpensive compared with context switches among *heavyweight* processes. Although a thread context switch still requires a register set switch, no memory-management related work need be done [Silberschatz *et al.* 1994].

Having multiple threads of control allows an application to overlap I/O operations within a single address space. Operating system support for threads is becoming the norm [Birrell 1989, Tanenbaum 1995]. Unfortunately, most versions of Unix do not inherently support threads, but threads can still be implemented without kernel support. These issues will be considered later.

2.4.5 Scheduler and dispatcher

The scheduler is responsible for selecting the next process to execute, and the dispatcher is responsible for performing the actual context switch. The time taken by the dispatcher to stop one thread and start another running is called the *dispatch latency*. The sum of the time taken by the scheduler to select the next thread and the dispatch latency is the *context switch time* or *context switch overhead*. The greater the allowable context switch time and the lower the dispatch latency, the more time the scheduler can take in deciding which thread to run next [Silberschatz *et al.* 1994].

2.4.6 Scheduling

The scheduling algorithm is usually *Round Robin* (RR) or *Shortest Job first* (SJF). SJF aims to schedule the thread (or process) which has the shortest *CPU burst*, in other words the one which will perform I/O the soonest. The idea is that the sooner the I/O operations are started, the sooner they will be completed, since the I/O occurs while the CPU does other work, giving the best overall response time.

Scheduling can also be *preemptive* or *non-preemptive*. Non-preemptive scheduling allows a context switch to occur only if the thread encounters an I/O operation. The obvious disadvantage of non-preemptive scheduling is that a CPU bound thread can hold the CPU for a lengthy period, starving other threads of processing in the mean time. Under preemptive scheduling, a thread is executed until it encounters an I/O operation or the end of its time slice

or *quantum* has been reached. Preemptive scheduling facilitates good resource utilization and most current operating systems use it [Silberschatz *et al.* 1994].

2.4.7 Asynchronous I/O

The objective of multiprogramming is to maximize resource utilization and provide good response times for users. When a thread requests a resource that is not immediately available, it must wait. For example, an I/O operation such as a page fault involves a slow disk access and if processing was to halt each time this occurred, the CPU would spend large amounts of time being unproductive. Since I/O can occur without CPU assistance by using DMA, the CPU is free to perform other work until the I/O has been completed if it is switched to another process or thread. This is called *asynchronous* I/O and is effective because the overhead involved in switching the CPU to another process is small in comparison to the amount of time the I/O operation takes [Silberschatz *et al.* 1994, Tanenbaum 1987].

Chapter 3 – Technology Trends

3.1 Introduction

This chapter examines the technology trends that have played a role in creating the difference between CPU and DRAM speeds. These trends are important in that they allow us to make predictions regarding the further widening of the CPU-DRAM speed gap in the future.

3.2 The CPU-DRAM Speed Gap

During the last few decades, the rate of improvement in CPU speed has exceeded the rate of improvement in DRAM memory speed. Each is improving exponentially, but their difference is also growing exponentially. Over the past few decades, CPU speeds have been improving at 50 to 100% per year, while DRAM memory latency has been improving at only 7% per year [Hennessy and Patterson 1996], resulting in a doubling of the time (in terms of instructions) to access DRAM every 6.2 years [Boland and Dollas 1994]. Figure 3 illustrates this by comparing typical memory and processor cycle times from 1955 to 1990[2].

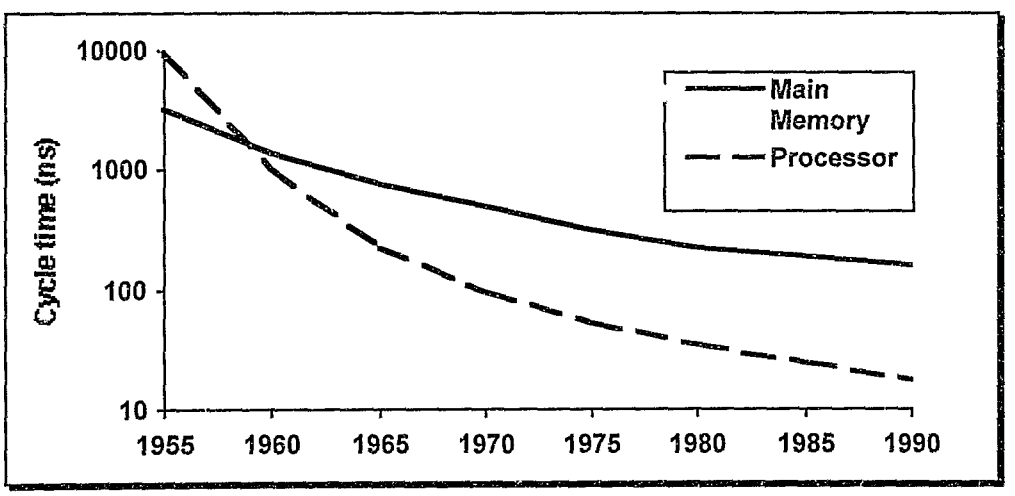

***Figure 3*: IBM mainframe processor and main memory performance**
Source: Stallings [1993].

For example, a Silicon Graphics 4D/380 multiprocessor system from the late 1980s had a L2 cache miss cost of 1μs. The more recently available Silicon Graphics Indigo 2 (R4400

[2] This situation is currently much worse, due to the ongoing performance improvements from RISC architectures.

processor) with a peak execution rate of 200 MIPS had a L2 miss cost of 1.17μs [McVey and Staelin 1996]. The cache miss cost has not improved over this period; in fact, it has worsened, even though DRAM has become faster. This is because other changes in the memory system such as larger caches and block sizes have prevented misses from becoming faster [Machanick 1996].

The detrimental effect on performance emerges when a cache miss is seen in terms of missed instruction executions or processor cycles. The 4D/380 lost about 30 instructions per L2 miss at its peak instruction issue rate, while the Indigo 2 cache miss costs 234 instructions. A DEC Alpha 21164 running at 600MHz [Digital 1998] with a peak instruction issue rate of 4 per cycle and a cache miss cost of 200ns has a worst case cache miss penalty of 480 instructions.

This trend indicates L2 cache miss costs will soon be in the region of thousands of lost instruction executions.

To see how the CPU-DRAM trend will affect system performance in future, consider again the equation for average memory access time:

$$T_{ave} = pT_c + (1\text{-}p)T_m$$

where p is the probability of a cache hit, $(1\text{-}p)$ the probability of a cache miss, and T_c and T_m are the cache and DRAM access times respectively.

If we assume that the only misses are compulsory misses (i.e. no capacity and conflict misses) and the cache has a cycle time consistent with the processor (no hit penalty), then $1\text{-}p$ is just the probability of accessing a new location. The fact that $1\text{-}p$ can never be eliminated entirely means that as T_c and T_m diverge, improvement of T_{ave} will be restricted by T_m. This means that at some point in the future, system performance will be dominated by T_m. In fact, at this point any improvement in CPU performance will not improve overall system performance at all – a wall will have been reached [Wulf and McKee 1995].

Ideally, memory should be constructed using the same technology as the CPU, but this would prove too costly in reality. The semiconductor technologies resulting in this difference in performance are examined in the next two sections.

3.3 Semiconductor Technology

Because the design emphasis for caches is on performance whereas main memory is designed with an emphasis on capacity, their underlying technologies differ.

Main memory is constructed from DRAM (dynamic random access memory), while caches use SRAM (static random access memory). DRAM requires a periodic refresh and data must be written back after being read, resulting in a difference between the cycle and access times. SRAM uses more components to remain stable during a read operation, requiring no refresh or data to be written back after a read. Thus, for SRAM, cycle time and access time do not differ.

Currently, the density of DRAM (in effect the inverse of cost) is roughly 16 times that of SRAM, while SRAM is roughly 16 times faster (and more expensive) than DRAM [Hennessy and Patterson 1996].

3.4 Effects of Shrinkage

The high rate of microprocessor performance improvement has been driven by two major factors [Hennessey and Jouppi 1991]:

- The increase in the number of transistors available on a chip due to improvements in semiconductor manufacturing processes.
- Architectural advances, including pipelining, caches and RISC ideas.

Sustained progress in optical lithography has made possible the continued shrinking of components on a chip. Since the number of components per chip is an $O(n^2)$ function of feature size, a decrease in feature size increases the number of components per chip dramatically, as well as improving circuit characteristics such as switching time and power dissipation. Moreover, having more transistors available on a chip allows the use of advanced features such as extra functional units and pipelines to exploit instruction level parallelism and branch prediction to keep pipelines full.

More on-chip real estate also allows caches to be moved on-chip, thereby reducing communication delays and enabling the cache cycle time to be the same as the CPU cycle time. Based on expected optical lithography advances, Geppert [1996] predicts that in a few years time microprocessors will employ more than 60 million transistors and have clock rates of over 1GHz.

Another effect of the miniaturization of components is the increase in DRAM densities. Each new generation of DRAM is four times the density of the previous generation, and as the integration level increases, the price of memory continues to drop.

Thus, feature size improvements tend to push up performance in the case of CPUs and density in the case of memory, leading to a worsening of the performance disparity.

3.5 Cache Trends

Reducing the miss rate of a cache involves the use of larger, more complex, expensive and slower caches with consequent increases in access time. Because of this there has been a trend towards using simple direct-mapped caches that have favourable access times but higher miss rates [Hill 1988].

Although larger caches have a higher access time, this penalty is usually lower than that imposed by higher associativity. The higher conflict miss rate in direct-mapped caches has resulted in these caches becoming substantially larger to overcome the absence of associativity [Agarwal *et al.* 1988]. This trend can be expected to continue as the effect of the miss cost, T_m becomes the dominant component of the average memory access time equation.

3.6 Software Trends

Trends in program size and implementation methodologies have important, often complex effects on the memory system and vice versa. This section looks at some of these effects.

A side effect of higher DRAM densities is the decreased importance of ensuring that programs conserve memory, allowing memory to be used as a tradeoff for speed and reduced design complexity. The use of RISC designs was not attractive until DRAMs became relatively cheap and offered high capacities, because RISC architectures use more memory than CISC architectures, but offer higher speeds [Hennessy and Jouppi 1991].

As application and operating system software evolves to include more features and to become more portable, maintainable and reliable, it also tends to consume more memory resources [Uhlig *et al.* 1995]. This "code bloat" affects a memory hierarchy at all levels in a variety of ways. The larger working sets of bloated programs require more main memory, bloated programs use virtual memory in a sparser and more fragmented manner, making their page-table entries less likely to fit in TLBs, and the increased path lengths of bloated code can reduce its locality, making caches less effective.

Another important software trend is the move towards object-oriented programming. This programming style with its emphasis on separate objects tends to make relatively poor use of locality, with detrimental effects on cache performance [Cheriton *et al.* 1993].

Some of these trends have been offset by improvements in memory technology, such as the increase in DRAM densities and cache sizes. However, recall that caches are limited in size and associativity because of cycle time requirements. Code bloat and object-oriented

programming are resulting in higher cache miss rates and there is no obvious way of reducing these misses. Since these trends can be expected to continue, reducing the frequency of cache misses will not be any easier in the future if current cache design methods are used.

Chapter 4 – Previous and Related Work

4.1 Introduction

The memory performance problem can essentially be tackled in two ways: firstly, by tackling the root of the problem by improving the speed of DRAM; and secondly, by trying to mask the problem by improving memory system performance. This chapter describes efforts to improve the performance of the memory system, namely the DRAM main memory and the cache. Approaches to improving the performance of DRAM have shown some potential. A few of the most promising DRAM technologies are outlined here.

Because of the limitations in DRAM performance improvement, the easiest way to improve the performance of the memory system remains that of cache improvement. The problem of improving cache performance can be approached from the point of view of the cache itself (hardware approach), or by optimizing the way software uses the cache (software approach). Both approaches are considered here, with emphasis placed on hardware methods. Several studies of simulation-based work focussing on cache performance under multiprogramming workloads are also described. The chapter concludes with a description of the RAMpage hierarchy.

4.2 Speeding up DRAM

4.2.1 Introduction

Since an improvement in DRAM speed reduces the CPU-DRAM speed gap and therefore the cost of a miss, this approach competes with context-switching-enabled RAMpage. However, improving DRAM performance offers a significant challenge. Firstly, although improvements have been made in increasing bandwidth, not much can be done about access time. Secondly, there is also a limit to bandwidth improvement, imposed by the speed the CPU-DRAM bus can run at. A few reasonably successful approaches to bandwidth improvement are briefly looked at here.

