Technical Report

UCAM-CL-TR-619
ISSN 1476-2986

Number 619

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Operating system support for simultaneous multithreaded processors

James R. Bulpin

February 2005

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted September 2004 by the author for the degree of Doctor of Philosophy to the University of Cambridge, King's College.

ISSN 1476-2986

Summary

Simultaneous multithreaded (SMT) processors are able to execute multiple application threads in parallel in order to improve the utilisation of the processor's execution resources. The improved utilisation provides a higher processor-wide throughput at the expense of the performance of each individual thread.

Simultaneous multithreading has recently been incorporated into the Intel Pentium 4 processor family as "Hyper-Threading". While there is already basic support for it in popular operating systems, that support does not take advantage of any knowledge about the characteristics of SMT, and therefore does not fully exploit the processor.

SMT presents a number of challenges to operating system designers. The threads' dynamic sharing of processor resources means that there are complex performance interactions between threads. These interactions are often unknown, poorly understood, or hard to avoid. As a result such interactions tend to be ignored leading to a lower processor throughput.

In this dissertation I start by describing simultaneous multithreading and the hardware implementations of it. I discuss areas of operating system support that are either necessary or desirable.

I present a detailed study of a real SMT processor, the Intel Hyper-Threaded Pentium 4, and describe the performance interactions between threads. I analyse the results using information from the processor's performance monitoring hardware.

Building on the understanding of the processor's operation gained from the analysis, I present a design for an operating system process scheduler that takes into account the characteristics of the processor and the workloads in order to improve the system-wide throughput. I evaluate designs exploiting various levels of processor-specific knowledge.

I finish by discussing alternative ways to exploit SMT processors. These include the partitioning onto separate simultaneous threads of applications and hardware interrupt handling. I present preliminary experiments to evaluate the effectiveness of this technique.

Acknowledgements

I would like to thank my supervisor, Ian Pratt, for his advice and guidance. I am also grateful to my colleagues in the Systems Research Group and local research labs for their friendship, support and many interesting discussions. In particular thanks are due to Jon Crowcroft, Tim Deegan, Keir Fraser, Steve Hand, James Hall, Tim Harris, Richard Mortier, Rolf Neugebauer, Dave Stewart and Andrew Warfield.

I would like to thank Keir Fraser, Ian Pratt and Tim Harris for proof reading earlier copies of this dissertation. Any errors that remain are my own.

I spent a summer as an intern at Microsoft Research Cambridge. I would like to thank my mentor, Rebecca Isaacs, and the others involved with the "Magpie" project for an interesting and enjoyable break from my own research.

Thanks are due to Intel Research Cambridge for their kind donation of a "Prescott" based computer which I used for some of the experiments in Chapter 3.

My funding was from a CASE award from Marconi Corporation plc. and the Engineering and Physical Sciences Research Council. I would also like to thank my current employers for their patience and support while I finished writing this dissertation.

Table of contents

Glossary **11**

1 Introduction **13**
1.1 Motivation . . . 13
1.2 Terminology . . . 14
1.3 Contribution . . . 15
1.4 Outline . . . 15

2 Background **17**
2.1 Simultaneous Multithreading . . . 17
2.1.1 Multithreading . . . 19
2.1.2 Research Background of SMT . . . 20
2.2 Commercial SMT Processors . . . 21
2.2.1 Alpha 21464 . . . 21
2.2.2 Intel Hyper-Threading . . . 21
2.2.3 IBM Power5 . . . 24
2.3 Operating System Support . . . 26
2.3.1 An Extra Level of Hierarchy . . . 26
2.3.2 Simultaneous Execution . . . 27
2.3.3 Cache Considerations . . . 31
2.3.4 Energy Considerations . . . 34
2.4 Summary . . . 35

3 Measuring SMT **37**
3.1 Related Work . . . 37
3.1.1 Simulated Systems . . . 37
3.1.2 Real Hardware . . . 38
3.2 Experimental Configurations . . . 39
3.2.1 Performance Counters . . . 39
3.2.2 Test Platforms . . . 40
3.2.3 Workloads . . . 41
3.2.4 Performance Metrics . . . 43
3.3 Thread Interactions . . . 44
3.3.1 Experimental Method . . . 45
3.3.2 Results . . . 46
3.3.3 Desktop Applications . . . 56

3.3.4 Summary 58
3.4 Phases of Execution 59
3.4.1 Performance Counter Correlation 66
3.5 Asymmetry 67
3.6 Summary 69

4 A Process Scheduler for SMT Processors 71
4.1 The Context 71
4.1.1 Problems with Traditional Schedulers 72
4.1.2 Scheduling for Multithreading and Multiprocessing Systems 72
4.2 Related Work 73
4.2.1 Hardware Support 73
4.2.2 SMT-Aware Scheduling 74
4.3 Practical SMT-Aware Scheduling 75
4.3.1 Design Space 76
4.4 Implementation 76
4.4.1 The Linux Scheduler 77
4.4.2 Extensible Scheduler Modifications 77
4.4.3 Performance Estimation 78
4.4.4 SMT-Aware Schedulers 79
4.5 Evaluation 82
4.5.1 Method 82
4.5.2 Throughput 83
4.5.3 Fairness 87
4.6 Applicability to Other Operating Systems 92
4.7 Applicability to Other Processors 93
4.8 Summary 94

5 Alternatives to Multiprogramming 95
5.1 A Multithreaded Processor as a Single Resource 95
5.1.1 Program Parallelisation 96
5.2 Threads for Speculation and Prefetching 96
5.2.1 Data Speculation 96
5.2.2 Pre-execution 98
5.2.3 Multiple Path Execution 100
5.3 Threads for Management and Monitoring 100
5.3.1 Mini-Threads 101
5.3.2 Subordinate Threads 101
5.4 Fault Tolerance 104
5.5 Operating System Functions 105
5.5.1 Exception Handling 105
5.5.2 Privilege Level Partitioning 106
5.5.3 Interrupt Handling Partitioning on Linux 106
5.6 Summary 113

6 Conclusions **115**
6.1 Summary 115
6.2 Further Research 116

References **119**

A Monochrome Figures **129**

Glossary

ALU	Arithmetic-Logic Unit
CMP	Chip Multiprocessor/Multiprocessing
DEC	Digital Equipment Corporation
D-TLB	Data Translation Lookaside Buffer
FP	Floating Point
HMT	Hardware Multithreading
IA32	Intel Architecture 32 bit
IBM	International Business Machines
ILP	Instruction Level Parallelism
I/O	Input/Output
IPC	Instructions per Cycle
IQR	Inter-Quartile Range (of a distribution)
IRQ	Interrupt Request
ISA	Instruction Set Architecture
ISR	Interrupt Service Routine
I-TLB	Instruction Translation Lookaside Buffer
HP	Hewlett-Packard (bought Compaq who bought DEC)
HT	Hyper-Threading
L1/L2	Level1/Level2 cache
LKM	Linux/Loadable Kernel Module
MP	Multiprocessor/Multiprocessing
MSR	Model Specific Register (Intel)
NUMA	Non-uniform Memory Architecture
OoO	Out of Order (superscalar processor)
OS	Operating System
P4	Intel Pentium 4
PID	Process Identifier
RISC	Reduced Instruction Set Computer
ROM	Read-only Memory
SMP	Symmetric Multiprocessing
SMT	Simultaneous Multithreaded/Multithreading
TC	Trace-cache
TLB	Translation Lookaside Buffer
TLP	Thread Level Parallelism
TLS	Thread Level Speculation

UP	Uniprocessor/Uniprocessing
VM	Virtual Memory

Chapter 1

Introduction

This dissertation is concerned with the interaction of software, particularly operating systems, and simultaneous multithreaded (SMT) processors. I measure how a selection of workloads perform on SMT processors and show how the scheduling of processes onto SMT threads can cause the performance to change. I demonstrate how improved knowledge of the characteristics of the processor can improve the way the operating system schedules tasks to run.

In this chapter I outline the ideas behind SMT processors and the difficulties they create. I state the contributions that are described in this dissertation. The chapter finishes with a brief summary of later chapters.

1.1 Motivation

The difference in the rate of growth between processor speed and memory latency has led to an increasing delay (relative to processor cycles) to access memory. This is a well known problem and there are a number of techniques in common use to work around it. While caches are very effective, the cost of a level 1 cache miss causing an access to a lower level cache can cost many processor cycles. To try to minimise the effect of a cache miss on program execution, *dynamic issue superscalar* (or *out-of-order*, OoO) processors attempt to execute other instructions not dependent on the cache access while waiting for it to complete. Moreover, these processors have multiple execution units enabling multiple independent instructions to be executed in parallel. The efficacy of this approach depends on the amount of *instruction level parallelism* (ILP) available in the code being executed. If code exhibits low ILP then there are few opportunities for parallel execution and for finding sufficient work to perform while waiting for a cache miss.

Simultaneous multithreading (SMT) is an extension of dynamic issue superscalar processing. The aim is to increase the pool of instructions which the processor can choose to execute. An SMT processor will fetch instructions from more than one thread at a time. Since there will be no data dependency between instructions from different threads (ignoring higher-level, infrequent occurrences with shared data), the number of independent instructions the processor can choose between will generally be greater than in the non-SMT processor. An added benefit is the greater diversity of instructions making it more likely that there will be executable instructions available for an idle functional unit. Completed instructions are extracted separately for each thread so that architectural state can be updated in original program order, a process called retirement. The execution core of the processor need not be aware that instructions come from different threads as it is only concerned with executing instructions whose operands are ready.

SMT aims to provide a system speedup at the expense of per-thread performance. A thread running on an SMT processor will generally run slower than it would have done on a non-SMT processor because it has to share processor resources (including the caches). However, the combined throughput of simultaneously running threads should be higher than that of a single thread on a non-SMT processor due to the higher core utilisation. Therefore the time to execute a batch of programs using SMT should be less than running them sequentially on a non-SMT processor.

SMT can cause a number of problem for software:

- SMT threads have to share processor resources so will proceed at a speed that depends on the level and mix of demands placed on those resources by all the simultaneously running threads. This makes predicting and measuring processor allocation (such as in a real-time system) hard.
- The processor-wide performance depends on how well the threads share the processor resources. Typically it would be expected that a homogeneous mixture of threads would have a poor performance and a heterogeneous mixture would do better. In order to get the best from the processor it is necessary to be careful when selecting threads to run simultaneously.
- The processor cache(s) are shared by all threads. Sets of threads with large data footprints risk thrashing the cache and reducing their simultaneous throughput to less than their non-SMT sequential throughput.

SMT processors are becoming common. The most widespread implementation is Intel's "Hyper-Threading" available on all variants of the Pentium 4 processor. IBM's latest addition to the "Power" family, the Power5, is an SMT processor. The increasing ubiquity of SMT hardware has not been matched by changes in software. Most operating systems exploit the backwards compatible interfaces provided by Hyper-Threading creating a "virtual multiprocessor" system. Current operating system support is limited to the minimum needed to be able to use SMT processors without suffering major performance problems.

Intel's implementation of SMT, "Hyper-Threading", presents the two logical processors in a physical processor package as separate, independent processors which, although easing the job of the operating system developer, can have unwelcome effects on performance.

1.2 Terminology

Current SMT implementations all present the simultaneous threads as **logical processors**. A logical processor appears to the applications and, to a certain degree, to the operating system much like a physical processor in a multiprocessor system. In this dissertation I use the terms **logical processor**, **thread** and **Hyper-Thread** (when specifically discussing Intel Hyper-Threading) interchangeably.

I use the terms **physical processor** or **package** to describe the physical processor package which presents itself as a set of logical processors.

Each logical processor can execute a thread in its own address space in the same way as multiple processors would (unlike some specialist multithreaded hardware where all threads must share an address space). The running **threads** are therefore **heavyweight threads**, or **processes**. Note that the term **thread** can be used to describe both an operating system thread and a hardware simultaneous thread; where the intended meaning is not clear from the context a more descriptive term is used.

1.3 Contribution

The thesis of this dissertation is that simultaneous multithreaded processors can be used more effectively with an operating system that is aware of their characteristics.

As there has only recently been an implementation of an SMT processor available, there have been very few studies of the practical issues in the use of SMT. I present a series of measurements of applications running on Intel's Hyper-Threaded processor, looking at the mutual effects of concurrent execution. The main contributions from this work are:

- A detailed measurement study of a real simultaneous multithreaded processor.
- An examination of the fairness and mutual performance degradation of simultaneously executing threads as well as the total speedup obtained.
- The use of processor hardware performance counters to examine the causes of the performance interactions observed and to estimate performance.

I describe how operating system functions, particularly the process scheduler, can be made aware of the characteristics of SMT processors in order to improve system performance. I propose, implement and evaluate a number of SMT-aware scheduling algorithms and measure their performance. This work contributes the following:

- A technique to estimate thread performance on an SMT processor using data from processor hardware performance counters.
- Process scheduling algorithms that are able to improve throughout and/or fairness for sets of threads executing on an SMT processor.
- Techniques to utilise SMT-specific knowledge while still respecting existing scheduler functions such as priorities and starvation avoidance.

1.4 Outline

The remainder of this dissertation is structured as follows.

In Chapter 2 I describe simultaneous multithreading in more detail and describe current and proposed implementations, including Intel Hyper-Threading, the first commercially available implementation of SMT and the platform used for the experiments described in this dissertation. I introduce and discuss areas of operating system support important for SMT processors.

Chapter 3 describes experimental work performed to assess the realised performance of the Intel Hyper-Threaded processor. The results are presented and discussed.

In Chapter 4 I describe a new scheduler suitable for use with SMT processors that is able to monitor the performance of running threads and improve the system throughput by making intelligent decisions about what to run on the same processor. I measure the performance of the scheduler and compare it to traditional schedulers.

In Chapter 5 I discuss other ways in which SMT processors may be used beside the quasi-SMP model discussed in the earlier chapters. I discuss the advantages and limitations of allocating a multithreaded processor as a single resource. I describe ways in which threads could be used non-symmetrically.

Finally Chapter 6 summarises and concludes the dissertation and highlights areas of further research.

Related work is described in the relevant chapters.