4.2.2 Approaches to DRAM bandwidth improvement

Interleaving accesses to different banks of DRAM is a simple means of bandwidth improvement. The idea is that memory accesses are sent to different banks in parallel without waiting for each to complete first. Bandwidth is increased by a factor of the number of banks

that are accessed in parallel, but memory upgrades must be done in larger increments [Hennessy and Patterson 1996].

RamLink uses a ring architecture instead of a bus for the memory system. The time to transfer data is reduced and a bandwidth of 500MB per second is claimed for this arrangement [Gjessing *et al.* 1992].

Placing caches directly on DRAM modules is an approach which favours machines on which traditional caches don't work well, such as vector supercomputers where a single operation is performed on a large working set [Hsu and Smith 1993].

Synchronous DRAM (SDRAM) supports rapid transfers after the initial one, with a theoretical bandwidth of 1500MB per second using a 128 bit bus. The initial access is slow, but thereafter transfers occur at the speed the bus is clocked at [IBM 1997].

Rambus is currently poised to take the place of SDRAM. The direct Rambus design offers the same bandwidth as SDRAM with the advantage of smaller increments being required by using a narrower bus clocked at a higher speed. It is also possible to increase bandwidth by using multiple channels, but latency is not improved [Crisp 1997].

4.3 Improving Cache Performance

4.3.1 Introduction

Caches have traditionally been used to bridge the CPU-DRAM speed gap. However, the growing gap presents increasing challenges for cache designers. This section summarizes the approaches available for improving the basic design of caches.

This can be roughly divided into *hardware*, *software* and *compiler-based* methods. Most hardware methods aim to reduce the miss rate of conventional caches, and as such, can be seen as competing with the RAMpage hierarchy. Software and compiler-based approaches both try to structure code in a way that minimizes cache misses and are complementary to the RAMpage model with context switching on misses, as they may improve performance when used with it. These approaches are looked at in turn in this section, with an emphasis placed on the hardware approach.

4.3.2 Hardware methods

Changing cache parameters such as associativity, block size and cache size has an effect on the cache miss rate and cycle time. An enormous amount of work has been done in this area, and improvements are still being made. However, cache performance improvement is becoming an

increasingly tricky issue, with many tradeoffs, and is still ultimately limited by the cache miss penalty. Reducing this penalty is not easy to achieve.

The miss penalty T_m can be quantified as the total time for a block to be read from memory, given as:

$$T_m = ac + \frac{bs}{tr}$$

Where *ac* is the access time, *bs* is the block size and *tr* is the transfer rate between the cache and memory [Przybylski *et al.* 1988]. Since *ac* is the access time of DRAM, improving this is not possible since this is a fundamental problem of DRAM. Modifying *bs* will not provide a solution either since varying it too much either way impacts negatively on a particular cache design parameter (see 2.3.3). Therefore *tr* is the only variable amenable to improvement, showing that improving the bandwidth of DRAM is the best way to improve its performance. However, the problem of a high initial access latency remains.

An important development arising from the different cache design tradeoffs is the use of multiple levels of cache. The primary cache (L1) size is usually constrained by the available processor real estate so it tends to remain small. Moreover, the primary cache needs to have a low cycle time consistent with that of the CPU, thereby further limiting its size. This means it may have a relatively high miss rate. To reduce the effect of this miss rate a secondary cache (L2) is placed between the primary cache and DRAM memory. The secondary cache is larger in order to improve the miss rate but has a higher cycle time [Przybylski *et al.* 1988].

It is desirable to implement the largest on-chip primary cache that has an access time within the processor's cycle time. However, increasing the size of the primary cache increases its cycle time. Wilson and Olukotun [1997] have suggested techniques to mask the increased cycle time, such as multi-ported caches and pipelined caches. Making caches multi-ported and using multiple banks is similar to interleaving DRAM. Splitting the cache into banks allows multiple cache accesses to be overlapped if the accesses are to different banks.

In general, as the penalty for a miss increases, adding complexity (and cost) to the hardware to achieve fast hits with more associativity becomes more worthwhile. An improvement which can reduce misses without adding to the complexity of achieving fast hits is a *victim cache*, a small additional cache which contains recently replaced blocks [Jouppi 1990].

Another alternative to full associativity is the idea of *column associative caches* [Agarwal and Pudar 1993] which allow an alternative location for a block without the overhead of associativity in the best case for a hit.

While a reduction in the miss rate appears to offer the best potential for performance improvement, especially as the CPU-DRAM speed gap grows, hiding latency is also a common strategy. The probability of a future miss can be reduced by implementing non-blocking prefetch instructions [Chen and Baer 1992, Ki and Knowles 1997]. Non-blocking misses in a superscalar CPU allow other instructions to complete while waiting for a miss [Ki and Knowles 1997, Belaneh and Kaeli 1996].

Although some of these approaches compete with the RAMpage architecture, others complement it. The victim cache concept can be implemented as an extension of the page replacement strategy, using a conventional operating system approach: when a page is replaced, it is moved to a *standby page list*; the page which is on the list longest is the one actually discarded [Machanick *et al.* 1998, Crowley 1997].

Approaches for implementing associativity cheaply can be considered to compete with RAMpage. The conventional architecture of limited associativity implemented in hardware has up to now been capable of meeting design goals of speed and cost and is therefore the standard against which RAMpage is judged in this report. Taking a context switch on a miss can be thought of as similar to a non-blocking cache.

4.3.3 Software methods

As the memory access penalty increases, application performance becomes more sensitive to cache misses. This is especially true for applications with data sets larger than the cache size that do not exhibit a high degree of memory reference locality.

Tiling, also known as blocking, is a well-known software technique to improve the data locality of applications by restructuring a program to re-use certain blocks of data that fit in the cache. Tiling can reduce cache misses and any level of the memory hierarchy can use it [Lam *et al.* 1991].

Reorganizing software with the aim of making better use of caches is another way of tackling the problem. This involves optimizing programs for spatial and temporal locality. For example, *Memspy* is a software package that provides programmers with the means to use locality more effectively [Martenosi *et al.* 1995].

4.3.4 Compiler-based methods

The compiler can assist in restructuring code to improve its memory locality. Most efforts are directed at optimizing loops since this is an area with potential for locality improvement. This is however a static optimization and is limited in this respect. In addition, some classes of programs are more amenable to this type of improvement than others [Bacon *et al.* 1994].

4.4 Multiprogramming Effects on Caches

4.4.1 Introduction

Caches, being the portion of the memory hierarchy most amenable to improvement, have been subjected to numerous studies using simulation as the means of evaluation. Several studies which have focussed on operating system and context switching effects in multiprogramming environments are given a brief overview in this section, serving to illustrate some of the findings and approaches used in the past. The section ends with a summary of findings with relevance to context-switching-enabled RAMpage.

4.4.2 Previous research into multiprogramming effects on caches

Earlier work focussed on small (less than 64K) caches for mainly single-process user programs and either roughly modeled the effects of multiprogramming and system references or excluded them altogether. This omission was not significant for small caches, as was shown by the simulation results of Smith [1982], which correspond reasonably well with the actual measurements of systems [Clark 1983].

As applications and cache sizes grew during the 1980s new techniques were needed to implement the larger simulations required to accurately model these systems. Moreover, the effects of multitasking on the larger caches could not be ignored anymore. Trace-based simulation of larger caches required much larger traces which were costly to generate and store. For example, Agarwal *et al.* [1988] analyzed the performance of large caches in real operating system and multiprogramming environments. They used a technique called ATUM (Address Tracing Using Microcode) to obtain traces of large multitasking workloads. It was shown that both the operating system and multitasking activity significantly degrade cache performance, especially for large caches.

The proliferation of ever larger multiprogramming systems meant the principle of locality on which caches relied was compromised to an increasingly greater degree. Studies using simulation were undertaken to better understand this effect. Two studies of the effects of context switching on cache performance are described below, the first shows how

multiprogramming changes the way a cache is referenced and the second focuses on the performance impact of multiprogramming.

An analytical model to characterize the behaviour of a cache under a multitasking workload was developed using results from trace-driven simulation [Thiebaut and Stone 1987]. From this study, multiple *cache-reload transients* were observed when running processes were switched periodically in a multitasking system (see figure 4). This occurs when the first instructions in an interrupting process cause a large number of cache misses as the process brings its working set from main memory into the cache, thereafter, fewer misses result in a leveling off of the miss rate.

In this model of cache behaviour, the active portion of a process present in the cache is known as its *footprint*. When the size of the cache is very large compared to the size of the process footprints, conflicts between the footprints are small. Consequently, the cache-reload transient experienced by the processes as they resume execution is negligible. If on the other hand the cache is small relative to the process footprints, there will be greater competition for cache blocks as each process erases the other's footprints, resulting in a high reload transient.

This model divides cache behaviour into two phases: the reload transient and the steady state, as opposed to just the average miss ratio. In other words cache misses tend to occur in bursts between which are relatively long periods virtually free of misses.

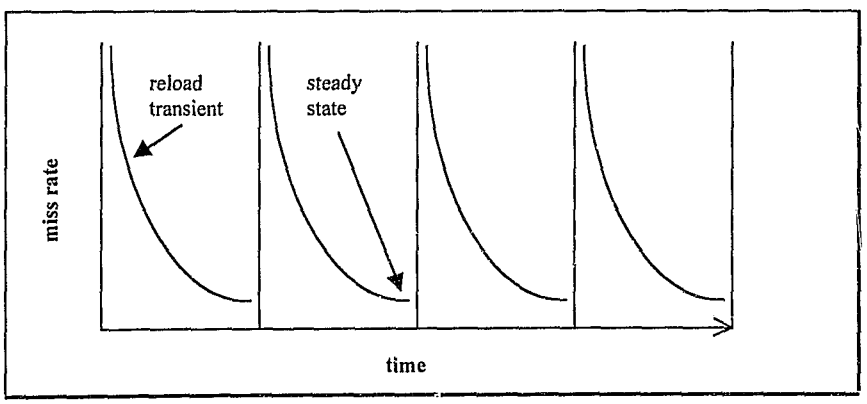

***Figure 4*: Cache behaviour after a context switch.**

To determine the extent of the impact of multiprogramming on cache performance, Mogul and Borg [1991] looked at how the cache hit rate varies after a context switch. Address traces from a multi-tasking operating system were generated and fed on the fly to a cache simulator. By marking the output of the simulation whenever a context switch occurred and then aggregating

the post-context switch results of a large number of context switches, the cache performance reduction caused by a context switch was estimated. It was shown that the net cost of a context switch due to increased cache misses was thousands of cycles or tens to hundreds of microseconds, comparable to the time it takes to send or receive a network packet.

4.4.3 Putting it in perspective

Multiprogramming is an important way of ensuring computer resources (in particular CPU time) are optimally used. However, context switching impacts negatively on caches and this effect needs to be understood, particularly as an increased rate of context switching is advocated in this research.

The reason for bad cache performance in the presence of context switches in the study by Mogul and Borg [1991] is due to conflict misses between processes and capacity misses because the cache is too small to hold the working sets of all the processes. From the study by Thiebaut and Stone [1987], increasing the cache size (to accommodate more process footprints) should help to solve this problem. In addition, increasing the associativity (to reduce conflict misses) would greatly benefit the memory performance of a system in which frequent context switches occur.

For the purposes of the research presented in this report, a high degree of associativity is crucially important in keeping conflict misses to a minimum, since the rate of context switching is greatly increased if a switch is taken on each SRAM miss. Given a fully associative cache, the number of process footprints able to be simultaneously accommodated (degree of multiprogramming) is dependent on the cache size. Thus, a larger cache is desirable if the system typically runs many simultaneous processes. The RAMpage hierarchy is ideal in that it is fully associative and able to scale with increasing SRAM size without an increase in the access time [Salverda 1997].

4.5 Overview of the RAMpage Hierarchy

4.5.1 Introduction

The RAMpage hierarchy [Salverda 1997, Machanick *et al.* 1998] has been proposed as a possible solution to the CPU-DRAM speed gap. The main benefit offered is a reduction in the number of accesses to DRAM, but also significant is the ability to pin operating system code, such as TLB miss handling and context switching code, in SRAM. This section describes the structure and tradeoffs of this hierarchy, as well as simulation performance results.

4.5.2 Basic RAMpage structure

The RAMpage hierarchy differs from a standard hierarchy in the way the specific levels of the hierarchy are managed. Figure 5 illustrates the traditional versus the RAMpage hierarchy.

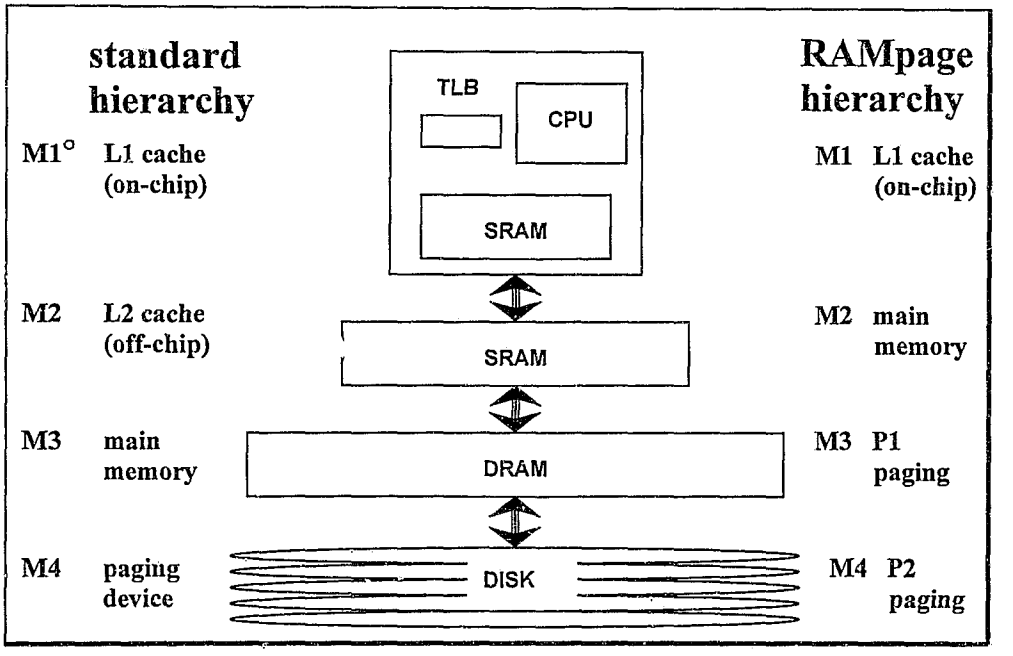

Figure 5: **A standard hierarchy vs. a RAMpage hierarchy.**

Implementing the RAMpage hierarchy does not require many hardware changes, other than the elimination of L2 tags and possibly modifications to improve performance. However, modifications to the operating system are required as an additional level of paging is introduced.

Paging from SRAM to DRAM differs from DRAM-disk paging in terms of the cost of a miss. Miss costs are significantly lower for SRAM–DRAM paging since the cost of a disk access is much higher than that of a DRAM access. This implies different tradeoffs regarding the page size and replacement algorithm. A more complex algorithm rather than more misses favours DRAM-disk paging, while the opposite is true for SRAM-DRAM paging. The optimal page size is likely to be smaller in SRAM-DRAM paging.

4.5.3 RAMpage simulation results

Trace-based simulations were performed by Salverda [1997] to compare RAMpage hierarchy performance to standard hierarchy performance. Page size (block size in the standard

hierarchy) was varied to find an optimal size for each hierarchy, and processor cycle time was varied to model the effects of an increasing CPU-DRAM speed gap on each configuration. For the sake of simplicity and comparison, clock frequency figures in this model represent instruction execution rate, which includes multiple instruction execution per clock cycle rather than pure clock speed. All performance figures here are in terms of clock frequency with an instruction issue rate of one per cycle. For example, a 500MHz machine with an instruction issue rate of two per cycle would be represented here as a 1GHz machine with a single instruction issue per cycle.

To illustrate the lower SRAM miss rate of the RAMpage organization, a simulation was performed with the CPU speed fixed at 200MHz and a 100MHz bus connecting DRAM to SRAM. DRAM is modeled as synchronous DRAM (SDRAM). Table 1 shows the improvement of the RAMpage hierarchy over the standard hierarchy in reducing the number of read and write references to DRAM.

SRAM block/page size (B)	Percent improvement of RAMpage over standard design
128	41
256	59
512	73
1024	84
2048	93
4096	96

***Table 1*: Reduction of DRAM references (read and write) by RAMpage over the standard hierarchy.**

The potential of RAMpage is evident from the large improvement in miss rate performance. This improvement occurs because of the increased associativity of RAMpage as well as the improved replacement algorithms made possible by a software implementation. Of particular interest is the fact that the relative improvement increases as the block size increases. However, this only shows that overall performance has the potential to improve, since working against this improvement is an increase in the miss cost as the page size increases.

Due to these effects, overall improvements in performance for RAMpage over the standard hierarchy are 3% for the base 200MHz system. This increases to 17% as the CPU-DRAM gap widens. Most importantly, the RAMpage hierarchy scales with an increasing CPU-DRAM gap much better than the standard hierarchy does.

4.5.4 Feasibility of context switching in the RAMpage hierarchy

Initial RAMpage simulation results are promising. The hierarchy has been shown to substantially reduce the SRAM to DRAM miss rate when compared to the standard hierarchy. Although these gains are at the expense of a much higher miss penalty, an overall performance improvement is still observed. If the effect of this high miss penalty can somehow be reduced, the benefit of the low miss rate can be better utilized. Thus, there appears to be significant potential for context switching to increase CPU utilization and mask the miss-handling overhead. This is further expanded upon in this section.

Table 2 shows miss penalty figures for varying SRAM page sizes against varying CPU performance values.

SRAM page size (B)	200MHz	500MHz	1GHz	2GHz	4GHz
128	28	70	140	280	460
256	44	110	220	440	880
512	76	190	380	760	1520
1024	140	350	700	1400	2800
2048	268	670	1340	2680	5360
4096	524	1310	2620	5240	10480

***Table 2*: RAMpage miss penalties in processor cycles for varying SRAM main memory page sizes and processor speeds.**

This data is arrived at in the following manner: The M3 implementation in the RAMpage hierarchy (main memory in the standard hierarchy) is based on SDRAM with an initial access time of 50ns, subsequent accesses occurring once every bus cycle and the bus cycling at 10ns (100MHz bus). The bus between M2 and M3 is set at a width of 128-bits, so a 512B page requires 32 bus cycles to transfer (64 for a 1KB-page size).

This translates into the following M2 miss penalty:

- 1 bus cycle to transfer the address to DRAM.
- 5 bus cycles to perform the first read.
- 31 bus cycles to transfer the result of the previous read and simultaneously read the next 128 bits.
- 1 bus cycle to transfer the last 128-bit portion of the M2 page.

The total cycles consumed are 38. Since 1 cycle = 10ns, this gives an M2 miss penalty of 380ns. Assuming a 2ns (500MHz) CPU, this represents a miss cost of 190 CPU cycles. If the

CPU is a modern, 4-way superscalar machine capable of a peak instruction issue rate of 4 per cycle, this represents a maximum of 760 instructions. For a 1KB-page size, this figure is almost double.

Considering the fact that there is a tradeoff between bus width and the speed the bus can be clocked at, (speed is limited due to factors like signal cross-talk [Louri and Sung 1994]) buses are not scaling as fast as CPUs. The strength of SDRAM lies in its burst mode capability, being able to be clocked at the speed of the bus once the initial access has been made, but this is still limited by the bus cycle time.

Current processors are capable of achieving performance figures in the 1GHz range, whereas performance in the region of 4GHz should be achievable in the near future. This means that if a page size of 4KB is used, the miss penalty could be as high as 2620 cycles for a currently available CPU. This increases to 10480 cycles for the 4GHz processor.

These miss values are, if anything optimistic, since the M2-M3 bus is wide and fast by today's standards (today's commodity bus is 64 bits running at 100MHz). A slower bus, such as 75 MHz, can be shown to dramatically increase the miss penalty.

RAMpage performance is strongly dependent on page size. Larger page sizes favour the RAMpage hierarchy because TLB miss and page fault-handling overheads are much bigger with small pages than with larger ones. However, although the number of references to DRAM are the lowest with a 4KB page size, the RAMpage hierarchy performs best with a block size of 1KB, thereafter the performance decreases slightly (see figure 6). This is because the larger pages incur a higher miss penalty, which offsets the gain from the reduced miss rate.

It is at this point that bringing in context switching may improve performance the most. Because of the much lower miss rate of the large block sizes, there is increased potential for performance improvement if the greater overhead associated with larger pages can be tolerated.

4.6 Summary

Lagging DRAM speed has prompted the development of a number of techniques aimed at improving its bandwidth, since this is the only aspect of DRAM amenable to significant improvement. Many existing solutions are proprietary and expensive.

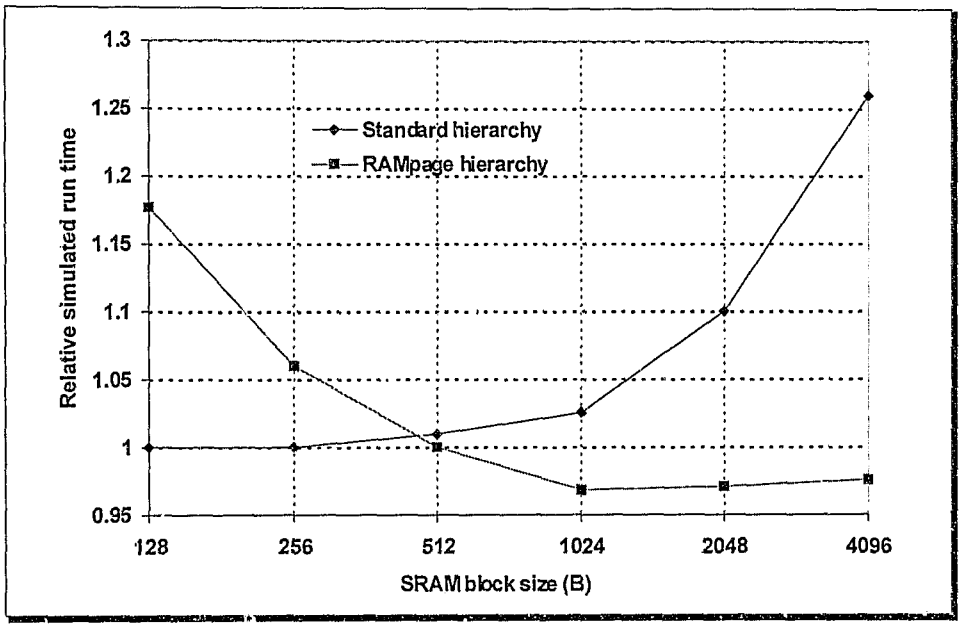

***Figure 6*: Relative performance of the RAMpage and standard hierarchies.**

Synchronous DRAM (SDRAM), however, has largely replaced the previously popular Fast Page Mode (FPM) DRAM and Enhanced Data Output (EDO) DRAM and consequently the price has dropped to below that of EDO due to higher volume production.

Rambus offers slightly better performance than SDRAM and is a candidate to replace it in the near future. It is also a potentially cheaper option, requiring fewer units to be installed when upgrading. However, improving the time for the first DRAM access will remain a significant problem until a better, cheap enough DRAM competitor without this limitation is developed. Until then, the addition of latency-hiding SRAM is necessary.

Caches are crucial for bridging the performance gap between the CPU and DRAM. The growing gap presents increasing challenges for cache designers. The traditional cache design goal is to produce a cache which delivers high performance for a wide range of software, and to structure the software to use the cache more efficiently.

The conflicting goal of a cache is to reduce the miss rate and/or mask its miss latency while maintaining or improving its cycle time and implementation cost. Caches are reaching limitations in this respect and improving the way software uses the cache is a tricky issue with limited potential, particularly when considering current software trends.

Although much has been done to reduce the cache miss rate, factors like code bloat and object-oriented programming are working against caches, ensuring that the frequency of cache misses won't decrease to a negligible point. Since the relative cost of a DRAM access is expected to increase substantially as CPU performance improves, a means of avoiding a costly CPU stall will be necessary to maintain optimal system performance.