Chapter 2

Background

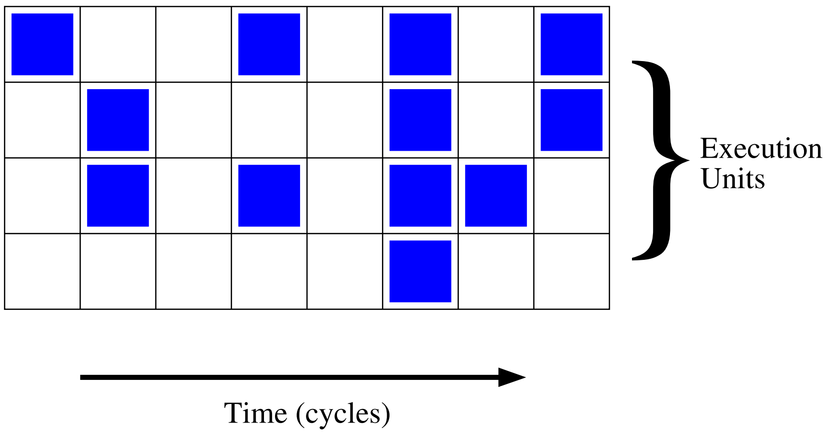
In this chapter I elaborate on the description of simultaneous multithreading in the previous chapter and describe research work on SMT hardware design. I describe commercial implementations of, and proposals for, SMT processors including Intel's Pentium 4 with Hyper-Threading and IBM's Power5. I go on to introduce and discuss those areas of operating system support that are important for SMT.

2.1 Simultaneous Multithreading

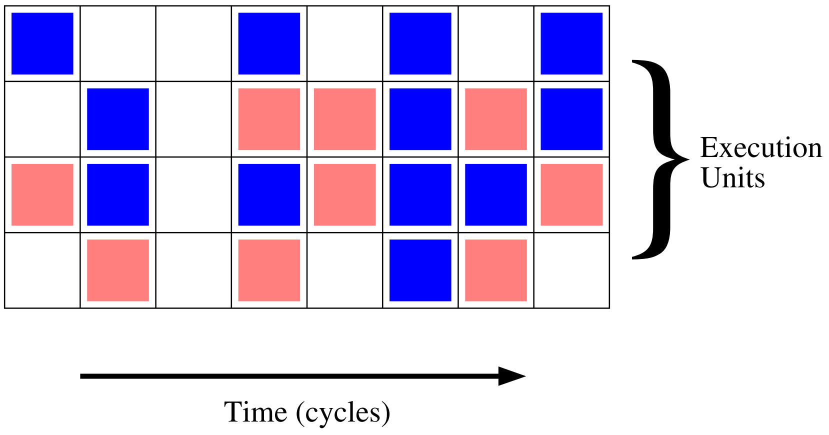
In this section I describe the basic architecture of a simultaneous multithreaded processor based on a dynamic issue superscalar processor.

Modern *dynamic issue superscalar* (or *out-of-order*, OoO) processors such as the Intel Pentium 4, AMD K7 and modern Sun SPARC and IBM Power processors are able to execute instructions in a *data-flow* manner, where an instruction is a candidate for execution once all of its operand register values are available. This technique allows for parallel execution of non-dependent instructions. Superscalar processors have multiple *execution units* such as arithmetic logic units (ALUs) and memory access units. The out-of-order nature means that non-dependent instructions beyond a memory load can be executed while that memory load completes. It is this mechanism that makes OoO processors more tolerant to level 1 cache misses and structural and data hazards, than in-order processors. Figure 2.1(a) shows a simplified example of the operation of a 4-execution unit OoO processor. Each column represents a successive clock cycle; a blue square denotes an instruction occupying the execution unit for that cycle. There are completely empty columns because an earlier instruction has not completed (e.g. a memory load). Gaps elsewhere tend to be caused by insufficient parallelism in the code.

The processor keeps a pool, or *window*, of instructions from which it finds those instructions with satisfied dependencies for execution. The pool is filled by the fetch and decode stages of the pipeline and instructions are removed (*retired*) from the pool in the original program order once they have been executed. To be able to execute out-of-order and ahead into the instruction stream requires that the processor can execute beyond branches. A *branch predictor* is used to decide on which direction to follow and execution proceeds *speculatively* from there with the predictions being checked once the outcome is known. Instructions are only retired when it is known they were on a correctly predicted path. Incorrectly speculated instructions are discarded



(a) OoO Superscalar



(b) SMT

Figure 2.1: An illustration of the operation of out-of-order superscalar, and SMT processors. Squares of each colour represent instructions from different threads occupying execution units.

and their results are not committed to the architectural state. The window can extend deep into the instruction stream of the program so will contain multiple, independent uses of the same numbered register. In order to disambiguate these different uses, the registers specified in the fetched instructions are renamed to be (temporally locally) unique [Tomasulo67, Hennessy03].

The modifications required to make this architecture simultaneously multithreaded are fairly minor: because the processor execution core is only concerned with instructions and data dependencies between them it is possible to feed instructions from more than one thread into it. A dynamic-issue SMT processor will fetch instructions from multiple threads and rename the registers in each such that at any point in time the threads are using non-overlapped sets of physical processor registers. By doing this the core sees more non-dependent instructions which can increase the amount of parallelism that can be exploited. The fetching, decoding and register renaming of instructions can be implemented in per-thread hardware or can use the same hardware in an interleaved manner. The processor needs to keep architectural state, including the program

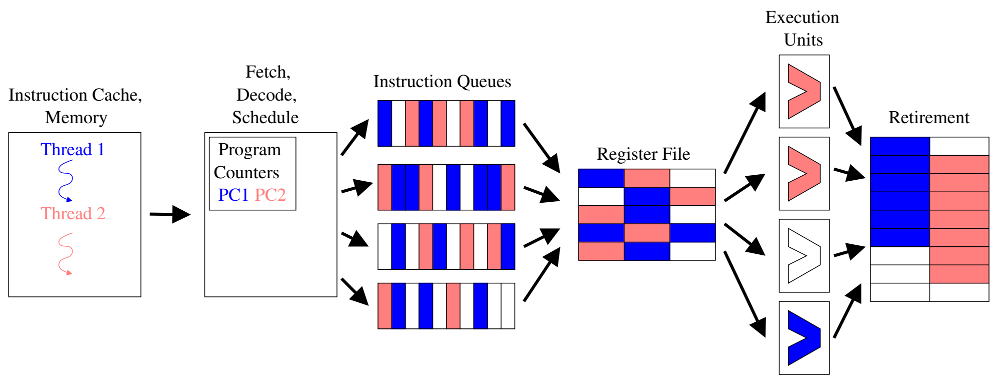


Figure 2.2: A simplified 2-thread dynamic SMT architecture.

counter, for each thread. This state is commonly referred to as a *context*. Figure 2.2 shows a simplified architecture for a dynamic-issue SMT processor.

Figure 2.1(b) shows how the modified OoO architecture can execute instructions from two threads simultaneously. Note that the instructions from the second thread are filling gaps in the first thread's execution caused by both pipeline stalls (due to long latency instructions) and lack of instruction parallelism.

The execution resources of the processor are shared dynamically with a variable number from each thread (possibly including none) being issued in each cycle.

The processor caches, including the branch prediction cache and memory management unit translation lookaside buffers (TLBs), are shared by all threads. The method of sharing could be a purely dynamic scheme where the identity of the related thread is irrelevant to the cache location or eviction policy. The sharing of the data caches would be a typical example of this scheme. Alternatively a cache could be statically partitioned between threads. This scheme limits the amount of statistical multiplexing of demands to available resource but stops any one thread hogging the resource. A branch prediction cache or TLB could use such a scheme.

2.1.1 Multithreading

Simultaneous multithreading is just one of a number of different kinds of hardware multithreading. Traditionally, hardware multithreaded (HMT) processors have been able to execute multiple threads "concurrently" from the operating system and user point of view. HMT processors generally alternate between threads per cycle (*interleaved multithreading* - IMT) or context switch between threads on some event, such as a cache miss (*blocked multithreading* - BMT). SMT processors are able to execute instructions from multiple threads within each cycle and are able to share the execution resources dynamically. There has also been a design for a mulithreaded dataflow processor [Moore96].

There are a number of common characteristics between the different forms of multithreading, the sharing of the cache hierarchy being the most notable. SMT and IMT will place similar demands on the caches as they both share execution resource in a fine-grained manner so the stream of memory references will contain references from all threads. BMT architectures that context switch on (L2) cache misses will present the caches with a reference stream with a more coarse-grained thread alternation. The different techniques vary in their predictability: IMT's alternating execution is fairly predictable but BMT's event-induced context switching is less so. The dynamic nature of SMT's sharing of execution resource makes it harder to predict each thread's throughput making it more like BMT in this respect. A full survey of the issues of non-SMT multithreaded processors is beyond the scope of this dissertation, however ideas from this field are discussed where they are relevant to SMT. A detailed survey of hardware multithreaded processors is presented by Ungerer *et al* [Ungerer03].

Existent forms of non-SMT hardware multithreading tend to be confined to the supercomputer/mainframe field (such as the Cray/Tera MTA [Alverson90]) or to specialised processors such as network processors. SMT is the first form of multithreading to become common on commodity desktop and server computers. Since desktop computers usually run different workloads to supercomputers and network processors, they present their own set of issues. Network processing is usually easy to parallelise with threads working on different packets or on different parts of a processing pipeline. Additionally, code is generally written specifically for a particular network processor, with knowledge of its characteristics. Programmers targeting supercomputers put a lot of effort into making their programs highly parallel. On the desktop, the diversity of hardware and uses means that the workloads are also diverse with a mix of single-threaded and multi-threaded applications, the latter not generally being written with the constraints of SMT in mind. Multithreaded desktop workloads often possess a single dominant thread [Lee98].

2.1.2 Research Background of SMT

The history of SMT research can be broken into two distinct areas: architectures based on dynamic superscalars and those based on other architectures such as VLIW. The earliest work was in the non-superscalar field; the "MARS-M" system [Dorozhevets92] and the "Matsushita Media Research Laboratory processor" [Hirata92] are examples. The move towards superscalar based designs started with the "Multistreamed superscalar processor" [Serrano94, Yamamoto95].

The model of SMT widely used in recent years [Tullsen98] is based on work carried out at the University of Washington. The Washington design was originally based on a static superscalar architecture [Tullsen95]. The design was evaluated against a single-threaded dynamic superscalar with the same issue width (number of instructions that can be executed per cycle) and found to outperform the dynamic superscalar (which itself outperformed an IMT-style design).

The Washington work moved towards a dynamic, out-of-order, superscalar design when it was found that such an architecture could be made to be multithreaded with only a small cost [Tullsen96b, Eggers97, Lo97b]. The design started with a high-performance out-of-order superscalar design similar in spirit to the MIPS R10000. To support multiple threads multiple program counters were added; these were fetched from on an interleaved basis. Structures that needed to be per-thread, such as retirement and the trap mechanism, were duplicated. Thread tagging support was added to shared data structures, such as the branch target buffer, where the ownership of an entry was not made clear by the renaming of registers. The design supported eight threads which placed a large demand on physical registers therefore they opted to increase the size of the register file and pipeline its access.

At around the same time Loikkanen and Bagherzadeh presented results of simulation studies of a fine-grain multithreading processor [Loikkanen96]. Whilst not described as SMT their design had many of the characteristics of subsequent SMT processors including a shared (but partitioned) register file and dynamically shared reorder buffer and execution units.

2.2 Commercial SMT Processors

In this section I describe current and proposed commercial realisations of simultaneous multi-threaded processors. I concentrate on Intel Hyper-Threading as this is the most widely available and forms the basis for the work in the subsequent chapters of this dissertation. I describe the IBM Power5 with a focus on its microarchitectural differences from Hyper-Threading where these affect the interaction with software and the operating system. For completeness I briefly describe the proposed, but never built, Alpha 21464.

2.2.1 Alpha 21464

The University of Washington's SMT work was developed commercially in collaboration with the Alpha processor group at Digital Equipment Corporation. This union ultimately led Compaq (who had bought DEC by the time) to include SMT in their forthcoming Alpha 21464 (EV8) processor [Diefendorff99], which was cancelled before production.

The processor would have provided a 4-thread core with shared caches. Each thread was to be abstracted as a logically independent processor called a "thread processing unit". Almost all processor resources would have been dynamically shared with only the register allocation tables being duplicated per-thread. Compaq had planned to provide a multiple channel direct interface to RAMBUS memory in order to satisfy a 4-way SMT's significant memory bandwidth requirement. In his *Microprocessor Report* article, Diefendorff speculates that the processor could have had more than 3MB of on-chip L2 cache.

The extra hardware needed for SMT would have added 6% to the processor's die size [Preston02].

2.2.2 Intel Hyper-Threading

Intel introduced SMT into the Pentium 4 [Hinton01] processor as "Hyper-Threading" [Intel01c, Marr02, Koufaty03]. Intel state that adding Hyper-Threading to the Xeon increased the chip size and peak power usage by approximately 5%. In common with the proposed Alpha, Hyper-Threading presents the threads as logically independent processors. In traditional Intel style, backwards compatibility was maintained by allowing operating systems to detect and use the logical processors in the same way as physical processors in a multiprocessor system thereby allowing immediate use of the hardware by standard multiprocessor-capable operating systems. The first processor family to implement Hyper-Threading was the Pentium 4 Xeon, aimed at the workstation and server market. Later the technology was also incorporated into the desktop Pentium 4 [Krewell02] and subsequently the mobile version of the processor. The core processor in each of these variants is essentially the same; the main differences are with the non-core components such as caches. The following description uses the term "Pentium 4" in the general sense to cover all variants.

The core microarchitecture of the Hyper-Threaded Pentium 4 is based on the Intel "NetBurst" microarchitecture of the original Pentium 4. In common with recent generations of Intel IA32 processors the Pentium 4 dynamically translates program instructions into RISC-like micro-operations ("uops") which are natively executed by the processor core. A problem with the IA32 instruction set is that it contains variable length instructions. This means that successive instruction address are dependent - it is impossible to tell where the next instruction starts until some way in to the decoding of the current instruction. Decoding is a stateful operation and sharing of the decoding resource is difficult without expensive duplication of state. For this reason Intel chose to multiplex the use of the decoding stages of the pipeline between threads at a several-instruction granularity. The decode bandwidth need not be shared evenly; if one thread is idle or stalled, the other thread can utilise the full bandwidth.