The RAMpage hierarchy has been presented as a promising alternative approach to dealing with an increasing CPU-DRAM performance gap. Managing a cache in software, as opposed to hardware has been shown to substantially reduce the miss rate, but at the expense of added complexity in handling misses and a resultant increase in the miss penalty. This makes context switching while a miss is being handled an attractive possibility, and thereby justifies further investigation.

The following chapter considers the issues involved in adding context switching to the RAMpage architecture and goes on to explore ways of reducing the context switching overhead.

Chapter 5 – Context Switching and the RAMpage hierarchy

5.1 Introduction

As the cost of a cache miss to DRAM increases, processor performance will be increasingly restricted by the memory system. Work on reducing cache misses will delay the problem, but as was noted in section 3.2, this approach has limited potential.

Context switching on main memory page faults to disk is a technique used by operating systems to improve processor efficiency. Since the CPU-DRAM speed gap is growing to the point where a DRAM access costs potentially thousands of instructions, it is reasonable to extend this idea to SRAM misses to DRAM.

Although there are obvious differences in the comparison to page faults to disk, the following issues are similar [Machanick *et al.* 1998]:

- The peak disk transfer rate is significantly higher than the rate for a small transfer because of its high latency; DRAM shares this property, although with significantly different numbers.
- The amount of time lost due to a miss is large enough to create space for doing useful work if other processes are available.

The paging system and associated replacement algorithm of the RAMpage hierarchy have been shown to achieve fewer DRAM references compared to the standard hierarchy. As the CPU-DRAM speed gap widens, the RAMpage hierarchy is less likely than the standard hierarchy to suffer a severe degradation in performance, and is thus more scalable. Because the lowest RAMpage miss rates occur at the cost of a high miss penalty, context switching is an attractive strategy. In addition, the RAMpage hierarchy provides natural support for context switching on DRAM accesses by treating SRAM as a paged memory.

This chapter gives an overview of the page fault context switching model, and investigates the tradeoffs and performance implications of context switching on page faults under the RAMpage hierarchy. Assumptions are made to simplify the simulations in this study and real system implementation issues are raised. Lastly, techniques for reducing the cost of a context switch are examined.

5.2 The Page Fault Context Switching Model

A simplified context switching model consists of a queue of processes, with each process in one of two states: *ready* or *blocked*. Processes in the ready state are waiting to be given the CPU, while blocked processes are waiting for a page fetch from DRAM. A process is run until either a page fault is experienced or its time slice expires. Thus, if the CPU burst time of the process (see section 2.4.6) is greater than the time slice, the CPU will be preemptively switched to another process to prevent one process from starving the others. Round robin scheduling is used.

All information related to each process is stored in the relevant process control block (PCB) entry. This is divided into:

- The process state, which includes the program counter (PC), stack pointer (SP), processor status word (PSW) and the registers.
- Accounting information, which includes the processor state (*blocked* or *running*), CPU time, cost, reason for suspension and memory management data structures.

A page fault context switch in the RAMpage hierarchy, operating on the same principle as a page fault to disk, follows the sequence as outlined in figure 7. Assume process *A* experiences a page fault resulting in a trap to the operating system. Its context and related run-time data are immediately saved to its process control block entry. After the cause of the interrupt has been determined to be a page fault, the memory manager is invoked, which locates the needed page in DRAM and initiates a transfer. At this point, the memory manager (MM) state is saved and the next ready process, *B* in this case, is selected by the scheduler. Process *B*'s state is restored from its PCB entry and *B* is run until an interrupt from the memory system signals that the page transfer is complete. The state of *B* is then saved and the state of MM restored. The page table is then updated to reflect the new page brought in, after which the state of *A* is restored and the execution of the instruction which caused the fault resumed.

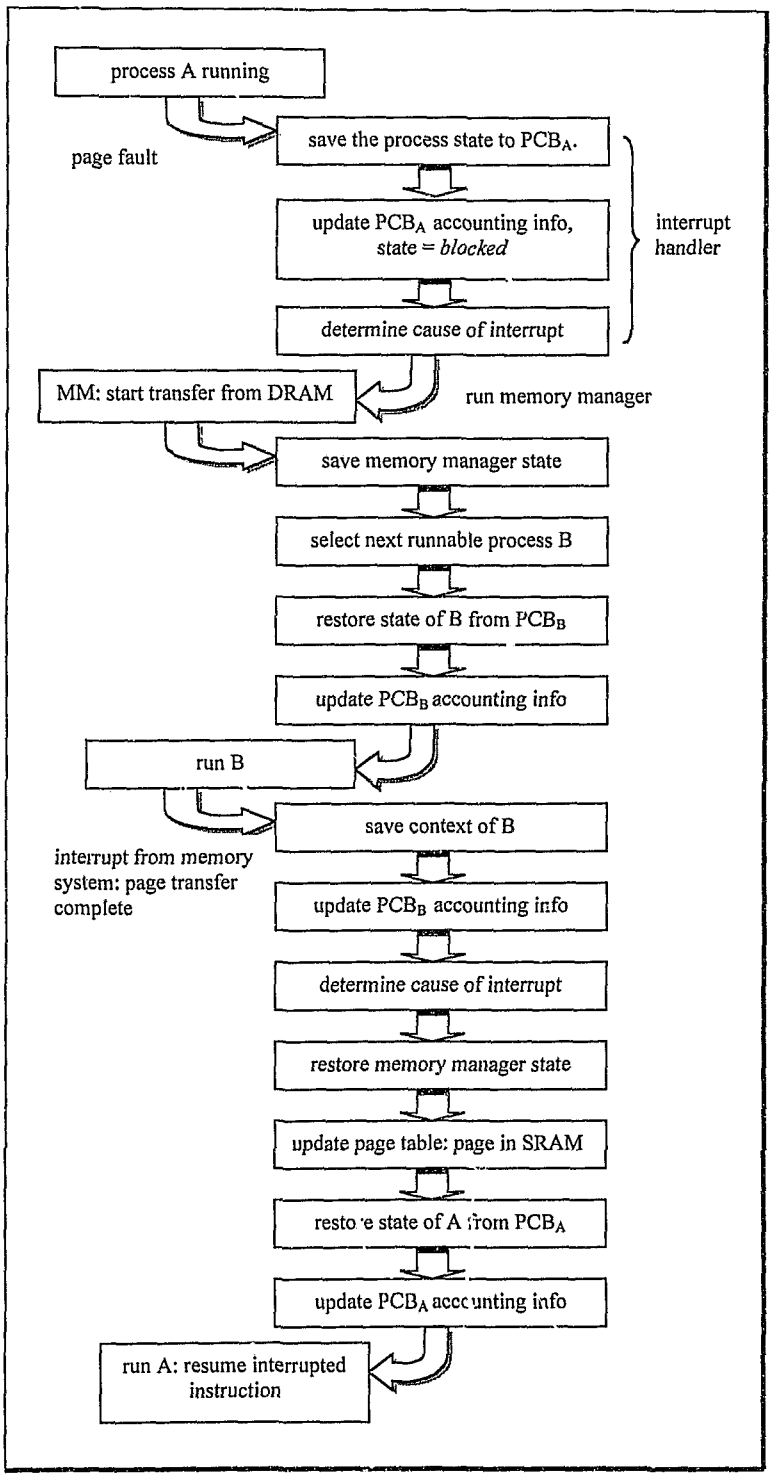

***Figure 7*: Sequence of events in a RAMpage page fault context switch.**

5.3 Performance Tradeoffs

The benefit offered by switching to another process on a DRAM access is an increase in the processor utilization, resulting in more work being done in a unit of time. This must be weighed against the cost of the switch.

As long as the average time taken to perform a context switch (context switch overhead) is less than the average time the CPU spends stalling on DRAM accesses, a performance gain is possible. This represents the tradeoff between increasing the CPU utilization due to overlapping DRAM access delays with useful computation, and the overhead associated with context switching.

A negative secondary effect may be evident in an increase in the SRAM miss rate (capacity misses) due to the increased rate of context switching. Fortunately, the fact that RAMpage is fully associative means there can be no conflict misses, which would be expected to be the greatest miss rate casualty of increased context switching in a non-fully-associative cache (see Chaudhry and Li [1994]). An increase in capacity misses may occur simply because more process footprints need to be held in the cache simultaneously, but this effect is not as significant as the effect on the conflict miss rate would be in a non-fully-associative cache. However, this is an argument for slightly larger SRAM sizes in a context-switching-enabled RAMpage implementation.

Due to the ability to pin context switching code in SRAM, a lower context switch cost can be guaranteed in the RAMpage hierarchy when compared with the standard hierarchy. This comes at the small cost of less available SRAM storage, but the benefits can be expected to substantially outweigh this cost.

5.4 Assumptions and Implementation Issues

This research does not attempt to focus on any architecture or operating system in particular. Since the cost of a context switch is highly dependent on certain system characteristics and may vary widely from system to system, a number of assumptions are necessary to simplify implementation and focus on the problem. This section outlines these assumptions as well as a few issues to consider if context-switching-enabled RAMpage were implemented.

Establishing the amount of overhead associated with context switching is not a trivial matter, since this depends not only on the particular processor architecture, but on the operating system as well. For example, some architectures have an instruction that saves all the registers, while

others require several instructions for this. Thus, figures on the cost of a context switch on modern systems are not readily available.

For the purposes of this research, the cost of a context switch was arrived at by looking at typical operations which are performed by the operating system on a switch. A straightforward implementation of these operations based on operating systems theory was used to simulate a context switch. This switch simulation considers events like the saving and restoring of the processor state but does not include operations such as the running of scheduling algorithms. Process selection is kept as simple as possible, being just the next runnable process in the queue.

Using these assumptions, sample context switching code when traced produced 418 references, and this was taken to be the cost of a switch. Since there are ways to reduce this switching overhead, and given variations on different systems, this cost is taken only as an estimate based on operating systems theory, and is not reflective of any system in particular. Obviously, the lower the switch overhead, the more successful this research will appear to be. However, the aim of this research is not to reduce the cost of the switch, but rather to show how the RAMpage hierarchy performs with a fixed switch cost under varying page sizes and CPU speeds.

Given that implementation of the RAMpage hierarchy requires modifications to the operating system, including context switching support on page faults in these modifications is trivial. However, making improvements to context switching time implies more extensive modifications and possibly hardware support. As the CPU-DRAM speed gap widens, this may become a more attractive proposition and is therefore considered next.

5.5 Reducing the Cost of a Context Switch

5.5.1 Introduction

Success of the RAMpage context switching approach relies on the average cost of context switches being lower than the average cost of SRAM misses. The growing CPU-DRAM speed gap has the effect of increasing the miss cost relative to the switching overhead and thereby making context switching on misses more successful. This can also be achieved by lowering the switch overhead, thus methods which may reduce switching overhead are worth considering. Although not the objective of this research, reducing the context switch cost will enhance context-switching-enabled RAMpage. This section looks at processor architectures

designed to speed up switches, followed by ways in which operating systems can more efficiently switch processes.

5.5.2 Multithreaded processor architectures

Processor architectures that perform a context switch in very few cycles have been proposed and developed. The motivation behind this work is the high cost of a cache miss in a shared-memory multiprocessor. A cache miss in these systems necessitates a shared bus or network access to the shared memory, resulting in a significant drop in processor utilization. Performing a context switch rapidly means the CPU does not waste much time on switching, and the main tradeoff is between better processor utilization and the higher cache miss rate which results. The general idea is to have multiple hardware contexts on the processor and simply switch between them. Some examples of studies of this approach follow.

The MASA architecture [Halstead and Fujita 1988] assigned each processor a fixed number of hardware task frames. Each task frame was capable of storing a complete process context and consisted of a set of auxiliary registers (such as the program counter) and a set of general-purpose registers. Since the number of processes may exceed the number of task frames, the process contexts were allowed to overflow into memory.

The benefits of multiple hardware contexts per processor was investigated by varying the number of contexts between one and four [Weber and Gupta 1989]. It was shown that up to 80% higher processor utilization could be achieved if the context switch overhead was kept low. The gain in processor utilization when increasing the number of contexts also depends on the nature of the application, one that already has a high CPU utilization does not benefit greatly from an increase in the number of contexts.

Reducing the time taken for a context switch requires hardware that is more complex, and beyond some overhead value, multiple contexts do not help anymore, since a long latency operation will complete before the context switch is accomplished. As the latency of an operation such as a cache miss increases, the benefit of multiple contexts over a single context increases. Thus, the best gains are made under the following conditions:

- The architecture has large read and write latencies.
- The context switch overhead is low.
- The cache interference due to multiple contexts is small.

In a multiprocessor, the cache miss latencies can be expected to increase as more processors are added. Using a fixed number of hardware contexts with low overhead ensures that the

context switch overhead is kept to a minimum, and cache interference can be kept low by using large caches.

Chaudhry and Li [1994] describe a simulator model for a shared-memory multiprocessor in which multiple contexts are catered for on each processor. They explore the tradeoff between increased processor utilization due to overlapping communication delays with useful computation and the higher cache miss rates. It is noted that increasing the number of contexts per processor increases processor utilization up to a point where the increased cache miss rate reduces performance.

5.5.3 Thread implementation issues

Using threads in a way that serves to reduce the context switch overhead is a software-oriented approach to faster context switching.

There are two main ways to implement a threads package: in user space and in the operating system kernel [Tanenbaum 1995]. A user space implementation allows threads to be implemented in an operating system that does not support threads, as well as offering the benefit of very efficient switching between threads. With kernel managed threads, an expensive call to the kernel is needed for switching threads, and a context switch occurs between different address spaces, requiring more extensive updates to process tables.

Threads in a user space run on top of a collection of thread management procedures. When a thread does something that results in a delay it calls a runtime procedure which checks to see if the thread must be suspended. If so, it saves the thread's state and selects an unblocked thread to run. If the machine has an instruction to store all the registers and another one to load them all, the entire thread switch can be done in a handful of instructions. Performing thread switching like this is at least an order of magnitude faster than trapping to the kernel and is a strong argument in favour of user-level threads packages.

Another advantage of user-level threads is that they allow each process to have its own customized scheduling algorithm. However, user-level threads have a major problem in that blocking[3] system calls are difficult to implement, and unfortunately, this type of call is very common in threaded applications. Letting the kernel manage threads solves the blocking problem, but performance suffers.

Various researchers have attempted to combine the good performance of user threads with the blocking advantage of kernel threads. In an approach devised by Anderson *et al.* [1991] called

[3] A blocking call is one that results in the calling process stalling until a response is given.

scheduler activations, efficiency is achieved by avoiding unnecessary transitions between user and kernel space while maintaining the functionality of kernel threads.

5.5.4 Putting it in perspective

An important factor regarding the feasibility and performance of context switching on a cache miss is the overhead involved in performing the context switch. Methods of reducing this overhead can be divided into hardware and software approaches. Software approaches have the advantage of a lower implementation cost, but hardware approaches ultimately offer greater potential for improvement.

Including hardware support for context switching on a processor allows very rapid context switches, and therefore allows context switches on much shorter cache miss delays to be tolerated. Multiple hardware context support can be viewed as a cache for recently used process contexts, in the same way that a TLB is a cache for recently used page table entries. However, this has the major drawback of requiring expensive hardware modifications and processor redesigns and currently is not feasible, but in future as the higher cost of an SRAM miss makes a context switch more attractive, processor architectures may include hardware support for context switching.

A useful approach in the interim would be to improve the performance of a context switch in software. Work on threads package implementations such as the scheduler activations approach by Anderson *et al.* [1991] has the potential to reduce the cost of a context switch without requiring hardware support.

Chapter 6 – Methodology

6.1 Introduction

Measuring memory performance can be done in a number of ways, including *hardware measurement*, *analytical modeling* and *simulation*. Hardware measurement involves tapping into the memory system of the target machine with hardware probes to measure activity. Analytical modeling gives performance estimates based on mathematical models.

Since a new strategy is being proposed and a complete implementation is not possible, simulation is the most appropriate method. Simulation offers the advantages of flexibility and low cost. This chapter starts by reviewing simulation methods in general and trace-driven simulation in particular, then discusses issues relating to traces and applications. Lastly, configuration details of the RAMpage hierarchy simulator are given.

6.2 Traces and Applications

6.2.1 Introduction

There are three main approaches to simulating the cache and memory performance of a computer architecture [Veenstra and Fowler 1993]:

- *Program-driven* simulators perform the simulation as the traced program executes, with the addresses being generated as they would on the machine being simulated. There is no need to store a trace file on disk.
- *Execution-driven* simulation is a special case of a program driven simulation in which the program being simulated is modified by inserting function calls directly into the program code. It requires pre-processing and recompiling the application.
- *Trace-driven* simulators use a trace file containing memory references which are fed to the simulator.

Trace-driven simulation is the most appropriate simulation method for the purpose of this project. It provides more detail than analytical models and allows various configurations to be compared quickly and efficiently. It is much less expensive and more flexible than building hardware. Among simulation techniques it produces the most realistic results since actual programs are used as inputs rather than statistical models [Koldinger *et al.* 1991]. Some background to the trace-driven approach follows.

6.2.2 Generation of address traces

Trace-driven simulation is one of the most common experimental techniques used to explore architectural alternatives. This section describes the two alternative methods of generating traces of running programs. This research uses pre-generated traces for applications, but a trace of the context switching code is generated using Mint (MIPS interpreter [Veenstra and Fowler 1993]).

The collection of address trace data can be divided into *hardware* and *software*-based methods [Borg *et al.* 1990]. Hardware methods involve the use of modified microcode or a special purpose device to intercept and record memory references as they occur. Software methods usually require the traced programs to be augmented with instrumentation code such that address trace data is generated as a side effect of program execution. There are two main problems with the software approach:

- The instrumentation process can alter the behaviour of the traced system.
- Operating system code is difficult to instrument for address tracing.

Hardware methods avoid the problems of software methods by intercepting trace information at a very low level, capturing all activity regardless of the source, and the behaviour of software is unaffected. They also run much faster than software methods, typically up to 100 times faster. This is an ideal method, but unfortunately several properties of current processor design make hardware-based tracing difficult:

- Microcode based designs are not useful because current RISC (reduced instruction set computer) machines do not use reloadable microcode.
- The tendency of current processors to incorporate memory components such as caches and buffers into an integrated microprocessor package. The address references are transferred on submicron structures inside the chip, making them impractical to access with a hardware monitor.
- Modern high performance computers operate at very high clock speeds, making devices that can keep up difficult and expensive to build.

For these reasons, software-based methods are becoming the only way to generate traces [Chen *et al.* 1994, Borg *et al.* 1990].

6.2.3 Length of address traces

Typical trace lengths used in previous simulations of CISC machines with relatively small caches rarely had more than 10 million references [Borg *et al.* 1990]. In order to simulate the behaviour of larger caches on RISC systems accurately, much longer traces are needed for the following reasons:

- Larger caches take longer to *warm up* or reach the point where compulsory misses make up a small portion of total misses. Thus a trace which is too short will show a bias towards compulsory misses.
- A program will have a certain amount of *initialization code* when it starts running which may be quite different from its behaviour further along. If the trace consists of a relatively large amount of initialization code, results will be biased towards this and not the average behaviour of the program.
- Only a few seconds of program execution time result in a trace containing billions of references.

Using long traces results in the following problems:

- *Large storage requirements.*
  Mass storage devices are needed to cope with the large amount of storage needed. This can take the form of tape or large hard disk storage. Hard disks are preferable for performance reasons. Compression techniques such as PDATS [Johnson and Ha 1994] which have been developed specifically for the purpose of address trace compression are very effective in reducing trace storage requirements to reasonable amounts.
- *Computationally intensive simulation.*
  Running the large address traces through the simulator results in lengthy running times. This is alleviated by running multiple simulations with varying parameters on several fast machines with adequate main memory.

6.2.4 Obtaining address traces

Address traces can be generated using an application like Mint. Mint is a program driven cache simulator running on Silicon Graphics Systems under Irix [Veenstra and Fowler 1993]. Unfortunately, using Mint poses several problems: all simulated programs must be statically linked; Mint does not support all Unix system calls; and the programs used should reflect a typical system workload. Other tools could in principle also be used but generally also present

the problem of requiring the traced applications to be in a statically executable form. For these reasons, it is easier to use pre-generated traces if suitable traces are available.

6.2.5 Choice of user programs

Ideally, the programs traced should characterize the average workload of a real system. However, the facilities available for this research place constraints on the degree to which a system can be realistically simulated. In addition the average workload of a system is difficult to define. Not only do the processes on a system vary from one machine to another, but the usage changes with time. Therefore, an approximation of a typical system's workload is constructed by putting together a suite of commonly used applications.

The SPEC [SPEC 1998] suite of programs is a widely used benchmark for comparing the performance of different computers. These benchmarks are also very popular with computer architecture researchers for evaluating the performance of memory hierarchies (see Weiker [1997] for a discussion of the use of the SPEC benchmarks in computer architecture research).

The trace suite selected for this study is taken from programs in the SPEC92 benchmark suite, and comprises 4 integer programs (SPECint92) and 10 floating point programs (SPECfp92) compiled for the MIPS R2000 processor[4]. Four common Unix text utility programs are included in this suite as well (see table 3).

This trace suit is available for download from the Tracebase International Trace Archive at New Mexico State University[5]. In total, 1.1-billion references are run through the simulator in each simulation.

6.3 Configuration of the Simulation

This section describes simulation details, firstly the operation of the scheduler, followed by RAMpage simulator parameters and lastly the simulation structure. The experiment is divided into two parts: the effect of pinning context switching code in SRAM; and switching on page faults. Each is described in this section.

6.3.1 Simulated scheduler

To simulate a multitasking system, multiple address traces are interleaved using a task scheduling simulator. As in a typical preemptively multitasking system, each trace is given a

[4] Note, although the SPEC92 suite has been rendered obsolete by the SPEC95 benchmarks, it is still adequate for the purposes of this experimental study.
[5] Obtainable as <ftp://tracebase.nmsu.edu/pub/R2000>.

time slice in which to run before control is given to the next process. Round robin scheduling is employed since it is commonly used and simple to implement.

All processes are initially marked as *ready*. The scheduler picks the first ready process from the queue and dispatches it, after which one of two things may happen:

- The process runs to the end of its time slice without experiencing a cache miss. The scheduler dispatches the next process in the ready queue.
- A miss occurs, the process is assigned a *blocked* state and the next process in the queue is dispatched. If no process is found in a non-blocked state, the CPU stalls until the page fetch completes and the process unblocks.

Program	Description	Program Type	Total References	Instruction fetches
Cexp	Logic array generator	SPECint92	18,782,177	14,256,512
Comp	File compression utility	SPECint92	10,459,159	8,014,160
Gcc	C compiler	SPECint92	7,458,670	5,689,551
Mdljd	Motion equation solver	SPECfp92	84,233,871	65,000,000
Wave	Particle equation solver	SPECfp92	78,282,009	65,000,000
Ora	Ray tracer	SPECfp92	82,942,488	65,000,009
Alvn	Neural net trainer	SPECfp92	72,814,137	59,027,112
Ear	Human ear simulator	SPECfp92	80,400,251	65,000,001
Mdljs	Motion equation solver	SPECfp92	76,954,695	65,000,003
Swm	Physics computations	SPECfp92	87,416,474	65,000,001
Ucomp	Decompression utility	SPECint92	18,729,855	14,204,142
Su2	Physics computations	SPECfp92	88,755,536	65,000,001
Hydro	Physics computations	SPECfp92	10,985,700	8,248,427
Nasa7	NASA applications	SPECfp92	99,731,796	65,000,000
Awk	Text processor	Text utility	86,435,124	62,834,833
Sed	Stream editor	Text utility	9,752,248	7,717,459
Tex	Text formatter	Text utility	66,829,759	50,288,264
Yacc	Parser generator	Text utility	12,186,384	9,664,466

***Table 3*: Trace suite used in the simulations.**

6.3.2 Fixed simulation parameters

This section describes the parameters used in the RAMpage simulator. The cache and main memory parameters have been selected to reflect a typical high end system available today.

The processor is assumed to be capable of executing one instruction per cycle when it is not waiting for the memory system. The memory hierarchy consists of a primary cache, secondary cache and DRAM main memory, with DRAM assumed to be large enough to avoid page faults to disk.