Decoded uops are cached in the *trace cache* (TC) which is used instead of a level 1 instruction cache. The set-associative TC is dynamically shared by threads with each line tagged with its thread. Uops are fetched from the TC (or microcode ROM for particularly complex instructions) one thread at a time, alternating each cycle as necessary. Once again, an idle or stalled thread will allow the other thread to use the full uop fetch bandwidth available. The fetched uops are fed into a queue partitioned to give each thread half of the entries.

The out-of-order superscalar core takes uops from the queue. An allocator chooses instructions from the queue and allocates buffers to them. Fairness is provided by limiting the number of each buffer type (re-order, load and store buffers) that a thread can have to half of the number of buffers available. Additionally if both threads have instructions in the queue then the allocator will alternate between threads' uops each cycle. If a thread is stalled then the other thread will be given full allocate bandwidth but will still be limited to its share of buffers.

The architectural registers are then renamed to internal physical registers. Since each thread has a separate architectural register set, the *register alias table* tracking the architectural to physical register mappings needs to be duplicated for each thread.

The dataflow execution core need not be aware of which thread each instruction belongs to. However to ensure fairness the processor limits the number of entries in each scheduler queue that each thread may have. The Pentium 4 can issue up to six uops per cycle to the seven execution units (two integer ALUs, a floating-point move unit, an integer shift/rotate unit, a floating point unit, a load unit and a store unit). Completed uops are placed in the re-order buffer and are retired in program order. Retirement, like fetching, will alternate between threads but give all of its bandwidth to a thread if the other thread has no instructions to retire at that time.

The cache hierarchy is essentially physically addressed so can be used by both threads with no explicit tagging or partitioning. The data TLB is dynamically shared and each entry is tagged by logical processor. The instruction TLB is replicated for each thread but is small.

The way resources are shared: dynamically, statically partitioned or duplicated, is summarised in Table 2.1.

The IA32 `HLT` instruction is used on single-threaded processors to put the processor into a low-

Area	**Duplicated**	**Dynamically Shared**	**Tagged or Partitioned**
Fetch	ITLB Streaming buffers	Microcode ROM	Trace cache
Branch prediction	Return stack buffer Branch history buffer		Global history array
Decode	State	Logic	uop queue (partitioned)
Execute	Register rename	Instruction schedulers	Retirement Reorder buffer ($\leq$ 50% per thread)
Memory		Caches	DTLB

Table 2.1: Resource division on Hyper-Threaded P4 processors.

Feature	**Northwood Xeon**	**Prescott**
Pipeline stages	20	31
Store buffers	24	32
L1 data cache	8kB 4-way	16kB 8-way
L2 cache	512kB 8-way	1MB 8-way

Table 2.2: Northwood and Prescott Pentium 4 differences [Glaskowsky04].

power dormant mode which is exited on reception of an interrupt. This instruction is typically used in operating system idle tasks. With Hyper-Threading both logical processors need to be `HLT`ed in order for the physical processor to go into a low power mode but a single `HLT`ed logical processor will go into the logical dormant state. In this scenario the processor will recombine partitioned resources so that the non-`HLT`ed processor can use all available fetch, decode and execution bandwidth. The performance of the active thread should be the same as if Hyper-Threading was disabled or even non-existent.

The logical processor abstraction extends to the handling of interrupts. Each logical processor has its own interrupt controller and interrupts can be sent to each logical processor in exactly the same manner as in traditional multiprocessor systems.

2.2.2.1 *Prescott*

The "Prescott" Pentium 4 processor from Intel is the second implementation of Hyper-Threading. Table 2.2 summarises the key architectural differences from "Northwood", the first instantiation of a Hyper-Threaded processor described above. A number of the differences between the two cores suggest that the Prescott may be better suited to simultaneous multithreading than its predecessor. In particular the Prescott's larger, more associative L1 cache is likely to perform better than the smaller Northwood cache when the two threads are making many memory accesses. Intel also claim to have improved the control logic associated with Hyper-Threading. In Chapter 3 I measure both versions of the processor to evaluate the effectiveness of the Prescott changes.

Several microarchitectural enhancements have been made to the Prescott core [Boggs04]. Several entries have been added to the floating-point schedulers to improve floating-point performance under Hyper-Threading. The number of outstanding L1 data cache accesses has been doubled to 8. This has negligible effect on single-threaded performance but is useful for Hyper-Threading. Extra write-combining buffers were also added to complement the extra store buffers. These changes reduce the chance of a thread stalling when both threads are making a significant number of memory writes.

Intel have increased the available parallelism in the memory access circuits. The Northwood core serialised page table walks initiated by both threads. This is problematic if one thread's walk causes a cache miss and the other thread requires a page table walk. The Prescott core can perform both walks in parallel.

The problem of one thread stalling in the core causing performance loss to the other thread, or resource wastage, is tackled by reducing the time for the trace cache to react and dedicate all of its resources to the non-stalled thread.

A number of new instructions have been added to the Prescott processor [Intel03]. Of particular interest is the `MONITOR`/`MWAIT` pair. In summary, `MONITOR` allows a region of memory of sized fixed by the implementation (believed to be a cache line) to be marked for use by subsequent issues of the `MWAIT` instruction. `MWAIT` causes the processor to block until the specified region of memory has been written to. The actual operation is a little more detailed. For example, `MWAIT` enters an "implementation-dependent optimized state" until one of a list of events occurs, including writes to the region of memory and interrupts. `MONITOR` and a loop containing `MWAIT` can be used as a reasonably light weight inter-thread synchronisation mechanism.

2.2.3 IBM Power5

IBM's Power5 processor contains two processor cores each of which is capable of running two SMT threads [Kalla04]. Up to 32 packages can be built into a symmetric multiprocessing system. IBM claim that providing more than two threads per core is not justified due to the extra complexity outweighing the diminishing returns from the additional threads, particularly with cache thrashing.

In common with Intel's Hyper-Threading the Power5 fetches from only one thread in a clock cycle with the instruction fetch stage alternating between the threads. The threads share the instruction cache and instruction TLB.

As with Hyper-Threading, the Power5 has per-thread call return prediction stacks as a shared stack would be of no value. The branch prediction state is entirely shared in the Power5 whereas Intel choose to share the main global history table (with thread-tagged entries) but give each thread its own branch history buffer.

Both the Power5 and Hyper-Threading have dedicated per-thread instruction fetch queues (called "streaming buffers" with Hyper-Threading). These queues hold instructions awaiting decode.

One of IBM's major advances over Intel's initial Hyper-Threading offering is the ability to assign priorities to threads. This function is implemented in the choice of which thread to decode instructions from in a given cycle. This particular location in the pipeline is appropriate because it is the last stage where the threads have little interference with each other before instructions enter the shared core. If the throttling of threads to implement thread priority was to have been performed at the fetch stage then very little would be gained as the subsequent instruction queues are duplicated and building a shared one would save only a small amount of silicon. By allowing both threads to fetch as much as they can fit into the instruction queues allows instruction cache misses to be tolerated. The back pressure from the throttled decoding will have the desired effect and limit the fetch rate for the lower priority thread. Having instructions from a low priority thread ready to be decoded reduces the amount of time the execution units are idle if the high priority thread stalls.

Once a group of instructions from one of the threads has been chosen and routed through the decoding and register renaming stages the instructions feed into issue queues for the eight execution units. The instructions are now independent of their threads therefore the entire execution core is dynamically shared by the threads.

The main difference between Power5 and Intel Hyper-Threading in the decode and execute areas of the pipeline is that the Power5 uses decode for thread resource usage balancing while Intel places limits on access to shared execution resources such as the re-order buffer and the number of instructions from each thread that may be scheduled per cycle. The Intel approach is much more of a static partitioning of resource while the Power5 is more faithful to the dynamic model of the University of Washington.

The *global completion table* (GCT), records the completion of instructions in groups of up to 5 instructions from each thread. The GCT has 20 slots for such groups, shared between threads. The Power5 can commit one group from each thread per cycle.

In common with Intel, IBM made some improvements to the caches when introducing multi-threading. Compared to the non-SMT Power4 the Power5 has doubled associativity on both level 1 caches (although they remain the same size at 64kB I and 32kB D) and introduces a fully associative D-TLB (still 128 entries).

To provide a good level of fairness between threads, something Hyper-Threading can have difficulty doing, the Power5 implements *dynamic resource balancing*. The logic monitors the GCT and load miss queues to determine if a thread is "hogging resources". A particular problem is a blocked dependency chain due to an L2 miss; this can cause instructions to back up in the issue queues preventing further dispatch thereby slowing the other thread. The processor has a threshold of L2 misses which if reached causes the thread to be throttled. The mechanisms used for throttling depend upon the situation:

- reduce the thread's priority (used if there are too many GCT entries),
- inhibit the thread's decoding until congestion clears (used if more than the threshold number of L2 misses), and
- flush the thread's instructions that are waiting for dispatch, and suspend decoding for that thread until congestion clears (used if the thread is executing a long latency instruction such as a `SYNCH` memory barrier).

Idle occupation of buffer slots by instructions from a thread stalled on L2 misses has been shown to be a problem in research work [Limousin01].

The *adjustable thread priority* implemented in the decode stage is used to vary the share of execution resources given to each thread. Eight levels are defined; 0 stopping the thread completely and 1 to 7 being used for low to high priorities. The difference between the priorities of each thread determines the degree of bias given to the decode rates and therefore the use of the execution resources. If both threads have a priority of 1 then the processor throttles back the decode rate in order to conserve power. Some priorities can be set by privileged instructions only, some from all levels.

The processor can be run in a single-threaded mode in which all physical resources are given to the one thread. This mode can be entered either disabling the second thread entirely (using the system BIOS) or by the operating system putting one thread into a dormant state. Intel Hyper-Threading supports equivalents of both scenarios.

2.3 Operating System Support

In this section I describe the various possible interactions between simultaneous multithreaded (SMT) processors and operating systems. I begin by describing the main differences between a uniprocessor (UP) or symmetric multiprocessor (SMP) system and an SMT system. I explain how features are abstracted to avoid the need to provide SMT-specific OS support but how the OS could improve performance if it is aware of the differences.

2.3.1 An Extra Level of Hierarchy

Abstracting threads in an SMT processor as logical processors is a convenient way to provide instant backwards compatibility; however, it introduces complications into the otherwise simple action of counting processors.

Many commercial operating systems are licensed for a particular number of processors. An interesting problem with building logical processors on top of physical processors is which level of the hierarchy should be counted for the license. If it is decided to count logical processors then a further complication is caused by the ability to disable threads. Should the licensing count be based on the number of threads enabled, or the total number possible? An argument for the latter case is that a check could be carried out at boot time and the threads re-enabled later. Hyper-Threading is now a standard feature on all new high-end Pentium 4 processors so if a system without SMT is required then the second logical processor on each package must be disabled. A license using the logical processor count is likely to be unfair in such a situation.

The two obvious choices for numbering the logical processors are:

1. number first by package and then by logical processor within each package,
2. number through the first logical processor in each package then through the second and so on.

The second method is useful in situations where the operating system is licensed for a particular number of processors and runs on the lowest numbers processors, ignoring the remainder. In the first numbering system this would cause entire physical packages to be ignored while logical processors compete on the active packages. The enumeration of processors can be performed by the OS or BIOS. In the latter case it is important that the OS knows how the BIOS performed the enumeration.

When the Intel Pentium 4 Xeon with Hyper-Threading first came out Microsoft Windows Server 2000 was unaware of the logical/physical processor distinction so used the BIOS enumeration for licensing purposes [Borozan02]. If the BIOS enumerated processors according to Intel specifications, with the first logical processor in each package counted first, then a 4 physical processor (each of two logical processors) machine running a copy of Windows 2000 licensed for 4 processors would use the first logical processor on each package and leave the other ones idle. This is effectively the same as running on a 4-way non-Hyper-Threaded machine so no harm is done although the extra benefit of the Hyper-Threading is not gained. A BIOS enumerating using the alternative method would cause Windows 2000 to use both logical processors on the first two packages leaving two physical packages completely idle in the machine described above. The successor Windows version, .NET Server 2003, was able to distinguish between logical and physical processors and was licensed based on the physical package count.

The Linux kernel was made aware of Hyper-Threading from version 2.4.17. It enumerates the processors itself and uses the first method outlined above, numbering through all the logical processors in the first package then all in the second and so on. Later versions extended this support further by recording the hierarchy as a specific case of the OS's generic NUMA support.

Knowledge of the difference between logical and physical processors is useful for load-balancing and scheduling. In a scenario where a system has two physical packages each of two logical processors, and has two runnable processes, the scheduler has to decide which processors to use and which to leave idle. A scheduler unaware of the processor hierarchy may assign the tasks to the lowest numbered processors; using the Linux enumeration method these would be the two logical processors of the first package. The entire second package would be idle which would not give the highest system throughput. Although Linux 2.4.17 was "Hyper-Threaded aware" it exhibited this problem; later versions were able to support the logical/physical processor distinction.

A further problem of the logical processor abstraction is when accessing per-package state such as when updating microcode and setting the memory type range registers (MTRRs) which are used to control caching for physical memory ranges. An operating system treating two logical processors as SMP may run into concurrency problems[1]. Linux now serialises these accesses.

2.3.2 Simultaneous Execution

The threads, or logical processors, of an SMT processor dynamically share and compete for processor execution resources. As I show in Chapter 3 one thread always slows down the other (compared to it running alone on the physical processor) and the mutual effect of the threads on

[1]Current usage should not be a problem as Intel require all processors in an SMP system to have the MTRRs set identically [Intel01b], their initialisation is write-only and their writing is idempotent.

each other can be quite detrimental. Therefore care has to be taken to avoid situations where threads execute instructions which perform no useful computation, such as busy-waiting. The variable effect of threads on each other is important to scheduler design, particularly where fairness, quality of service or priorities are being used. I address the problem of fair and efficient scheduling for SMT in Chapter 4; in this section I describe the issues surrounding thread priorities.

Redstone considered a number of aspects of operating system-SMT interaction [Redstone02]. He used a full-system simulator to evaluate the performance of operating system functions on an SMT processor. Unlike earlier work, the experiments were designed to take into account those periods of an application's execution which are executing inside the kernel. Redstone found that the greater level of data and code sharing within the kernel (compared to the user-space code) is beneficial to IO-bound applications such as the Apache web server. He noted that such workloads suffer on non-SMT systems because of the constant crossing of privilege boundaries and changes of control-flow. Whilst these negative effects are still present on SMT processors the positive sharing effects go some way to countering them. Additionally the latency-hiding ability of SMT makes up for the remaining performance shortfall giving such workloads a good net speedup over a non-SMT processor.