The primary cache is direct-mapped, virtually addressed and split into instruction and data caches (16K each). The write policy is write back with write allocate miss handling. Block size is fixed at 32B and a full block is fetched on a miss. A dirty miss (block content has been modified) causes the block to be written back to secondary cache. Read hits are assumed to take one CPU cycle while write hits take two. The processor stalls on primary cache misses.

The secondary cache in the standard hierarchy is unified, physically addressed, direct-mapped and off-chip. Write policy is the same as for the primary cache. The cache size is fixed at 4MB and has a block size varying between 128B and 4K. The L1 and L2 cache are connected by a 128 bit bus cycling at one third the speed of the CPU. A miss from the primary to the secondary cache requires 12 CPU cycles while a dirty write miss in the data cache requires an additional 12 cycles.

RAMpage SRAM main memory takes the form of synchronous SRAM which cycles at the same rate as the L1-L2 bus (one third CPU speed). Read accesses have the same cost as the L2 cache access cost while write hits are 25% faster due to the lack of a tag check. Therefore, 12 cycles are required for a read and 9 additional cycles for a dirty L1 write miss. Since tag storage is eliminated, an additional 0.125MB of storage is available, giving a total of 4.125MB.

DRAM is modeled on synchronous DRAM (SDRAM) and is assumed to be infinitely large so that page faults to disk are not modeled. Through a combination of burst mode access and interleaving, SDRAM is capable of sustaining continuous data rates at bus speed. The first access is 5 bus cycles with subsequent accesses occurring every bus cycle. Since SDRAM is currently the most commonly used DRAM, performance is comparable to that of a typical current machine. DRAM is connected to the lowest SRAM level by a 128 bit bus with a cycle time of 10ns.

A simulation of typical operations performed during a context switch (from operating systems theory) has been traced to produce 418 instruction and data references. Thus a context switch is simulated by inserting this trace at the appropriate time.

6.3.3 Simulation structure

The experiment is divided into two separate simulations, simulation A and simulation B. The aim of simulation A is to observe the effect on performance of pinning the context switching code in SRAM. Simulation B forms the focus of this research and investigates the effect of introducing context switching on RAMpage page faults.

In both simulations, the trace set is run through the standard hierarchy and the RAMpage hierarchy. The standard hierarchy serves as a comparison for the RAMpage hierarchy and is identical to it in all respects apart from the manner in which the second level of SRAM is managed. To ensure the experimental work is kept within reasonable limits, the variables are restricted to block size and CPU speed. The values chosen for variables are as in the RAMpage study by Salverda [1997] to facilitate comparison with RAMpage results.

Simulation A: effect of pinning context switching code in SRAM

The RAMpage hierarchy has the important advantage of allowing critical operating system code to be kept in SRAM so that it will never need to be accessed in DRAM. The first simulation attempts to measure this effect by pinning a trace of context switching code in SRAM in the RAMpage hierarchy and comparing the performance with the standard hierarchy in which context switching code present in the cache is subject to normal cache replacement.

For this experiment, CPU clock speed is maintained at 2GHz to represent a high performance processor available in the near future. Block size is varied through 128B, 256B, 512B, 1K, 2K and 4K for both the standard and RAMpage hierarchies, resulting in 6 runs for each hierarchy. With the memory bus cycling at 100MHz, SRAM miss penalties range from 280 CPU cycles for a 128B block to 5240 cycles for a 4K block using SDRAM as main memory.

Context switches in this simulation occur only on clock timeout interrupts at the end of each process's time slice, set at 500 000 references. No context switches are simulated on page faults to ensure only the pinning effect is measured. In both the standard and RAMpage hierarchies, the 418 reference trace is run on each clock timeout to simulate the context switch.

In the case of the standard hierarchy, there is the possibility of much of the context switching references in the L2 cache being replaced by the user process references by the time it is run again on the next context switch. Pinning the context switching code in RAMpage SRAM ensures that this code cannot cause misses to DRAM once the code is in SRAM, but it does reduce available SRAM capacity marginally, potentially causing slightly more capacity misses in user processes. However, this effect is minimal, since the context switching code takes only 0.1% of the total page frames in the worst case (4K pages).

Performance is compared by measuring the average number of CPU cycles spent context switching on each configuration for each hierarchy. When cache misses occur in the standard hierarchy, the cache miss cost is added to the total context switching CPU cycles, increasing the average.

Simulation B: page fault context switch

The principal objective of this research is to investigate the feasibility and performance implications of a full context switch to another process being taken on each page fault under the RAMpage hierarchy. A simulation to satisfy this objective has been performed using the page fault context switching model given in section 5.2. In this simulation, block size is varied as in simulation A, while CPU speed is varied between 200MHz, 500MHz, 1GHz, 2GHz and 4GHz, resulting in 30 runs for each hierarchy.

Whenever a page fault with the resultant page fetch from DRAM occurs, a context switch is simulated by inserting the 418 reference trace of context switching code, the faulting process is marked as blocked and the next process in the ready queue is run. Blocked processes are unblocked and re-join the ready queue when the number of references corresponding to the particular miss cost have elapsed. If no processes are found in the ready queue, i.e. all processes are blocked waiting for a miss to be handled, the CPU stalls with no context switch, as in the standard hierarchy. Simulation continues in this way until all the traces are exhausted.

Measurement is achieved by comparing the total runtime of the new hierarchy with the total runtime of the standard hierarchy. An improvement in the RAMpage runtime is a result of the reduction in simulated CPU cycles needed to process the trace set.

Chapter 7 – Performance Results

7.1 Introduction

To measure the performance of the RAMpage hierarchy under the new context switching strategy, simulations have been conducted on both the RAMpage and the standard hierarchy. As in the initial RAMpage simulations [Salverda 1997], the block size (page size under RAMpage) is varied to identify the performance-optimal configuration for each hierarchy, while processor speed variation serves to model the effects of an increasing CPU-DRAM speed gap. Each simulation has used the same 1.1 billion reference trace set as input.

The remainder of this chapter is organized as follows. Section 7.2 presents the results of simulation A in which the effect of pinning context switching code in SRAM is measured. It is shown that this substantially reduces the average time taken to perform a context switch, particularly for large pages. The results of simulation B – the effect of context switching on DRAM accesses – are presented in section 7.3. This strategy is shown to be beneficial for large page sizes and high CPU speeds. The cause of this improvement is investigated, after which the chapter concludes by demonstrating the improved scalability of this approach with the increasing CPU-DRAM speed gap.

7.2 Simulation A Results

This simulation looks at the effect of pinning context switching code in SRAM by comparing the performance of the RAMpage and standard hierarchies over the range of page sizes, with a CPU speed set to 2GHz and a context switch interval of 500 000 references.

Table 4 presents the average context switch time taken for each configuration. As can be expected, the average switch time for the RAMpage hierarchy remains relatively constant for all page sizes. Since no page faults can be incurred while running context switching code, the small variations noticed are caused by TLB and L1 cache behaviour. The standard hierarchy, on the other hand, may incur SRAM misses during a context switch, and as a result average context switching time is greater. Standard hierarchy performance is the best for small block sizes, with performance dropping off substantially as the block size increases, due to the large increase in block fetch time.

Page/block size (B)	Average switch time (CPU cycles)		RAMpage Improvement (%)
	RAMpage hierarchy	Standard hierarchy	
128	358	428	19.5
256	348	424	21.8
512	351	440	25.3
1024	369	485	31.4
2048	368	616	67.4
4096	360	917	154.7

***Table 4*: Comparison between average switch time for the RAMpage and standard hierarchies.**

The best performance of RAMpage is an improvement of 21.8% over the best performance of the standard hierarchy when taken over the cumulative time spent switching (see figure 8).

The improved context switching time under the RAMpage hierarchy will greatly benefit context switching on page faults to DRAM. In addition, as the page size increases, the switch cost does not increase as occurs with the standard hierarchy. This is an important result for context-switching-enabled RAMpage, since large page sizes with the lowest miss rates are not subject to any increase in the cost of a context switch as is the case with the standard hierarchy.

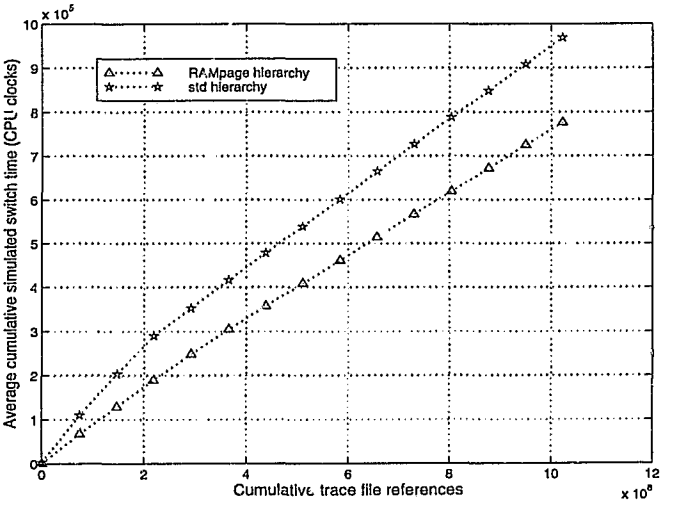

***Figure 8*: Best average switch cost performance for page/block size which gave the best results in each hierarchy: 256B for both the standard and RAMpage hierarchies.**

7.3 Simulation B Results

7.3.1 Introduction

The primary objective of the RAMpage hierarchy is to reduce the total number of DRAM accesses through an improved SRAM hit rate. Although this objective has to a large extent been met, the benefit of a lower miss rate is unfortunately somewhat reduced by the higher miss costs associated with larger page sizes.

By doing useful work on another process rather than stalling on a page fault, it is hoped that the effective cost of a miss can be masked, resulting in fewer processor cycles being lost to the memory system. Overall results indicate that this is so, but a performance gain depends on the page size and CPU speed. This section presents these results, firstly in terms of noticeable improvement – the overall run time and secondly in terms of the underlying causes of the improvement. Since context switching code is pinned in SRAM, context switches do not cause any DRAM accesses in this RAMpage simulation.

7.3.2 Run times and speedup

Table 5 compares the simulated run times for each configuration of context-switching-enabled RAMpage and the standard hierarchy. Two significant trends can be observed:

- As the page size increases (moving from left to right in Table 5), the performance of the RAMpage hierarchy with context switching improves over the standard hierarchy.
- A similar improvement occurs as CPU speed increases (moving from top to bottom in Table 5).

Thus, the break even point in Table 5 occurs roughly to the left of the bottom-left to top-right diagonal, all values below this diagonal are net gains in performance. The greatest gain of context-switching-enabled RAMpage over the standard hierarchy of 68.9% is experienced with the largest page size and CPU speed.

However, the figures in Table 5 are a little misleading in that they do not give the whole picture, since whereas RAMpage is optimal on the large page sizes, the standard hierarchy is optimal on smaller sizes. Thus, when the best case of each hierarchy is considered, the difference between them will be smaller. This is considered later, after the reasons for the performance improvement are examined in a little more detail.

CPU clock speed	SRAM page/block size					
	128B	256B	512B	1K	2K	4K
200MHz	11.8456	11.8550	11.9520	12.1802	13.0246	14.8871
	15.3885	13.1365	12.1850	11.6300	11.5170	11.5755
	-23%	*-9.8%*	*-2%*	*4.5%*	*11.6%*	*22%*
500MHz	4.8492	4.8621	4.9436	5.1331	5.8102	7.3346
	6.1486	5.2666	4.8730	4.7488	4.7112	4.6160
	-21%	*-7.7%*	*1.4%*	*7.5%*	*18.9%*	*37%*
1GHz	2.5172	2.5311	2.6074	2.7840	3.4054	4.8170
	3.0719	2.6313	2.4765	2.3484	2.3148	2.3422
	-18.1%	*-3.8%*	*5%*	*15.6%*	*32%*	*51.4%*
2GHz	1.3510	1.3656	1.4393	1.6095	2.2030	3.5583
	1.5372	1.2987	1.2164	1.1773	1.1660	1.1658
	-12.1%	*4.9%*	*15.5%*	*26.9%*	*47.1%*	*67.2%*
4GHz	0.7680	0.7829	0.8553	1.0221	1.6018	1.8552
	0.7359	0.6496	0.6014	0.5998	0.5843	0.5775
	4.2%	*17%*	*29.7%*	*41.3%*	*63.5%*	*68.9%*

Table 5*: Relative performance of the standard and RAMpage hierarchies, showing elapsed simulated time (in seconds) by each hierarchy to process the 1.1 billion-reference trace set. *Values for the standard hierarchy appear in the upper portion of each row and those for the RAMpage hierarchy below. The last value shows the improvement of RAMpage over the standard hierarchy – positive percentages depict an increase in performance, negative values a decrease.

7.3.3 Reducing CPU cycles

The above observed run time improvements are driven by a reduction in the total number of CPU cycles needed to process the trace suite. One way of measuring this effect is to take the difference between the total context switching overhead and the total time (in terms of CPU cycles) that would have been spent stalling on a page fault had no switching been performed. If the former figure is lower than the latter, fewer CPU cycles will be consumed to process the trace suite, resulting in a speedup. The following two subsections explore these results, first by finding the performance optimal page size, then using this to see what happens when the CPU-DRAM speed gap increases.

7.3.4 Varying the page size

The effect of page size on the performance of context-switching-enabled RAMpage can be seen in figure 9. A CPU speed of 4GHz is simulated, with the reduction in CPU cycles due to context switching shown for the range of page sizes. A page size of 128B performs the worst, with no significant change in performance when compared to the standard hierarchy. Performance consistently improves as the page size grows, with the 4K page clearly the best.

7.3.5 Varying the CPU-DRAM speed gap

The effect of an increasing CPU-DRAM speed gap appears in figure 10. The performance optimal page size of 4K is selected for this comparison, with positive curves indicating a performance gain, and negative curves a reduction in performance.

Firstly, notice that the DRAM access cost of a 200MHz CPU is not high enough to outweigh the overhead of a context switch. However, as the CPU speed increases, so its ability to process the context switch overhead improves. The point at which this overhead is amortized and a performance gain is experienced occurs when the CPU speed for a given fetch delay is sufficient to allow useful instructions to be executed over and above the switch overhead. This point occurs between the CPU speeds of 200 and 500MHz. Since the 4K page is optimal, smaller pages will exhibit performance gains only at higher CPU speeds.

Thus, overall performance is a result of the combination of CPU speed and page size. If the combination of these two factors is sufficient to offset the overhead of switching, performance improves. In other words, if the CPU is sufficiently fast and/or the page size is large enough to cause a sufficiently long fetch delay, the performance of RAMpage with context switching shows an improvement over the standard hierarchy.

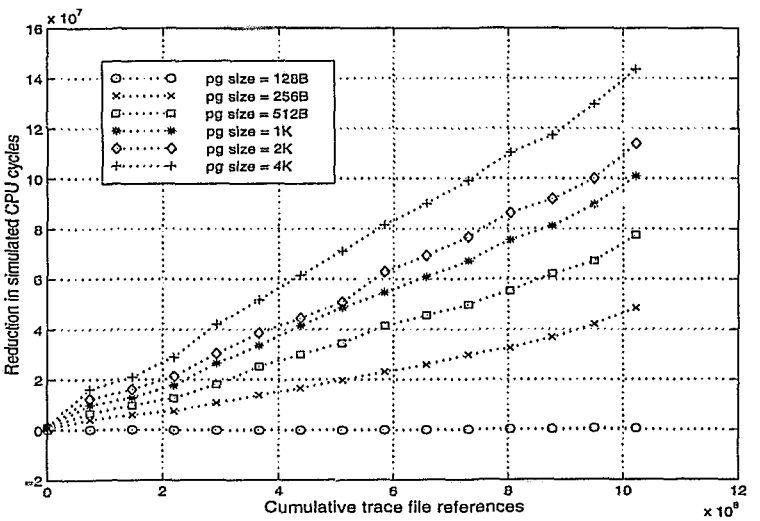

Figure 9*: Page size effect.** ***Simulated CPU cycle reduction due to RAMpage context switching as compared to the standard hierarchy with a CPU speed of 4GHz, calculated from the difference between the total context switching overhead and the total time that would have been spent stalling on page faults had no switching been performed.

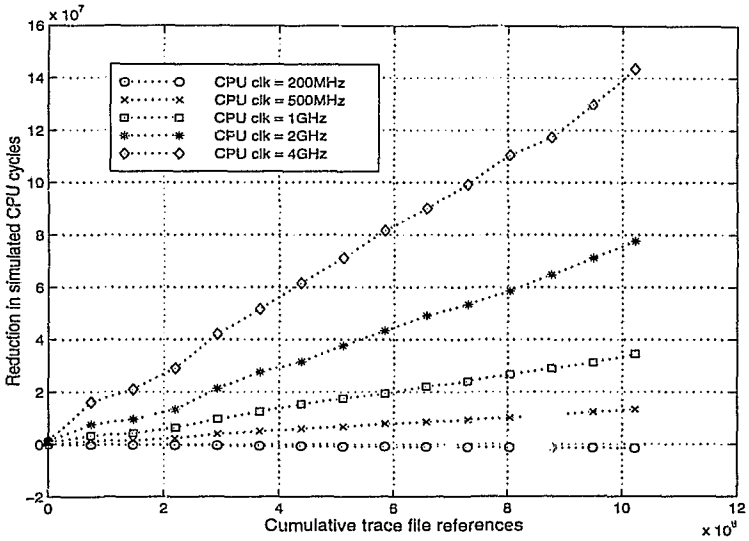

Figure 10*: CPU speed effect.** ***Simulated CPU cycle reduction due to RAMpage context switching as compared to the standard hierarchy with a page size of 4K, calculated from the difference between the total context switching overhead and the total time that would have been spent stalling on page faults had no switching been performed.

7.3.6 Scalability

Figure 10 shows the effect on simulated CPU cycles as the CPU speed is increased. A page size of 4K is used for both RAMpage and the standard hierarchy. The standard hierarchy is optimal at the 128B block size, meaning that if this is used as a comparison the improvement would be a little lower than shown above. However, the trends observed would still be the same as above as the optimal RAMpage page size is still a reasonable improvement over the optimal standard hierarchy block size.

Observe in figure 10 that as the CPU speed increases, so the number of CPU cycles reduced improves by an increasing amount. This demonstrates that the RAMpage hierarchy with context switching is scalable with increasing CPU speed. But how does this compare with base RAMpage with no context switching? Figure 11 shows a comparison between the best case performance of base RAMpage and context switching-enabled RAMpage given as a relative improvement over the standard hierarchy over each variation in processor speed.

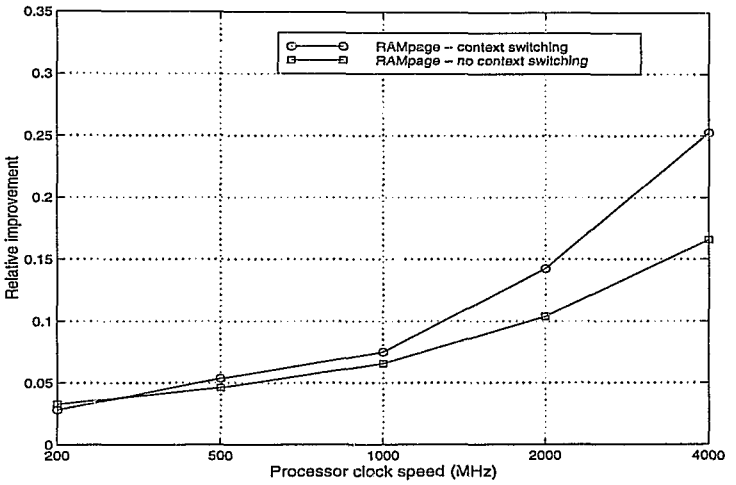

Figure 11*: Relative improvements in performance of best case RAMpage configurations over that of the standard hierarchy.** ***1K RAMpage page size/128B standard hierarchy block size in the non context switching case and 4K RAMpage page size/128B standard hierarchy block size in the context switching case (non context switching data is from Salverda [1997]).

With no context switching on page faults, the RAMpage hierarchy achieves a speedup of almost 17% over the standard hierarchy at the highest simulated CPU speed and a page size of 1K. Performing context switches on page faults has allowed the RAMpage hierarchy to better exploit its lowest miss rate configuration, the 4K page, resulting in a speedup of 25% over the standard hierarchy. This means RAMpage with context switching is (1.25-1.17)/1.17 = 6.8% faster than base RAMpage, with an improvement of (25-17)/17 = 47% over base RAMpage.

Notice that the performance of RAMpage with context switching is worse than base RAMpage for a CPU speed of 200MHz. This is due to the inability of the improved CPU utilization to compensate for the context switching overhead at this low CPU speed. As CPU speed increases beyond 1000MHz, the ability to process more references in the page fetch delay shows in the increasing rate of growth of the context switching curve relative to the non context switching one. This result demonstrates the value of context switching in providing a scalable solution as the CPU-DRAM speed gap widens.

Chapter 8 – Conclusions

8.1 Introduction

Semiconductor technology trends have brought about the situation where the current trend of continued performance improvement cannot be sustained indefinitely. The growing CPU-DRAM performance disparity is a direct and as yet unavoidable result of the tradeoff between speed and capacity in semiconductor components. Consequently, the use of SRAM is crucial to delaying the onset of the memory performance barrier, and will be even more so as CPUs continue to get faster relative to DRAM. However, current methods of managing SRAM caches may prove inadequate as the memory wall approaches, indicating new techniques are needed.

The RAMpage hierarchy offers the potential of improved performance that scales with the CPU-DRAM speed gap and thus is an attractive alternative design approach. Moreover, through managing caches in software by moving main memory from DRAM to the lowest level of SRAM, DRAM can now be treated as a peripheral. This allows context switching – a technique traditionally employed when accessing slow magnetic disk drives – to be employed in SRAM, resulting in further performance improvement. By performing context switches on page faults, the high miss costs of the miss-rate-optimal page size of RAMpage can be masked.

The goal of this work has been to determine the feasibility of context switching on misses to DRAM as an alternative approach to the problem of lagging memory speed. This objective has been made possible with the RAMpage hierarchy.

The next section summarizes the results and their significance. A brief discussion of the limitations and suggestions for future work concludes the chapter.

9.2 Summary of Results

Trace-driven simulation of the RAMpage and standard hierarchies has been used to evaluate the effect of context switching on SRAM misses. The RAMpage hierarchy, due to its ability to pin operating system code in SRAM, has been shown to reduce context switching time. This is significant for the following reasons:

- Since a context switch on a miss to DRAM does not have as large a time interval to perform the switch as a miss to disk does, its success is very sensitive to the cost of the switch.
- Avoiding misses while switching means the miss handler does not have to be concerned about the complications of recursion when context switching on a miss causes a miss itself.

In addition, because misses to DRAM do not occur when context switching code is pinned in SRAM, the cost of a context switch does not increase as the page size increases. Thus, the miss rate optimal page size of 4K can be exploited without incurring an increase in the context switch cost, as occurs under the standard hierarchy.

By context switching on RAMpage page faults, the relative improvement of the RAMpage hierarchy has been increased from a previous 17% to 25% for the largest simulated CPU-DRAM speed gap, using the performance optimal page size in each case. Moreover, the rate of performance improvement of RAMpage with context switching is greater than that of base RAMpage as the CPU-DRAM speed gap grows. This means the addition of context switching has improved the scalability of RAMpage.

Thus, the objectives of this research have been met, namely to investigate the effect of context switching on misses to DRAM and evaluate the scalability of this approach with the increasing CPU-DRAM speed gap. Context switching under RAMpage has been demonstrated to be a technique worthy of consideration as an alternative approach to dealing with the memory wall problem.

9.3 Limitations and Future Work

A number of simplifications have been necessary to keep this research within reasonable limits. Therefore there is ample scope for a more detailed investigation. Several issues future work may consider are outlined below.

More detail should be considered when designing the simulation. The context switch simulated here was independent of any particular architecture. Modeling a context switch closely on a real architecture would lend more credibility to the results. Using longer traces would be necessary to model real system behaviour more accurately. A better understanding of program behaviour in SRAM would help. For example, knowledge of the cache footprint of each simulated application would assist in better understanding context switching behaviour.