2.3.2.1 *Unproductive Execution*

There are a number of circumstances where instructions are executed but no useful computation is performed. These include spin locks, idle loops and timing loops.

A spin lock is a simple mechanism for a thread to wait for a resource held by another thread to be released. Spin locks are typically implemented as a loop around code that tests the lock's status. Spin locks are commonly used due to their simplicity but have undesirable performance effects in many scenarios as they cause processing resource to be used with no useful work being done. Spin locks are an attractive implementation choice when the chance of the contention for a lock is low. In this situation the common case of the lock being available entails a single test and a not-taken branch; a lock found to be held will entail costly spinning but if this is rare then over-all performance will be good.

On an SMT processor where one thread is holding a lock while another thread is spinning on it the shared execution resources of the processor mean that the spinning thread will have a performance impact on the lock-holding thread. Spinning is particularly resource hungry as the loop closing branch is easy for the processor to predict allowing the loop to be unrolled multiple times creating a large number of instructions to compete with the other thread(s) for execution resources.

There are a number of ways to avoid this problem.

- Avoid using spin locks; there exist a number of lock-free techniques [Greenwald96] most of which would be very beneficial in an SMT environment.
- Yield the processor while waiting on a lock; this would involve a costly entry to the scheduler but would prevent wasted processor time. Since most locks are only held for a short time a compromise is to spin for a limited time and then yield.

- Implement locks as a processor primitive. Tullsen *et al* describe hardware locks with explicit acquire and release instructions [Tullsen99]. When a thread tries to acquire a locked lock then the processor will cause all resources being used by that thread to be freed for use by the other threads. The thread is allowed to continue when the lock is released.

- Implement hardware support to reduce the performance cost of spinning. The Intel `PAUSE` instruction [Intel01a] can be inserted into the body of the spin lock loop. `PAUSE` is logically a no-operation instruction but causes an architecturally dependent delay in the issue of instructions (believed to correspond to the pipeline latency). Whilst this does not eliminate the cost of spinning it does reduce it. Intel's second generation Hyper-Threaded processor core, Prescott, introduced a new pair of instructions, `MONITOR` and `MWAIT` [Intel03]. `MWAIT` causes the thread to enter an "implementation-dependent optimized state" (which for a Hyper-Threaded processor should involve releasing held resources) until a region of memory specified with `MONITOR` has been written to by another thread. This mechanism can be used to build locks[2].

- Use unmodified spin-locks but provide hardware/OS support to limit their impact by lowering the (hardware) priority of the spinning thread. This is the method suggested by IBM for use on the Power5 processor [Kalla04].

A further application of spinning is the idle loop for processors that currently have no processes scheduled on them. The idle loop repeatedly checks for any new or migrated processes which will cause control to be handed to the scheduler. Clearly such behaviour on a logical processor on an SMT physical processor will consume execution resource thereby slowing the other logical processors. "Busy-waiting" idle loops are also problematic for non-SMT processors because the unnecessary use of resources consumes power and therefore generates heat and reduces battery life. An alternative to busy-waiting is to force the processor to enter a native idle state until such time as there is something to do. An example is Intel's `HLT` instruction which causes the processor to run in a reduced power (on post-386 processors) dormant state until an interrupt is received. On a Hyper-Threaded processor `HLT` causes the logical processor to enter the same dormant state and all partitioned and shared resources used by that logical processor are given to the other logical processor [Marr02]. If the second logical processor also executes `HLT` then the processor enters a lower power state.

Linux 2.6 makes use of the Prescott `MWAIT` instruction for the idle loop. The idle task waits on a part of the operating system data structure for that logical processor that is written to by scheduler functions executing on other logical processors. If the scheduler wishes to execute a process on a currently idle logical processor then it writes to this data structure which the idling thread will notice. This mechanism suffers less overhead than a traditional inter-processor interrupt but still allows the yielding of resources that using `HLT` would provide.

In early operating systems and applications it was common to use "timing loops" to cause a short delay. Timing loops are still found in modern operating systems but tend to be restricted to initialisation functions. A timing loop is a simple loop, or set of nested loops, that performs no useful work. The number of iterations is chosen to cause the loop to take a particular length

[2]At present the use of these instructions is limited to kernel mode, however it was intended to make them available to user level threads and it is likely that future revisions of the processor will allow this.

of time to execute. A typical operating system use of timing loops would be to insert a delay between sending a command to, and reading a response from a hardware device. Applications may use them to provide delay between frames in an animation. Timing loops are undesirable in modern systems for a number of reasons.

- Knowledge of the clock speed is required to calculate the number of iterations. Different clocks on different and newer systems will change the loop timing.
- Modern superscalar systems execute multiple instructions per cycle. This must be taken into account when deciding the number of iterations.
- Timing loops waste processor time. This is particularly important in multitasking systems where that time could be more usefully given to another task.
- Timing loops in device drivers where interrupts are disabled are problematic since other interrupts may be arriving and awaiting servicing.

The first two problems can be avoided by calibrating the loop against a known time period at runtime (or boot time) on the actual processor. This technique is used by Linux for processors such as the Intel 386 and 486 that do not support more efficient delays.

Timing loops on SMT processors suffer the same problem as spin locks - they cause detriment to other threads' performance while doing no useful computation. Additionally the delay for a given number of iterations of the loop will vary depending on the resources being used by the other thread(s). In a similar manner to spin-locks, processor facilities, such as the Intel `PAUSE` instruction can be used to reduce the impact of a timing loop.

Redstone performs a detailed analysis of the effect of spinning on a simulated SMT system [Redstone02]. He notes that one thread spinning uses resources that the other thread(s) could have used. In the particular microarchitecture simulated, the spinning thread uses more than its fair share of resources; in particular, the architecture favours threads that make better progress which is what the spinning thread appears to do. He compares spinning with Tullsen's *SMT locks* and observes slowdowns of up to three times when using spinning compared to SMT locks.

2.3.2.2 *Priorities*

Operating systems generally support process priorities to influence scheduling. A common scheme is to assign a process some form of static base priority which is varied dynamically during execution to avoid starvation.

Take the scenario of a high priority single-threaded process *H* and a low priority background process *L*, which are both compute-bound. The desired behaviour is that *H* gets the largest share of the processor time with *L* being scheduled around it. *L* will always end up with some fraction of the processor time due to the starvation-avoiding dynamic priority change. A uniprocessor system being preemptively timeshared between the processes will exhibit the desired behaviour because the scheduler will bias the ratio of processor time given to the processes. A multiprocessor system will provide an entire processor for each of *H* and *L*; this does not differentiate the priorities but mostly fulfils the requirements of allowing *H* to continue unhindered while *L* uses the remaining resources (the processes do compete for access to memory so *L* could still hinder

H if the processes have high memory bandwidth requirements or are causing cache invalidations due to sharing memory). A single package, two thread SMT processor being used as two logical processors will have each of *H* and *L* executing on its two logical processors. The problem here is that both processes are dynamically sharing and competing for the processor resources. In an idealised SMT processor each would be eligible to get 50% of the processor resources. In practice it is possible that either process, regardless of its priority, would make better progress than the other due to the imperfect nature of resource sharing (see Chapter 3). The process priority is not taken into account since the multiprocessor scheduler given two runnable processes and two logical processors will always chose to run each process on an available processor.

The problem of extending the process priority into the SMT processor can be addressed in hardware or software. If the processor was to support a form of priority for the SMT threads (logical processors) then the above scenario could be easily accommodated. Hardware thread priorities can be implemented by imposing a ratio between threads of the number of instructions fetched (or decoded in the case of IBM's Power5 [Kalla04]) in some small time window. The current implementations of Intel Hyper-Threading (the Northwood and Prescott cores) do not have any support for explicit thread priorities.

In the absence of hardware support the operating system scheduler can try to provide the desired effect while still utilising the simultaneous nature of the processor. Lower priority tasks could be limited to only a fraction of the processor time forcing one of the logical processors to be idle even if there is a runnable low priority process. Assuming that one logical processor being idle allows all processor resources to be used by the other logical processor, this method would allow the high priority process to run unimpeded for much of the time, only suffering a slowdown for the fraction of the time the low priority process was running. In general this scheme provide a worse system-wide performance than constantly using both logical processors but does give the high priority task a better individual performance. An extension to this technique would have all low priority processes being scheduled on one logical processor while the fewer number of high priority processes share the other. A modification to these schemes would be to use processor performance counters to monitor the consumption of resources by low priority tasks, perhaps biased by their actual dynamic priority, and use this to limit their scheduling. The use of performance counters to assist and improve scheduling for SMT processors is discussed further in Chapter 3.

2.3.3 Cache Considerations

Threads executing on SMT processors, as with other multithreaded processors, share the cache hierarchy. One of the original motivations for SMT was to provide the processor with more work to do while waiting on a cache miss. However, the extra thread(s) with their own memory access requirements cause a greater demand on the caches often leading to more contention and capacity misses. The performance of the processor will be heavily influenced by the opposing effects of increased cache misses and greater miss tolerance. I show this trade-off occurring in practice in Chapter 3.

A particular problem occurred with Linux when Intel Hyper-Threaded processors first became available. The user stacks in Linux were allocated on 64kB boundaries which meant that they started from the same location within the processor's 8kB (2kB x 4 way) level 1 data cache. Most applications only use a small amount of the stack which leads to cache conflict due to the limited associativity of the cache. For single-processor context switching this is not a problem as the overhead of refilling the cache after a context switch is very small compared to the scheduling quantum. With SMT the continual contention greatly increases the level 1 cache miss rate. Stack *aliasing* was avoided in later versions of Linux by offsetting the start address of each process' stack to increase the likelihood of the simultaneous threads accessing disjoint areas of the cache.

The Intel Pentium 4 processor uses a *partial virtual tagging* scheme for its level 1 cache. Accesses from different threads that map to the same partial tag will cause contention. To reduce this Intel introduced into the Prescott processor a *context identifier* bit for each logical processor, kept with the partial tag [Boggs04]. These bits are set or cleared depending on whether the logical processors share a common address space; if they do the bits are the same, if not they differ. Threads executing in different address spaces are guaranteed to have different partial tags and therefore to not contend in this way

In their work on the performance of database systems on SMT processors, Lo *et al* describe some of the cache issues that must be considered in order to best use an SMT processor [Lo98]. The paper describes cache interference, data sharing effects and page placement schemes.

Constructive interference is where the threads can benefit from shared cache contents. This is most likely to be seen when a true multithreaded workload is running where code and data are likely to be shared. The effect may exist, to a smaller degree, where multiprogrammed task sets share common library code.

SMT has an additional benefit over multiple physical processors; if one thread is writing to a cache line, the other thread(s) can read or write the same line without the invalidation penalty that would happen in a conventional multiprocessor system.

When threads compete for space in the cache then the system experiences *destructive interference*. The situation is worst when the threads have a high degree of homogeneity in their memory layout and access patterns. Lo *et al* studied a multithreaded database system and found that each thread had a number of hot spots in its per-thread local data. Each of these hot spots was located at the same virtual address in each thread's address space. The simulated processor in this study used a virtually-indexed, physically-tagged level 1 (L1) cache and a physically-indexed, physically-tagged level 2 (L2) cache. There are two main effects on the cache performance:

- virtual address space layout; having hot spots at the same virtual addresses means they have the same L1 indexes and therefore will lead to conflicts within the relevant cache line sets.
- virtual page placement in physical memory; an unfortunate mapping of virtual to physical pages could cause hot spots to map to the same indexes, and hence the same set, in the L2 cache.

Lo *et al* describe two software mechanisms to deal with these two effects: page placement policy to reduce L2 conflicts and application-level offsetting to reduce L1 conflicts. Most operating systems already use a page placement policy to try to reduce cache contention between context-switching processes. This policy is obviously more important for simultaneous executing threads. Kessler and Hill describe a scheme based on giving each physical frame a colour (or "bin") [Kessler92]. Two pages with the same colour index to the same L2 cache line set. The operating system can use the colours to help decide which physical frames to map logical pages to using one of the following schemes:

- Page colouring; consecutive virtual pages of a process are mapped to consecutive colours. This mainly prevents contention within a process but can still lead to inter-process contention. Some operating systems randomise the start colour by hashing the process ID, for example.
- Bin (colour) hopping; when a new virtual page is allocated, the mapped physical page is chosen from the next colour in a round robin fashion. This means that pages allocated at the same time will have different colours.

Lo *et al* find that non-randomised page colouring is less effective in the case of their database on SMT because the layout will be the same for each thread and therefore reintroduces aliasing. They found that adding randomisation helps but bin hopping was best because it is most likely to assign the same hot spots in different threads to pages mapping to different areas in the cache.

Page placement only affects the L2 cache since it is only the physical addresses which are changed. A virtually-indexed L1 cache will still see hot spots with the same virtual addresses indexing into the same line set. The technique of application-level offsetting can be used to cause the hot spots to appear at different virtual addresses in each thread's address space. The starting virtual address can be offset (by the loader or the application) by some number of pages determined by the process ID. This technique also helps with page colouring's effect on the L2 cache since it introduces more randomisation and heterogeneity.

False sharing is a cache consistency problem experienced by shared memory multiprocessor systems. Two threads executing on different processors both access disjoint data that happen to be adjacent in memory and appear on the same cache line. Because the processors provide cache consistency on a per-line granularity the disjoint nature of this access pattern is ignored and it appears that both threads are trying to use the same data. The result is a constant "ping-pong" of that cache line between the processors. The same scenario applied to SMT should be less of a problem as the cache is shared. It has been suggested that false sharing on SMT systems is beneficial because less space is used in the cache compared to forcing the threads' data to be on different lines; additionally one thread will effectively prefetch the shared line for the other [Lo97a, McDowell03]. In practice this may not be the case due to the per-line granularity of metadata being extended further into the processor. The Intel Pentium 4 "NetBurst" microarchitecture allows the reordering of load and store instructions. If a store to a location followed by a load from that location are reordered then the processor will detect the memory ordering violation and replay the load using the correct value - a fairly expensive operation that disrupts execution of both Hyper-Threads. The load and store buffers that support this functionality are of cache line size. If a line is written to temporally after it is read by a store instruction that was earlier in the program than the load, a violation is assumed even if the actual location was different. This implementation is *safe* and allows for the complex variable length and non-aligned references common in the Intel architecture. The effect on a Hyper-Threaded processor in the presence of false sharing is that a disjoint write by one thread to a line that has been speculatively read by the other thread will cause a memory ordering violation to be assumed. The impact of this behaviour will depend on the frequency of occurrence but in the worst case could cause a six times slowdown over a more careful memory allocation[3]. It is therefore wise to use the same techniques for memory alignment for Hyper-Threaded processors as for multiprocessors.

2.3.4 Energy Considerations

In Chapter 3 I describe experiments measuring the difference in performance between SMT and SMP systems. I show that a two processor SMP system always outperforms a two-thread SMT system as would be expected. However, in some situations the difference between the two is not great. The processing performance per unit cost (financial or energy consumption and dissipation) is a useful metric. SMT will generally have a higher performance per unit cost than SMP. This suggests that an SMT system may be a suitable alternative to an SMP system in situations were financial cost or energy consumption are more important than raw throughput. Typical examples include laptop computers, where energy consumption matters, and high density data centres, where heat extraction is often a limiting factor.

Chip multiprocessors (CMP) would come somewhere between SMP and SMT; the degree of resource sharing of a CMP system is less than SMT so the statistical multiplexing of demands onto resources is limited and would generally yield a lower processing performance per unit cost. CMP systems typically share a level 2 cache so will generally benefit from better cache utilisation (so long as pathological aliasing problems are avoided) and therefore will be more cost (energy) efficient than a multichip SMP system. Both CMP and SMT, compared to SMP, contain multiple logical processors (SMT threads or cores) in a single package; this will reduce the amount of ancillary support required, such as external interface circuits, and therefore reduce the energy cost of a system.

In reality the situation is somewhat more complex with the “cost” including other components of the system which would be similar for SMT, SMP and CMP systems.

In Section 2.3.1 I described how a naïve scheduler could load-balance incorrectly by running two processes on the two logical processors of a single physical processor while leaving the other physical processor idle. While this would give a lower system throughput than could be achieved, it has benefits (perhaps unintentionally) for energy consumption. Two physical processors each with only one active logical processor will consume two processors worth of energy - Intel and IBM’s SMT processors dedicate all resources to the single running thread leaving no scope for powering down areas of the chip. If the two processes were to be running on the two logical processors of one package then the other package would have two idle threads and could therefore put itself into a low power state. The result is a substantial energy reduction at the expense of throughput. The correct action depends on the relative importance of throughput and energy consumption of a particular system at a particular time.

A further consideration is that of processor clock reduction to reduce energy consumption. Many processors, particularly those aimed at the mobile market, can be instructed to reduce their clock frequency by the operating system. The clock of an SMT processor is common to the physical package rather than the logical processors. Any clock reduction will affect all logical processors

[3]“Avoiding False Sharing on Hyper-Threading Technology-Enabled Processors” by Phil Kerly. Available at `http://www.intel.com/cd/ids/developer/asmo-na/eng/downloads/19980.htm`

in the same package. To avoid unexpected performance loss, Intel suggest that the BIOS or OS set the package clock rate to be the highest requested for both logical processors. This behaviour further supports the use of the "incorrect" process allocation described above because an entire physical package can be idled and have its clock rate slowed. CMP systems will not suffer this problem if the clocks on each core can be independently controlled.

SMT processors were designed to increase the processor throughput. However, if a fixed throughput is required then an SMT implementation may be more energy efficient than a single-threaded processor. Seng *et al* suggest that for a given level of instruction throughput, an SMT processor uses less energy per instruction than a single-threaded superscalar processor [Seng00]. This is because it is easier to extract parallelism from multiple threads than from a single-thread using aggressive speculation. Furthermore, it has been shown that SMT is preferable to multicore architectures for typical fixed-throughput workloads [Kaxiras01, Chen02].

2.4 Summary

In this chapter I have described simultaneous multithreading (SMT) and have introduced the hardware realisations of the technology. These include Intel's Hyper-Threading technology introduced into the Pentium 4 processor. I use this processor in the analysis studies in Chapter 3. I have introduced and discussed areas of operating system support important for SMT processors and highlighted areas where current support is lacking. In Chapter 4 I develop an SMT-aware process scheduler that is sensitive to the performance characteristics of SMT.

Chapter 3

Measuring SMT

Intel's Hyper-Threading technology is the first commercial implementation of simultaneous multithreading (SMT). The availability of systems based on this technology presents an opportunity to measure their realised performance and compare the results to the original research designs for SMT architectures. This chapter describes a series of experiments performed on a Pentium 4 Hyper-Threaded processor in order to assess the impact of threads upon each other. The results from these measurements are used in Chapter 4 to develop a scheduler sensitive to the characteristics of SMT processors.

3.1 Related Work

3.1.1 Simulated Systems

Early work in the development of SMT relied on simulation [Tullsen95, Tullsen96b, Eggers97]. The results were generally very promising for SMT and motivated its further development. As the design progressed and interest widened, a simulator, "SMTSIM", based on the out-of-order model was released [Tullsen96a]. SMTSIM has been used in much of the incremental development work, mainly in evaluating microarchitectural enhancements. The SMTSIM architecture utilises the Alpha instruction set. SMTSIM differs from Intel Hyper-Threading in both architecture (Intel CISC versus Alpha RISC) and microarchitecture. SMTSIM is able to support up to eight *contexts* (simultaneous threads) compared to Hyper-Threading's two and the microarchitecure of the former shares resources in a more dynamic way than Hyper-Threading. Measurement studies based on SMTSIM are useful in that they give a general indication of the behaviour that may been seen when running applications on SMT processors but cannot provide the specific details that will affect execution on real systems based on Hyper-Threaded processors.

Snavely *et al.* [Snavely99] defined *symbiosis* as the throughput rate of a batch of programs running together versus the throughput rate of the longest running program from the batch running alone. This definition provides an intuitive quantitative measure for batches of processes but does not fit well with more general loads. The authors investigated how symbiosis changed as different benchmarks were co-scheduled, noting the bottlenecks in some interesting cases. In later work on symbiotic scheduling [Snavely00, Snavely02], jobs were characterised by instruction mix and performance data gathered during a sampling phase of execution. The jobs were put together in "jobmixes" according to various policies in order to achieve good symbiosis.

Redstone *et al.* investigated the performance of workloads running on a simulated SMT system with a full operating system [Redstone00]. They concluded that the time spent executing in the kernel can have a large impact on the speedup measurements compared to a user-mode-only study. They report that the inclusion of OS effects on a SPECInt95 study has less impact on SMT performance measurements that it does on non-SMT superscalar results due to the better latency hiding of SMT being able to mask the poorer IPC of the kernel parts of the execution. This result is important as it means that a comparison of SMT to superscalar without taking the OS into account would not do the SMT architecture justice.

3.1.2 Real Hardware

Since its release, there have been some performance studies of the Hyper-Threaded Intel Pentium 4. Studies of real hardware are useful as the implementations of SMT processors differ from the simulated architectures used in the studies described above. When measuring a real system many more factors are taken into account than can be incorporated into a simulation.

Grunwald and Ghiasi investigated thread interaction effects on the Hyper-Threaded Pentium 4 [Grunwald02]. Using synthetic workloads they demonstrated that the execution rate of a thread running on one logical processor can drop by as much as 95% (where 50% would be the worst expected if everything was fair) with a carefully crafted (possibly malicious) process running on the second logical processor. In particular they find that a process making extensive use of self-modifying code will cause trace-cache flushes which affect the performance of both logical processors considerably. They argue that, although possible in software, fairness should be ensured by the hardware and propose some microarchitectural solutions which generally involve the processor monitoring the frequency of certain events occurring and throttling threads based on some policy.

The IBM Linux Technology Centre performed a series of microbenchmark and application level benchmark tests to compare the performance of a processor with Hyper-Threading enabled and disabled [Vianney03]. They noticed very little difference on most of the microbenchmarks and single applications but speedups of 20 to 30% on multi-threaded applications. They then executed the multi-threaded application benchmarks on a Linux kernel modified with many of the features described in Section 2.3 of this dissertation. Speedups of up to 60% were observed. The main contributor to the improvement was the modified scheduler's knowledge that two logical processors exist on a physical processor therefore share a cache. This leads to extra flexibility in scheduling, particularly when waking up a blocked process, as there is no significant penalty in "migrating" a process to the other logical processor in the same package.

Intel investigated the performance of data-parallel workloads threaded with OpenMP [Magro02]. They observed speedups of between 5% and 28% over a range of compute-intensive applications when using Hyper-Threading compared to running the threads serially. For comparison the workloads were run on a dual processor SMP system (without Hyper-Threading) giving speedups of between 54% and 100% over serial execution. The investigators noted that data-parallel workloads typically present very similar instruction streams which could compete for processor resources but still gain a benefit from Hyper-Threading. Intel have also measured media applications running on Hyper-Threaded processors [Chen02]. On average the processor utilisation was found to increase by around 20% when Hyper-Threading was being used.

Since my work was performed, Tuck and Tullsen have performed a similar study [Tuck03]; their measurements confirm my own. Where relevant I compare my results to those of Tuck and Tullsen. They used Intel's VTune performance analyser to track performance counters in order to explain a few of the interesting interactions. The results I present in this chapter take the performance counter measurement further. My experiments look at the bias in performance of two concurrently running processes and I make more extensive use of performance counter data to investigate the reasons for the observed effects.

3.2 Experimental Configurations

In this section I describe the common tools and infrastructure used in the experiments described in this chapter. The particular method used in each experiment is described in the relevant section.

3.2.1 Performance Counters

The Intel Pentium 4 provides a rich set of hardware performance counters. These count events ranging from instruction issue, execution and retirement through cache misses to memory bus activity. The number of event classes that can be counted greatly outnumbers the 18 available counters and there are restrictions regarding which combinations of events can be simultaneously counted.

The performance counters are per physical package. Most events can be filtered for either or both logical processors. This imposes a limitation on what events may be counted simultaneously. For example, counting instructions retired independently on each logical processor prevents mispredicted branches being counted.

The interface to the performance counter configuration is through the processor "model specific registers" (MSRs). Access to MSRs can only be performed from the most privileged mode necessitating support within the kernel. There exists software to allow access to the counters, including per-process virtualisation (achieved through context switching the counter configuration and values) [1] however this was too heavyweight for my needs. I developed a simple, low overhead interface to configuring and reading the counters using a Linux `/proc` file. The `/proc` file system allows quasi-files whose reads and writes are handled by kernel functions. My `/proc/perfcntr` file handled writes as commands to configure the performance counters. Reading the file caused all the performance counters and timestamps for each processor to be returned.

To ease the use of the counters I added a user-space tool to form the counter configuration bit patterns. The tool, `cpuperf`, takes a number of command lines describing the counters, event

[1]The package most popular for the Linux kernel is Mikael Pettersson's `perfctr`: `http://user.it.uu.se/ mikpe/linux/perfctr/`

Metric	**Counter** [Intel01b]	**Event mask**
x87 FP uOps	x87_FP_uop	ALL
Instructions Retired	instr_ret	NBOGUSNTAG, NBOGUSTAG
Branch Mispredictions	mispred_branch_retired	NBOGUS
Loads Executed	front_end_event uop_type	NBOGUS, BOGUS TAGLOADS
Trace Cache Misses	BPU_fetch_request	TCMISS
L1 D Misses	replay_event	NBOGUS 1stL_cache_load_miss_retired
L2 Misses	BSQ_cache_reference	RD_2ndL_MISS, WR_2nd_MISS
I-TLB Misses	ITLB_reference	MISS
D-TLB Misses	page_walk_type	DTMISS
Bus Activity	FSB_data_activity	DRDY_OWN, DRDY_DRV (Edge triggered)

Table 3.1: Performance counter configurations used in the experiments.

classes and other configuration parameters and creates the appropriate configuration bit patterns for the various MSRs. These are then communicated to the `/proc` file.

The counters are 40 bits wide and therefore may overflow during the course of experiments. It is possible to have the processor generate interrupts when an overflow occurs however this was unnecessary and, due to the cost and complexity of dealing with the interrupt, undesirable. Instead the post-experiment analysis detected overflows; the frequency of performance counter sampling was set such that it would be clear when an overflow had occurred.

Table 3.1 details the performance counter configurations for the counters used in the experiments described below. These event counters were carefully chosen from the large number available in order to provide a useful and informative picture of the processor's behaviour and to allow an insight into the reasons for the observed performance. Not all of these counters could be used concurrently.

3.2.2 Test Platforms

Most of the experiments were conducted on an system based on two Intel Pentium 4 Xeon Hyper-Threaded processors. These processors are based on the "Northwood" core (see Section 2.2.2). Some comparison experiments were performed on a system based on a Pentium 4 processor with the "Prescott" core; this processor has only recently become available and time limitations have only allowed a subset of the experiments performed on Northwood to be performed on Prescott. In all cases the systems used a RedHat Linux distribution with a locally built and modified version 2.4.19 kernel. This version of the kernel contains support for Hyper-Threading at a low level, including the detection of the logical processors and the avoidance of timing loops during normal

	Northwood	Prescott	Tuck and Tullsen
Model	Intel SE7501 based	Dell Precision 360	
CPU	2 x P4 Xeon 2.4GHz HT	1 x P4 2.8E GHz HT	1 x P4 2.5GHz HT
L1 cache	8kB 4 way D 12k-uops trace I	16kB 8 way D 12k-uops trace I	8kB 4 way D 12k-uops trace I
L2 cache	512kB 8 way	1MB 8 way	256kB 8 way
L3 cache	none	none	none
FSB	400MHz	800MHz	
Memory	1GB DDR DRAM	1GB DDR DRAM	512MB DRDRAM
OS	RedHat 7.3	RedHat 9.0	RedHat 7.3
Kernel	Linux 2.4.19	Linux 2.4.19	Linux 2.4.18smp

Table 3.2: Experimental machine details.

operation. The kernel was modified with a variation of the `cpus_allowed` patch[2]. This patch provides an interface to the Linux `cpus_allowed` task attribute and allows the specification of which processor(s) a process can be executed on. This is particularly important as the scheduler in Linux 2.4.19 is not aware of Hyper-Threading. Use of this patch prevented threads being migrated to another processor (logical or physical). The `/proc/perf` file described above was used to allow lightweight access to the processor performance counters.

Details of the experimental machines are given in Table 3.2. The Northwood machine contained two physical processors each having two logical processors (Hyper-Threads); the Prescott machine had a single, two-logical processor, package. Also shown in the table are details given by Tuck and Tullsen for their experimental machine [Tuck03] to which Intel gave them early access which would explain the non-standard clock speed and L2 cache combination.