There are many options available to the simulation designer regarding simulation parameters. For example, memory system parameters could be extended beyond those employed here, such as investigating the effect of a 64B block size for the standard hierarchy and an 8K RAMpage page size. Neither SRAM size nor the number of processes (degree of multiprogramming) were varied here. It may be worthwhile investigating these parameters, particularly when process cache footprints are considered.

A better approach would be to use an operating system simulator that could thoroughly model the sequence of events in a context switch. Several such simulators are available, for example *Nachos* [Christopher *et al.* 1998] and *SimOS* [Rosenblum *et al.* 1995]. Given enough resources, it may even be possible to implement an experimental context switching RAMpage system on a real operating system for which the source code is freely available, such as *Linux*.

References

[Agarwal *et al.* 1988] A Agarwal, J Hennessy and M Horowitz. Cache Performance of Operating System and Multiprogramming Workloads, *ACM Transactions on Computer Systems*, vol. 6 no. 4, November 1988, pp 393-431.

[Agarwal and Pudar 1993] A Agarwal and S Pudar. Column Associative caches: A Technique for Reducing the Miss Rate of Direct Mapped Caches. *Proc 20th Ann. Int. Symp. on Computer Arch.*, May 1993, pp 179-1990.

[Anderson *et al.* 1991] T Anderson, B Bershad, E Lazowska and H Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism, *Proc. 13th Symp. on Operating Systems Principles*, 1991, pp 95-109.

[Bacon *et al.* 1994] D Bacon, S Graham and O Sharp. Compiler Transformations for high performance computing, *ACM Computing Surveys*, vol. 26 no. 4, December 1994, pp 345-420.

[Belaneh and Kaeli 1996] S Belaneh and D Kaeli. A Discussion of Non-blocking/Lockup-free caches. *Computer Architecture News*, vol. 24 no. 3, June 1996, pp 18-25.

[Birrell 1989] A Birrell. *An Introduction to Programming with Threads*, Research Report 35, Digital Equipment Corporation, Systems Research Center, Palo Alto, CA, January 1989.
`http://www.research.digital.com/SRC/staff/birrell/bib.html`

[Boland and Dollas 1994] K Boland and A Dollas. Predicting and precluding problems with memory latency. *IEEE Micro*, vol. 14 no. 4, August 1994, pp 59-67.

[Borg *et al.* 1990] A Borg, R Kessler and D Wall. Generation and Analysis of very long address traces, *Proc. 17th Ann. Int. Symp. on Computer Arch.*, Seattle, WA, May 1990, pp 270-279.

[Chaudhry and Li 1994] G Chaudhry and X Li. A Case for the Multithreaded Processor Architecture. *Computer Architecture News*, vol 22 no. 4, September 1994, pp 55-59.

[Chen and Baer 1992] T Chen and J Baer. Reducing Memory Latency via Non-blocking and Prefetching Caches. *Proc. 5th Int. Conf. On Architectural Support for Programming Languages and Operating Systems*. September 1992, pp 51-61.

[Chen *et al.* 1994] J Chen, D Wall, and A Borg. *Software Methods for System Address Tracing: Implementation and Validation*, WRL Research Report 94/6, Western Research Laboratory, CA, September 1994.

[Cheriton *et al.* 1993] D Cheriton, H Goosen, H Holbrook and P Machanick. Restructuring a Parallel Simulation to Improve Cache Behaviour in a Shared Memory Multiprocessor: The Value of Distributed Synchronization. *Proc. of the 7th Workshop on Parallel and Distributed Simulation*, San Diego, May 1993, pp 159-162.

[Christopher *et al.* 1998] W Christopher, S Procter and T Anderson. *The Nachos Instructional Operating System*, Technical Report, University of California at Berkely. `http://cs-tr.cs.berkeley.edu/TR/UCB:CSD-93-739`

[Clark 1983] D Clark. Cache performance in the VAX-11/780, *ACM Transactions on Computer Systems*, vol. 1 no. 1, February 1983, pp 24-37.

[Coffman and Denning 1973] E Coffman and P Denning. *Operating Systems Theory*, Prentice Hall, Englewood Cliffs, NJ, 1973.

[Crisp 1997] R Crisp. Direct Rambus technology: The new main memory standard. *IEEE Micro*, vol. 17 no. 6, November 1997, pp 18-28.

[Crowley 1997] C Crowley. *Operating Systems: A Design-Oriented Approach*, Irwin Publishing. Chicago, 1997.

[Denning 1968] P Denning. The Working Set Model for Program Behaviour, *Comm. ACM*, vol. 11 no. 5, May 1968, pp 323-333.

[Digital 1998] Digital Semiconductor. Alpha 21164pc microprocessor (366-600MHz). `http://www.digital.com/semiconducor/alpha/dsc-21164.html`

[Geppert 1996] L Geppert. Semiconductor Lithography for the Next Millennium, *IEEE Spectrum*, vol. 33 no. 4, April 1996, pp 33-38.

[Gjessing *et al.* 1992] S Gjessing, D Gustavson, D James, G Stone and H Wiggers. A RAM Link for High Speed, *IEEE Spectrum*, vol. 29 no. 10, October 1992, pp 52-53.

[Halstead and Fujita 1988] R Halstead and T Fujita. Masa: A Multithreaded Processor Architecture for Parallel Symbolic Computing, *Proc. 15th Ann. Int. Symp. on Computer Arch.*, May 1988, Honolulu, pp 443-451.

[Hennessy and Jouppi 1991] J Hennessy and N Jouppi. Computer Technology and Architecture: An Evolving Interaction, *IEEE Computer*, vol. 24 no. 9, September 1991, pp 18-29.

[Hennessy and Patterson 1996] J Hennessy and D Patterson. *Computer Architecture: A Quantitative Approach* (2nd edition), Morgan Kaufmann, San Mateo, CA, 1996.

[Hill 1988] M Hill. A Case for Direct-Mapped Caches, *IEEE Computer*, vol. 21 no. 12, December 1988, pp 25-40.

[Hsu and Smith 1993] W Hsu and J Smith. Performance of Cached DRAM Organizations in Vector Supercomputers, *Proc. 20th Ann. Int. Symp. on Computer Arch.*, San Diego, May 1993, pp 327-336.

[IBM 1997] IBM Microelectronics. *Synchronous DRAM: The DRAM of the Future.* `http://www.chips.ibm.com/products/memory/sdramart/sdramart.html`

[Johnson 1995] E Johnson. Graffiti on the Memory Wall, *Computer Architecture News*, vol. 23 no. 4, September 1995, pp 7-8.

[Johnson and Ha 1994] E Johnson and J Ha. PDATS: Lossless Address Trace Compression for Reducing File Size and Access time. *Proc. of the IEEE Int. Phoenix Conf. on Computers and Communications*, 1994, pp 213-219.

[Jouppi 1990] N Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. *Proc. 17th Ann. Int. Symp. on Computer Arch.*, May 1990, pp 364-373.

[Ki and Knowles 1997] A Ki and A Knowles. Adaptive Data Prefetching Using Cache Information, *Proc. 1997 Int. Conf. On Supercomputing*, Vienna, 1997, pp 204-212.

[Koldinger *et al.* 1991] E Koldinger, S Eggers and H Levy. On the Validity of Trace-driven Simulation for Multiprocessors. *Proc. 18th Ann. Int. Symp. on Computer Arch.*, Toronto, May 1991, pp 244-253.

[Lam *et al.* 1991] M Lam, E Rothberg and M Wolf. The Cache Performance and Optimization of Blocked Algorithms, *Proc. 4th Int. Conf. On Architectural Support for Programming Languages and Operating Systems,* Santa Clara, CA, 1991, pp 63-74.

[Louri and Sung 1994] A Louri and H Sung. 3D Optical Interconnects for High-Speed Interchip and Interboard Communications, *Computer*, vol. 27 no. 10, October 1994, pp 27-37.

[Machanick 1996] P Machanick. The Case for SRAM Main Memory, *Computer Architecture News*, vol. 24 no. 5, December 1996, pp 23-30.

[Machanick *et al.* 1998] P Machanick, P Salverda and L Pompe. Hardware-Software Trade-Offs in a Direct Rambus Implementation of the RAMpage Memory Hierarchy, *Proc. 8th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, October 1998, pp 105-114.

[Martenosi *et al.* 1995] M Martenosi, A Gupta and T Anderson. Tuning Memory Performance of Sequential and Parallel Programs. *Computer*, vol. 28 no. 4, April 1995, pp 32-40.

[McVey and Staelin 1996] L McVey and C Staelin. lbench: Portable tools for performance analysis. *Unix*, Jan 1996.

[Mogul and Borg 1991] J Mogul and A Borg. The Effect of Context Switches on Cache Performance, *Proc. 4th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, CA, 1991, pp 75-84.

[Nagle *et al.* 1993] D Nagle, R Uhlig, T Stanley, S Sechrest and R Brown. Design Tradeoffs for Software Managed TLBs, *Proc. 20th Ann. Int. Symp. on Computer Arch.*, San Diego, May 1993, pp 27-38.

[Przybylski *et al.* 1988] S Pryzbylski, M Horowitz and J Hennessy. Design Tradeoffs in Cache Design, *Proc. 15th Ann. Int. Symp. on Computer Arch.*, Honolulu, May 1988, pp 290-298.

[Rosenblum *et al.* 1995] M Rosenblum, S Herrod, E Witchel and A Gupta. Complete Computer System Simulation: The SimOS Approach, *IEEE Parallel and Distributed Technology*, vol. 3 no. 4, 1995, pp 34-43.

[Salverda 1997] P Salverda. *An SRAM Main Memory Model*, MSc research report, University of the Witwatersrand, Johannesburg, September 1997.

[Silberschatz *et al.* 1994] A Silberschatz, J Peterson and P Galvin. *Operating System Concepts*, Addison Wesley, Reading, MA, 1994.

[Smith 1982] A Smith. Cache Memories, *Computing Surveys*, vol. 14 no. 3, September 1982, pp 473-530.

[SPEC 1998] `http://www.specbench.org/spec`

[Stallings 1993] W Stallings. *Computer Organization and Architecture. Principles of Structure and Function*, (3rd edition), Prentice Hall, Englewood Cliffs, NJ, 1993.

[Tanenbaum 1987] A Tanenbaum. *Operating Systems Design and Implementation.* Prentice Hall, Englewood Cliffs, NJ, 1987.

[Tanenbaum 1995] A Tanenbaum. *Distributed Operating Systems*, Prentice Hall, Englewood Cliffs, NJ, 1995.

[Thiebaut and Stone 1987] D Thiebaut and H Stone. Footprints in the Cache, *ACM Transactions on Computer Systems*, vol. 5 no. 4, November 1987, pp 305-329.

[Uhlig *et al.* 1995] R Uhlig, D Nagle, J Emer, T Mudge and S Sechrest. Instruction Fetching: Coping with Code Bloat, *Computer Architecture News*, vol. 23 no. 2, May 1995, pp 345-356.

[Veenstra and Fowler 1993] J Veenstra and R Fowler. Mint Tutorial and User Manual, *Technical Report 452*, Computer Science Department, University of Rochester, NY, June 1993 (revised August 1994).
`ftp://ftp.cs.rochester.edu/pub/packages/mint/mint-2.8.tar.Z`

[Weber and Gupta 1989] W Weber and A Gupta. Exploring the Benefits of Multiple Hardware Contexts in a Multiprocessor Architecture: Preliminary Results, *Proc. 16th Ann. Int. Symp. on Computer Arch.*, Jerusalem, May 1989, pp 290-298.

[Weiker 1997] R Weiker. On the Use of SPEC Benchmarks in Computer Architecture Research, *Computer Architecture News*, vol. 25 no. 1, March 1997, pp 19-22.

[Wilkes 1995] M Wilkes. The Memory Wall and the CMOS End Point, *Computer Architecture News*, vol. 23, no. 4, September 1995, pp 4-6.

[Wilson and Olukotun 1997] K Wilson and K Olukotun. Designing High Bandwidth On-Chip Caches, *Proc. 24th Ann. Int. Symp. on Computer Arch.*, Denver, CO, May 1997, pp 121-132.

[Wulf and McKee 1995] W Wulf and S McKee. Hitting the Memory Wall: Implications of the Obvious, *Computer Architecture News*, vol. 23 no. 1, March 1995, pp 20-24.