3.2.3 Workloads

3.2.3.1 *SPEC CPU2000*

Benchmarks from the industry standard SPEC CPU2000 suite [SPEC] were used in most of the experiments. This suite contains integer and floating-point benchmarks compiled from C, C++ and Fortran. The benchmarks are generally processor- (and sometimes memory-) bound and fit within the system memory. They perform relatively little I/O or other operating system activity. Table 3.3 shows the benchmarks used along with a brief description of each.

Experimental runs were run to completion (including successive runs with different input data where relevant) and used the reference data sets. Much of the simulation-based related work uses reduced data sets or executes only a fraction of each benchmark. As I will show, the changing phases of the benchmarks' execution limit the effectiveness of such an approach. The executables were compiled with GCC 2.96 using a common set of optimisation flags (mainly `-O2`). The

[2]The cpus_allowed/launch_policy patch was posted to the linux-kernel mailing list by Matthew Dobson in December 2001

Fortran-90 benchmarks, *178.galgel*, *187.facerec*, *189.lucas* and *191.fma3d* were not used due to GCC not supporting this language.

3.2.3.2 *Desktop Applications*

Although the SPEC CPU2000 suite provides a varied set of benchmarks and allows controlled, repeatable experiments, it is not intended to be representative of workloads commonly used on desktop computers. To address this, the experiments based on the SPEC suite were supplemented with some using a selection of benchmarks based on desktop applications.

mp3 The command-line music player `mpg123` decoded approximately 1 hour of music from 17

	Benchmark	**Description**
Integer	164.gzip	LZ77 compression
	175.vpr	FPGA place-and-route based on Dijkstra's algorithm
	176.gcc	GNU C compiler
	181.mcf	Combinatorial optimisation using much pointer arithmetic
	186.crafty	Chess playing using 64 bit arithmetic
	197.parser	A grammar parser for the English language
	252.eon	Probabilistic ray tracing
	253.perlbmk	Perl script interpreter, mainly text processing
	254.gap	Group theory processing
	255.vortex	In-memory object-oriented database
	256.bzip2	Burrows-Wheeler transform compression
	300.twolf	Chip layout place-and-route using simulated annealing
Floating-Point	168.wupwise	Physics: quantum chromodynamics
	171.swim	Weather prediction/shallow water modelling
	172.mgrid	Multi-grid solver: 3D potential field
	173.applu	Computational fluid dynamics using partial differential equations
	177.mesa	3D graphics library similar to OpenGL
	179.art	Image recognition using neural networks
	183.equake	Finite element simulation of seismic wave propagation in large basins
	188.ammp	Computational chemistry using ordinary differential equations
	200.sixtrack	High energy nuclear physics accelerator design
	301.apsi	Weather prediction: pollutant distribution

Table 3.3: SPEC CPU2000 benchmarks used in the experiments.

MP3 format files encoded at 192kbps. The decoded audio was sent to `/dev/null` and the decoding was allowed to progress as fast as possible (i.e. not in real time).

compile A build of the Linux kernel was performed. A version 2.6 kernel was used with as many modules and features as possible included. Both the kernel image and modules were compiled.

glx The `glxgears` application shipped with the XFree86 tools package on Linux systems. This application animates a set of 3 three-dimensional gears using the OpenGL graphics library. It records the frames-per-second (FPS) it achieves. Most of the processor time is spent within the X windows process so both the `glxgears` and `X` processes were given the appropriate logical processor affinity. To bring the performance result of *glx* into line with the other benchmarks which record an elapsed time, the frame-count values were interpolated to estimate a time to render 100,000 frames (approximately 5 minutes when running alone).

image A shell pipeline of three document processing commands: `pdf2ps`, `pstopnm` and `ppmtojpeg`. The sequence turns a large, complex PDF format file into a JPEG format image using postscript and the PPM raster format as the intermediates. Command line options were used to avoid the image being shrunk to a paper size.

3.2.4 Performance Metrics

The *performance ratio* (PR) of an individual benchmark running as one of simultaneously executing pair is described by its execution time when running alone divided by its execution time when running in the pair:

$$PR = standalone\_runtime/actual\_runtime$$

If a non-SMT processor is being timeshared in a theoretical perfect (no context switch penalty) round-robin fashion with no cache pollution then a performance ratio of 0.5 would be expected as the benchmark is getting half of the CPU time. A perfect SMP system with each processor running one of the pair of benchmarks with no performance interactions would give a PR of 1 for each benchmark. It would be expected that benchmarks running under two-thread SMT would fall somewhere between 0.5 and 1 because they should get at least half of the processor resources and are unlikely to do better than they would having the entire processor to themselves. A PR of anything less than 0.5 is an unfortunate loss.

The total *system speedup* for the pair of benchmarks is the sum of the PR values for the processes on all logical processors being considered:

$$system\_speedup = \sum_i PR_i$$

This speedup is compared to zero-cost context switching with a single processor so a perfect SMP system should have a system speedup of 2 while a single Intel Hyper-Threaded processor should come in somewhere between 1 and 2. Intel suggest that Hyper-Threading provides a 30% speedup which would correspond to a system speedup of 1.3 in my analysis.

A number of other performance metrics have been used in related work. These include *weighted speedup* and *SMT-speedup* from research work and various metrics from Intel.

Snavely *et al* define weighted speedup (WS) for a group of *n* simultaneously running threads for a period *t* in terms of their realised and stand-alone IPC [Snavely99]:

$$WS(t) = \sum_{i=1}^{n} (realised\, IPC_i / standalone\, IPC_i)$$

This metric is described as "weighted" because it allows for threads with natively low IPCs which would otherwise be penalised in simple IPC summation based metrics. WS is directly equivalent to my *system-speedup* metric when the IPC is averaged over the complete run of the program (the number of instructions cancels out).

Sazeides and Juan define a metric for comparing different SMT microarchitectures [Sazeides01]. They argue that simply measuring the total processor IPC is not sufficient because the effect on different threads may not be balanced and the actual work done may not have increased as much as the IPC suggests. They define a weight w_j for each thread *j* based on its single-threaded performance and the total number of instructions executed in the experiment *I*: $w_j = I_j/I$ where I_j is the number of instructions executed by thread *j* in the experiment. The metric for *SMT-speedup* is given by the processor IPC for the experiment multiplied by the sum of the per-thread weights each divided by the per-thread IPC from the single-threaded results. This is just a rearrangement of a more understandable metric in which the sum of times it would have taken to execute the threads in single threaded mode up to the point that they reached in the SMT experiment is divided by the time it took running under SMT.

Intel define *HT Scaling* in the same way as my *performance ratio* metric: a task's standalone execution time divided by its execution time running under Hyper-Threading. The difference is that Intel's metric is designed for use with parallel tasks so will yield results comparable to my *system speedup* metric. Intel also define a metric *HT effectiveness* based on Amdahl's Law which measures how well a parallel task performs on a Hyper-Threaded processor based on its observed performance on a dual-processor system and an assumed 30% Hyper-Threading speedup[3].

3.3 Thread Interactions

Intel Hyper-Threading is aimed both at multithreaded and multiprogrammed workloads. Intel initially marketed Hyper-Threading as a way to get two processors for the price of one; this pitch has since been toned down. The current claim, qualified by the usual disclaimers regarding performance depending on many variables, is:

> Built-in Hyper-Threading Technology (HT Technology) provides immediate value in today's computing environment by enabling the processor to simultaneously exe-

[3] "How to Determine the Effectiveness of Hyper-Threading Technology with an Application" by Shawn D. Casey. Available at `http://www.intel.com/cd/ids/developer/asmo-na/eng/microprocessors/ia32/pentium4/hyperthreading/20470.htm`

cute two software program threads. This lets you run two software applications in parallel without sacrificing performance [4].

In this section I measure the performance of running "two software applications in parallel". I use pairs of benchmarks from the industry standard SPEC suite [SPEC]. I perform the same experiments on both the Northwood and Prescott based systems in order to observe the different behaviours caused by the different implementations. The measurements and analysis of the Northwood core have previously been published [Bulpin04].

3.3.1 Experimental Method

The experiments were performed on the two systems described in Section 3.2.2. At the time of writing, Prescott systems are only beginning to become available and the range of processor models that are available with this core is limited. Only uniprocessor (in the physical processor sense) Prescott-based machines have currently been released. Therefore the two machines are not matched so results from the two machines should not be compared. The trends, however, are of interest.

The aim of the experiments was to find the performance ratio for each benchmark application running simultaneously with every other benchmark application. This *cross-product* of results provides two data points for benchmarks *X* and *Y* running simultaneously: *(X, Y)* to describe the performance ratio of *X* while running in that pair, and *(Y, X)* to describe the performance ratio of *Y* while running in that pair. Note that the two performance ratios will generally be different because the sharing of resources and the mutual slowing effect is not necessarily equal for both processes. The number of experiments required was reduced by recording data for both benchmarks *X* and *Y* in a simultaneously executing pair; this provided data points *(X, Y)* and *(Y, X)*.

For each pair of benchmarks studied the following procedure was used. Each benchmark was executed in a continuous loop on one of the logical processors. A random delay was inserted before entering the loop to give a staggered start. Execution time was ignored until both benchmark processes had completed at least one run. The experiment continued until both benchmarks had accumulated three timed runs each. Note that the benchmark with the shorter runtime will have completed more than three runs; only the first three timed runs were recorded. This method guaranteed that there were always two active processes and allowed the caches, including the operating system buffer cache, to be warmed. Note that successive runs of the one benchmark would start at different points within the other benchmark's execution due to the differing run times for both.

The experiments were run on Hyper-Threading, SMP (in the Northwood case) and single-processor context switching configurations. On the dual package Northwood the Hyper-Threading experiments were conducted using the two logical processors on the second physical processor and the SMP experiments used the first logical processor on each physical processor with the other processor idle (equivalent to disabling Hyper-Threading). The context switching experiments were run on the second physical processor and used the round-robin feature ("real

[4]From `http://www.intel.com`

time" scheduler option) of the Linux scheduler with a modification to allow the quantum to be specified. In all cases the machine was configured to minimise background system activity since these experiments were concerned solely with the interaction of the two threads; this was of particular importance on the single-package Prescott system.

A set of base run times and performance counter values were measured by running benchmarks alone on a single physical processor. A dummy run of each benchmark was completed before the timed runs in order to warm the caches. A total of 9 timed runs were made and the median run time was recorded. This procedure was performed twice; once using a single logical processor with the second logical processor idle (but still with Hyper-Threading enabled), and once with Hyper-Threading disabled. The run times for both configurations were almost identical. This behaviour is expected because the processor recombines partitioned resources when one of the logical processors is idle through using the `HLT` instruction [Marr02].

3.3.2 Results

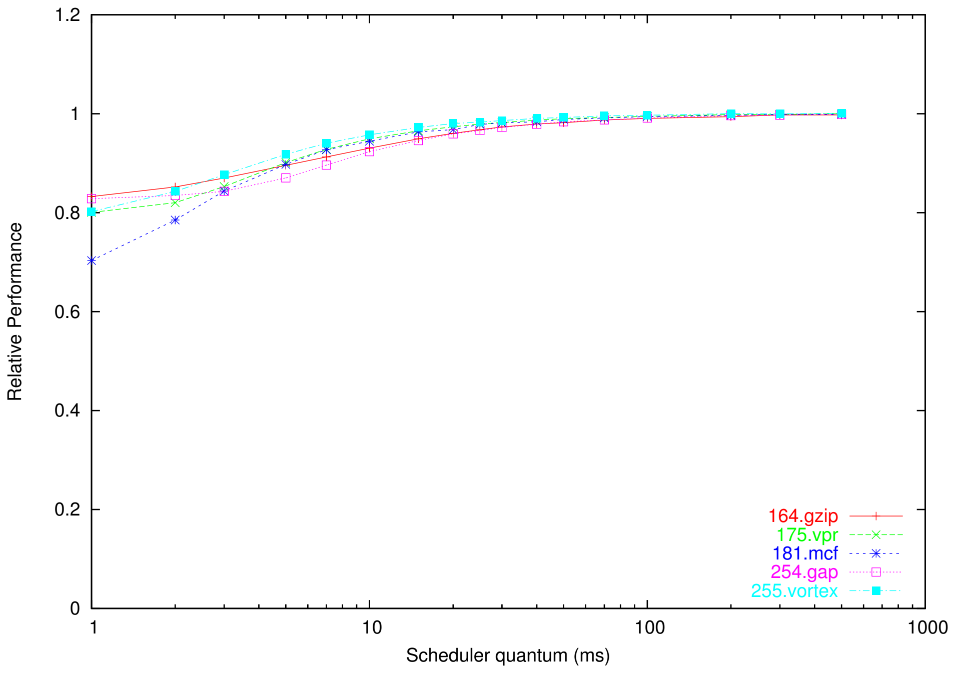
For the purposes of the analysis, one process was considered to be the *subject* process and the other the *background*. The experiments were symmetric therefore only one experiment was required for each pair but the data from each experiment was analysed twice with the two processes taking the roles of subject and background in turn (except where a benchmark competed against a copy of itself).

In the following sections I present a summary of results from the Hyper-Threading and SMP experiments. The single-processor context switching experiments using a quantum of 10ms generally resulted in a performance ratio (PR) of no worse than 0.48 for each thread, a 4% drop from the theoretic zero-cost case. As well as the explicit cost of performing the context switch the cache pollution contributes to the slowdown. The relatively long quantum means that the processes have time to build up and benefit from cached information. I do not present detailed results from these experiments however the impact of the quantum length can be seen in Figure 3.1.

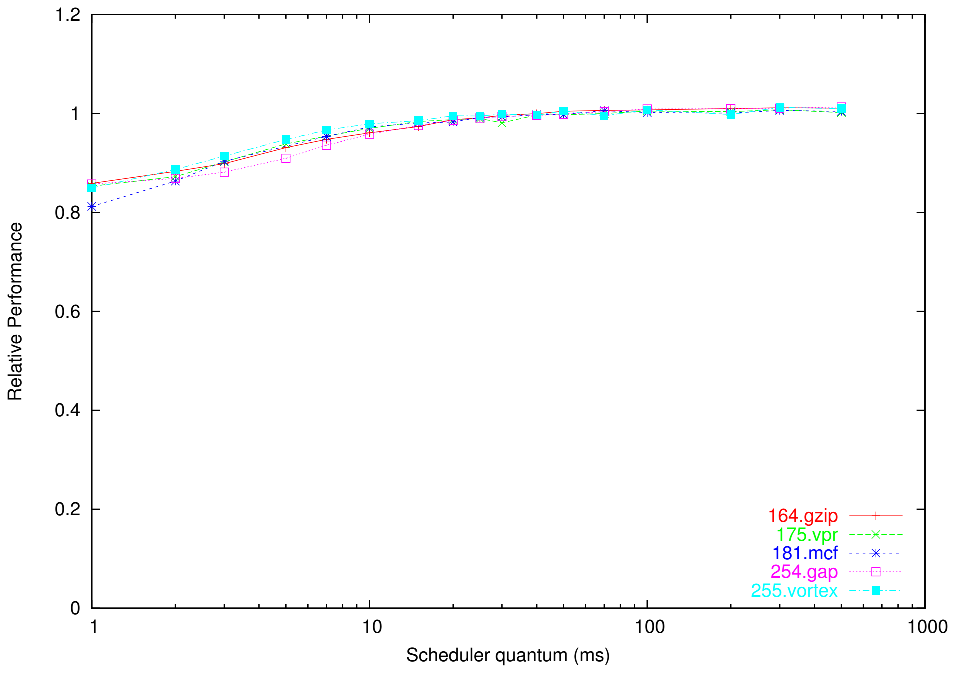
3.3.2.1 *Northwood Hyper-Threading*

Figure 3.2 shows the results for benchmark pairs on the Hyper-Threaded Pentium 4 using the same format as Tuck and Tullsen [Tuck03] to allow a direct comparison. For each subject benchmark a box and whisker plot shows the range of system speedups obtained when running the benchmark simultaneously with each other benchmark. The box shows the interquartile range (IQR) of these speedups with the median speedup shown by a line within the box. The whiskers extend to the most extreme speedup within 1.5 IQR of the 25th and 75th percentile (i.e. the edges of the box) respectively. Individual speedups outside of this range are shown as crosses. The gaps on the horizontal axis are where the Fortan-90 benchmarks would fit.

Tuck and Tullsen's experimental conditions differ from mine in a few ways, mainly the size of the L2 cache (my 512kB vs. 256kB), the speed of the memory (my 266MHz DDR vs. 800MHz DRDRAM) and the compiler (my GCC 2.96 vs. the Intel Reference Compiler). The similarities in the results given these differences show that the effect of the processor microarchitecture is important and that the lessons that can be learned can be applied to more than just the particular configuration under test.



(a) Subject benchmark: 164.gzip



(b) Subject benchmark: 175.vpr

Figure 3.1: The effect of quantum length on round-robin context switching of pairs of benchmarks. The data is for the subject benchmark running against 5 different background benchmarks.

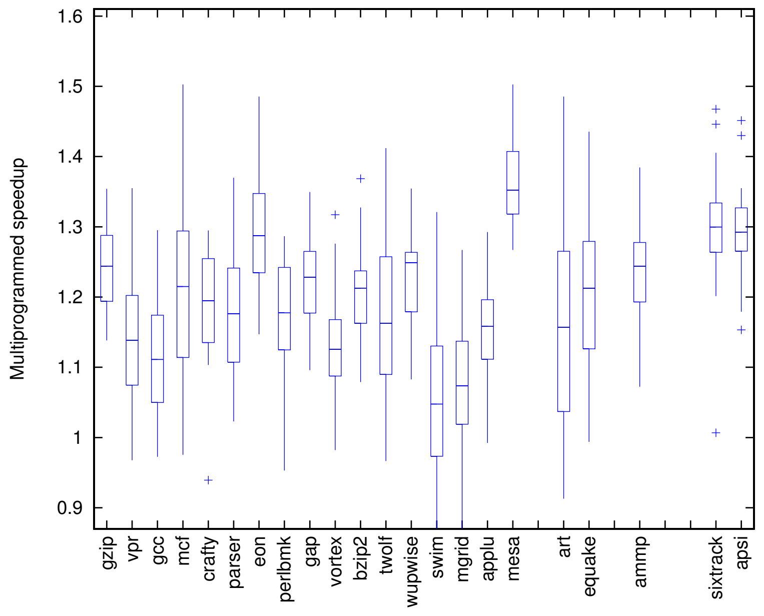


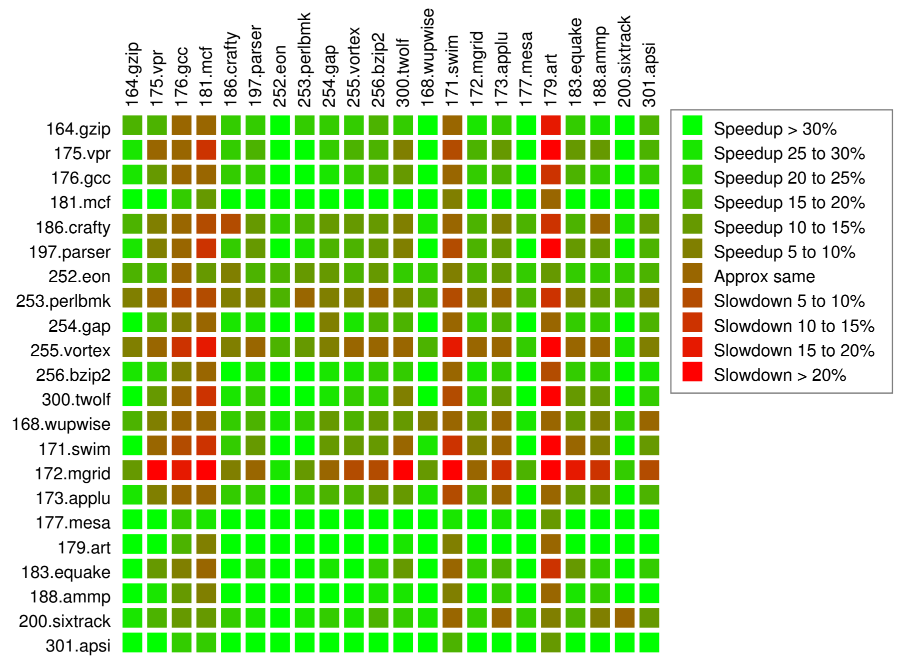
Figure 3.2: Multiprogrammed speedup of pairs of SPEC CPU2000 benchmarks running on a Hyper-Threaded processor.

I measure an average system speedup across all the benchmarks of 1.20 with best and worst case speedups of 1.50 (*mcf* vs. *mesa*) and 0.86 (*swim* vs. *mgrid*).

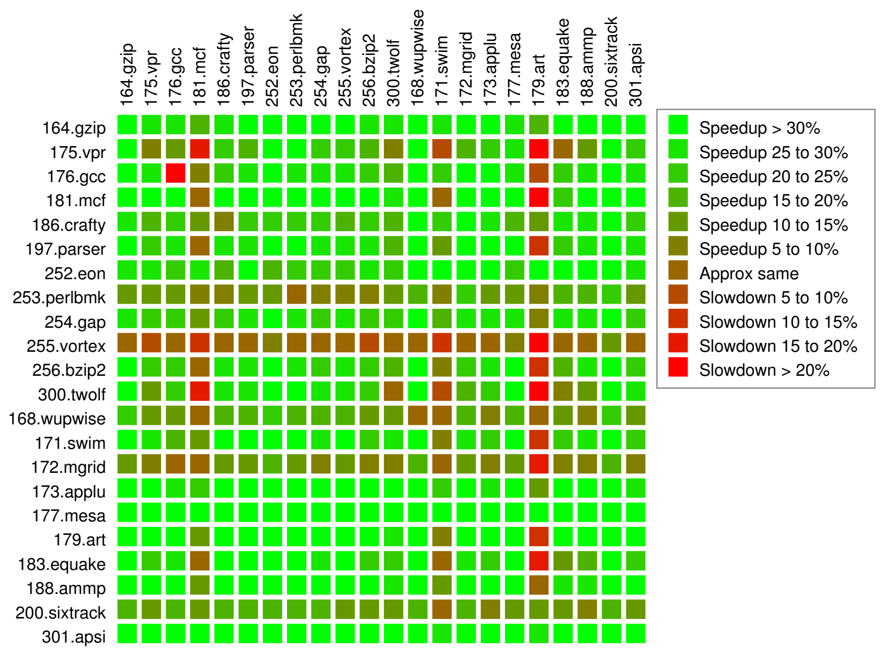
Figure 3.3(a) shows the individual performance ratio of each benchmark in a multiprogrammed pair. The figure is organised such that a square describes the PR of the row benchmark when sharing the processor with the column benchmark. The PR is considered bad when it is less than 0.5, i.e. worse than perfect context switching, and good when above 0.5. The colour of the square ranges from red for bad to green for good with a range of shades in-between. The first point to note is the lack of reflective symmetry about the top-left to bottom-right diagonal. In other words, when two benchmarks are simultaneously executing, the performance ratio of each individual benchmark (compared to it running alone) is different. This shows that the performance of pairs of simultaneously executing SPEC2000 benchmarks is not fairly shared. Inspection of the rows shows that benchmarks such as *mesa* and *apsi* always seem to do well regardless of what they simultaneously execute with. Benchmarks such as *mgrid* and *vortex* suffer when running against almost anything else. Looking at the columns suggests that benchmarks such as *sixtrack* and *mesa* rarely harm the benchmark they share the processor with while *swim*, *art* and *mcf* usually hurt the performance of the other benchmark.

The results show that a benchmark executing with another copy of itself (using a staggered start) usually has a lower than average performance ratio demonstrating the processor's preference for heterogeneous workloads which is not overcome by benefits in shared code and data. A subset of the homogeneous experiments was repeated without a staggered start; there was no change in the results.

The performance counter values recorded from the base runs of each benchmarks allow an insight into the observed behaviour:



(a) Northwood



(b) Prescott

Figure 3.3: Effect on each SPEC CPU2000 benchmark in a multiprogrammed pair running on a Hyper-Threaded processor. A green square represents a good performance ratio for the subject benchmark and a red square denotes a bad performance ratio. (A monochrome version of this figure is shown in Appendix A.)

mcf has a notably low IPC which can be attributed, at least in part, to its high L2 and L1-D miss rates. An explanation for why this benchmark rarely suffers when simultaneously executing with other benchmarks is that it is already performing so poorly that it is difficult to do much further damage (except with *art* and *swim* which have very high L2 miss rates). It might be expected that a benchmark simultaneously executing with *mcf* would itself perform well so long as it made relatively few cache accesses. *eon* and *mesa* fall into this category and the latter does perform well (28% speedup compared to sequential execution) but the former has only a moderate performance ratio (12% speedup) probably due its very high trace cache miss rate causing many accesses to the (already busy) L2 cache.

gzip is one of the benchmarks that generally does not detriment the performance of other benchmarks. It makes a large number of cache accesses and has a moderately high L1 D-cache miss rate of approximately 10%. It does however have small L2 cache and D-TLB miss rates due to its small memory footprint.

vortex suffers a reduced performance when running with most other benchmarks. It exhibits a moderately high number of I-TLB misses and a reasonable number of trace-cache misses although both figures are well below the highest of each metric. A more detailed examination of this benchmark's trace-cache miss rate shows that, although the mean miss rate is only moderate, there are regular spikes where the miss rate is very high. When running with other benchmarks the trace-cache miss rate of *vortex* (ignoring misses from the other thread) is about 30% higher than when it runs alone. This suggests that *vortex*'s instruction footprint is on the edge of the trace-cache capacity and is performance-limited by the hit rate in the trace-cache.

mcf, *swim* and *art* have high L1-D and L2 miss rates when running alone and have a low average IPC. They tend to cause a detriment to the performance of other benchmarks when simultaneously executing. *art* and *mcf* generally only suffer a performance loss themselves when the other benchmark also has a high L2 miss rate, *swim* suffers most when sharing with these benchmarks but is also more vulnerable to those with moderate miss rates.

mgrid is the benchmark that suffers the most when running under SMT whilst the simultaneously executing benchmark generally takes only a small performance hit. *mgrid* is notable in that it executes more loads per unit time than any other SPEC CPU2000 benchmark and has the highest L1 D-cache miss rate (per unit time). It has only a moderately high L2 miss rate and a low D-TLB miss rate. The only benchmarks that do not cause a performance loss to *mgrid* are those with low L2 miss rates (per unit time). *mgrid*'s baseline performance is good (an IPC of 1.44) given its high L1-D miss rate. The benchmark relies on a good L2 hit rate which makes it vulnerable to any simultaneously executing thread that pollutes the L2 cache.

sixtrack has a high baseline IPC (with a large part of that being floating point operations) and a low L1-D miss rate but a fairly high rate of issue of loads. The only benchmark it causes any significant performance degradation to is another copy of itself; this is most likely due to competition for floating point execution units. It suffers a moderate performance degradation when simultaneously running with benchmarks with moderate to high cache miss rates such as *art* and *swim*. The competitor benchmark will increase contention in the caches and harm *sixtrack*'s good cache hit rate. Tuck and Tullsen report that *sixtrack* suffers only minimal interference from *swim* and *art*. I believe the reason for this difference is that the larger L2 cache on my system

Best HT system throughput (1.50)	**181.mcf**	**177.mesa**
Int/FP	Int	FP
L1-D/L2 miss rates	high	low
D-TLB miss rate	high	low
Trace cache miss rate	low	high
IPC	very low	moderate
Worst HT system throughput (0.86)	**171.swim**	**172.mgrid**
Int/FP	FP	FP
L1-D miss rate	moderate	moderate
L2 miss rate	high	low
D-TLB miss rate	high	low
Trace cache miss rate	low	low
IPC	fairly low	fairly high
Stereotypical SMP vs HT performance	**183.equake**	**177.mesa**
Int/FP	FP	FP (less FP than equake)
L1-D/L2 miss rate	moderate	high
Trace cache miss rate	low	high
IPC	moderate	moderate

Table 3.4: Performance counter metrics for some interesting benchmark pairs. Metrics are for the benchmark running alone.

gives *sixtrack* a better baseline performance which makes it more vulnerable to performance degradation from benchmarks with high cache miss rates giving it lower relative speedups.

The best pairing observed in terms of system throughput was *mcf* vs. *mesa* (50% system speedup). Although *mcf* gets the better share of the performance gains, *mesa* does fairly well too. The performance counter metrics shown qualitatively in Table 3.4 for this pair show that heterogeneity is good.

Tuck and Tullsen note that *swim* appears in both the best and worse pairs. The reason for this is mainly down to the competitor. *mgrid* with its high L1-D miss rate is bad; *sixtrack* and *mesa* are good as they only have low L1-D miss rates so do little harm to the other thread.

The performance hit caused by L2 cache misses is not limited to interference with other threads which place non-trivial demands on the L2 cache. The above results show that L2 cache misses can hurt almost any other competitor. A particular inter-thread performance problem can occur on out-of-order (OoO) superscalar based SMT processors when long latency L2 misses occur. Tullsen and Brown [Tullsen01] and Cazorla *et al* [Cazorla03] note that this problem can be caused by the stalled thread consuming resources such as reorder buffer entries or physical registers, while it is performing no useful work.

A common aggressive OoO technique, used by the Pentium 4, is that the instruction schedulers assume all loads will hit in the L1 cache. If a load misses, dependent instructions may have already dispatched and will use incorrect data. Once the load has actually completed, the processor re-

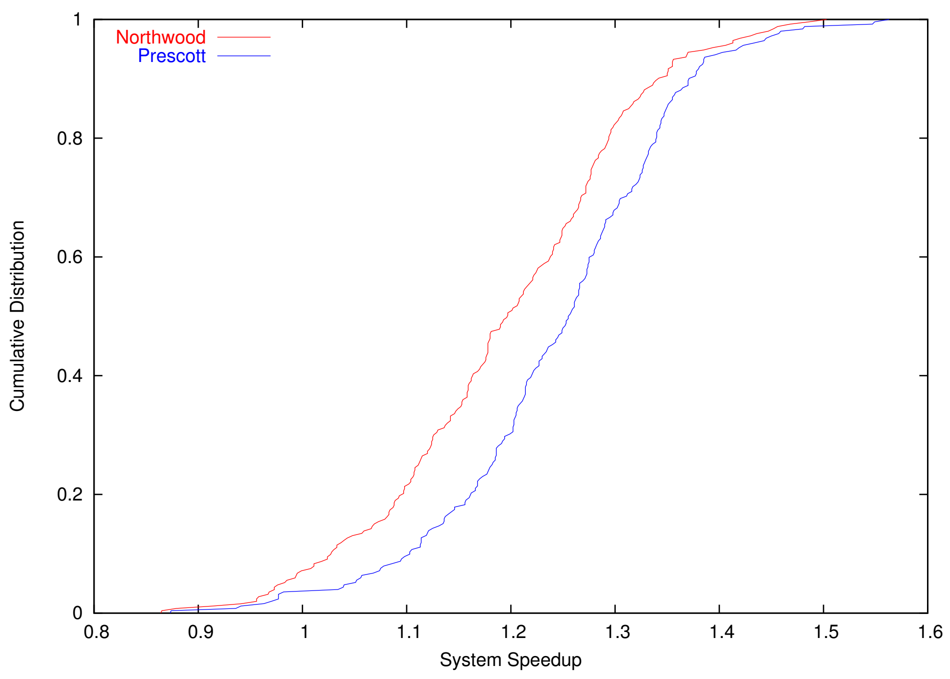


Figure 3.4: Distribution of system speedups for pairs of SPEC CPU2000 benchmarks running on a Hyper-Threaded processor. A point (x, y) shows that the fraction y of all of the pairs of benchmarks had system speedups of at most x.

executes the dependent instructions (this is called "replay" for the Pentium 4). For a single-thread superscalar this data-speculation mechanism works well but for SMT the extra resources used by the replayed instructions will reduce the resources available to the other thread(s).

3.3.2.2 *Prescott Hyper-Threading*

Figure 3.3(b) shows the performance ratio matrix for the same set of experiments performed on a machine using the Prescott Pentium 4 processor core. Note that this machine has a faster clock, larger L2 cache and newer operating system distribution than the Northwood-based machine used for the above experiments. Therefore direct, absolute comparisons should not be made; it is the trends that are important. The immediately clear difference from the Northwood results is the reduction in red and brown squares which denote a suffering benchmark. However, the same general pattern is evident, albeit less pronounced. The mean system speedup is 1.25 compared to 1.20 for Northwood. The standard deviation of the performance ratios of the individual benchmarks is 0.089 for Prescott compared to 0.097 for Northwood. These results suggest that the microarchitecture of the Prescott core provides a fairer sharing of processor resources than Northwood.

Figures 3.4 and 3.5 show respectively the distributions of system speedups and individual benchmark performance ratios for both Prescott and Northwood test platforms. As demonstrated above the Prescott generally performs better than the Northwood. The interquartile range for system speedups is 0.167 (1.11 to 1.28) for Northwood and 0.145 (1.18 to 1.33) for Prescott and for the performance ratios 0.125 (0.535 to 0.660) and 0.111 (0.569 to 0.681) respectively.

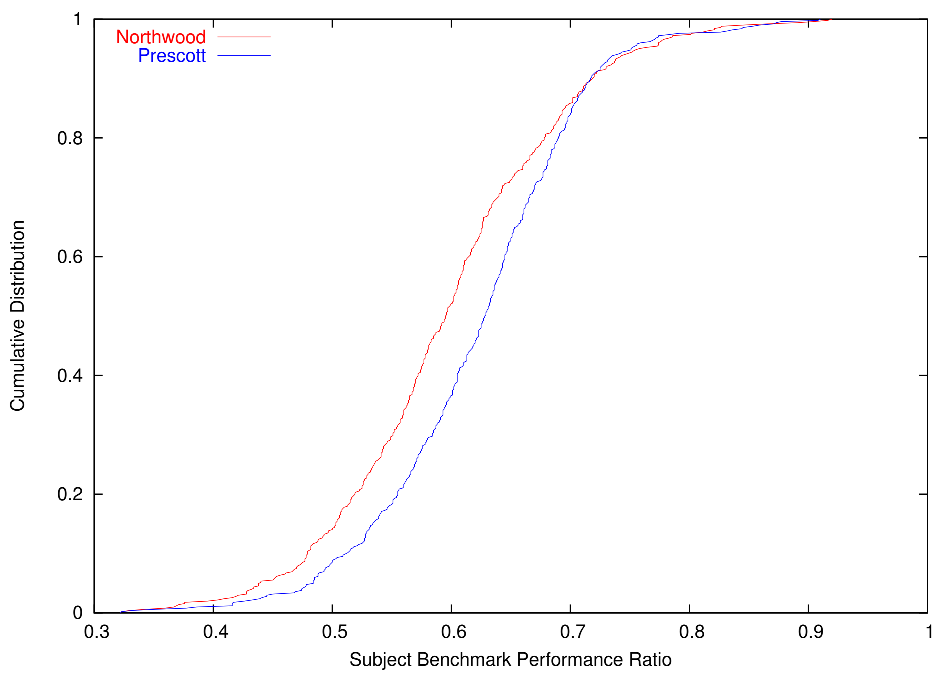


Figure 3.5: Distribution of performances for individual SPEC CPU2000 benchmarks running simultaneously under Hyper-Threading. A point (x, y) shows that the fraction y of all of the subject benchmarks in the cross-product had performance ratios of at most x.

The problem of pathological cases has not been eliminated in the Prescott core. Some notable observation of the data are:

- The experiments where two copies of the same benchmark compete show a greater range of behaviour with Prescott than with Northwood; with Prescott there are fewer homogeneous pairs with small speedups but more with larger speedups or slowdowns. The mean system speedup for these pairs was 1.08 compared to 1.09 with Northwood. Homogeneous demands for resource is an inherent problem with dynamically shared systems like SMT so is hard to eliminate.

- *255.vortex* still exhibits no, or a small negative, speedup in most cases. Since the Prescott's trace-cache is the same size and configuration as that of the Northwood, *vortex* suffers the same capacity problem as on the older core.

- *179.art* still detriments the performance of many other benchmarks, some considerably, but its effect is less pronounced on Prescott than Northwood. *181.mcf* shows a similar change but in two cases it slows the subject benchmark more with Prescott than with Northwood (*186.crafty* and *183.equake*).

3.3.2.3 *Northwood Hyper-Threading vs. SMP*

Figure 3.6 shows the speedups for the benchmark pairs running in a traditional SMP configuration. Also shown for comparison is the Hyper-Threading data as shown above. An interesting observation is that benchmarks that have a large variation in performance under Hyper-Threading also have a large variation under SMP. It might be imagined that the performance of a given

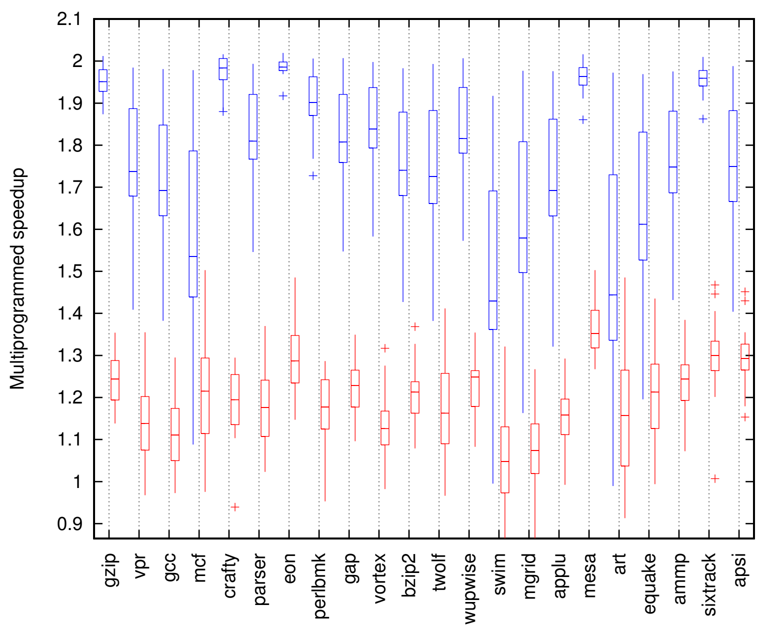


Figure 3.6: Multiprogrammed speedup of pairs of SPEC2000 benchmarks running on a Northwood Hyper-Threaded processor and non-Hyper-Threaded SMP. The right (red) of each pair of box and whiskers is Hyper-Threading and the left (blue) is SMP.

benchmark would be more stable under SMP than under Hyper-Threading since there is much less interaction between the two processes. The correspondence in variation suggest that competition for off-chip resources such as the memory bus are as important as on-chip interaction.

The mean speedup for all pairs was 1.20 under Hyper-Threading and 1.77 under SMP. This means that the performance of an SMP system is 48% better than a corresponding Hyper-Threading system for SPEC CPU2000.

Figure 3.7 shows the individual performance ratios of the benchmarks in the same style as Figure 3.3 described in Section 3.3.2.1 above. Unlike Hyper-Threading, SMP does not show any notable unfairness between the concurrently executing threads. This is clearly due to the vast reduction in resource sharing with the main remaining resource being the memory and its bus and controller. This means that the benchmarks that reduce the performance of the other running benchmarks are also the ones that suffer themselves. The benchmarks in this category include *mcf*, *swim*, *mgrid*, *art* and *equake*: all ones that exhibit a high L2 miss rate which further identifies the memory bus as the point of contention.

An example of expected behaviour is *equake* vs. *mesa*. This pair exhibits a system speedup of just under 1 for context switching on a single processor, just under 2 for traditional SMP and a figure in the middle, 1.42, for Hyper-Threading. As *mesa* has a low cache miss rate it does not make much use of the memory bus so it not slowed by *equake*'s high L2 miss rate when running under SMP. Similarly for round robin context switching the small data footprint of *mesa* does not cause any significant eviction of data belonging to *equake*. Under Hyper-Threading there is little contention for the caches and the smaller fraction of floating-point instructions in *mesa* means

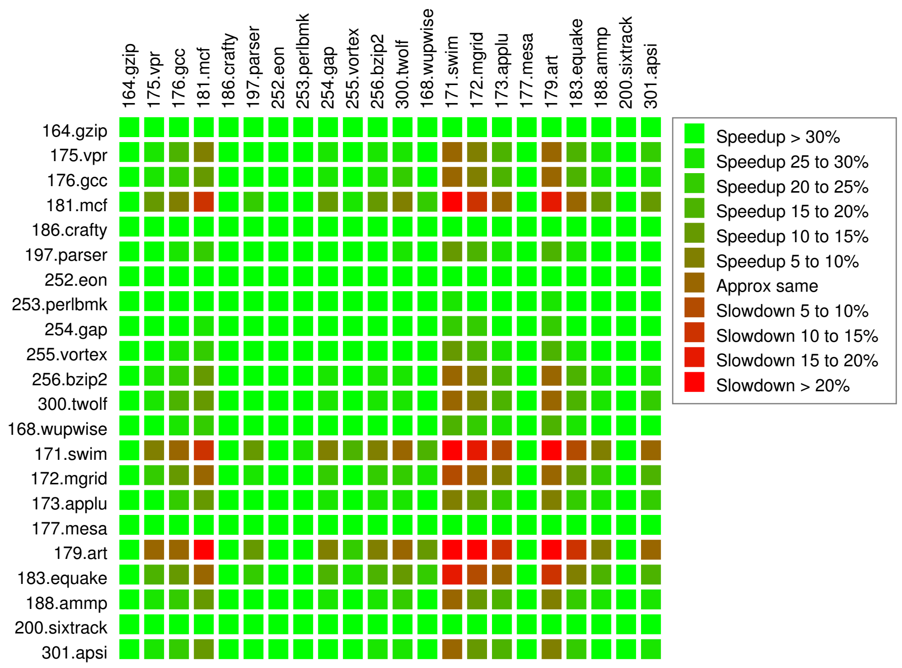


Figure 3.7: Effect on each SPEC CPU2000 benchmark in a multiprogrammed pair running on an SMP configuration. A green square represents a good performance ratio for the subject benchmark and a red square denotes a bad performance ratio (relative to "perfect" SMP). (A monochrome version of this figure is shown in Appendix A.)

that the workloads are heterogeneous and therefore can better utilise the processor's execution units.

art and *mcf* perform similarly under SMP, Hyper-Threading and round robin context switching. This is almost certainly due to the very high L1 and L2 cache miss rates and the corresponding low IPC they both achieve.

When executing under Hyper-Threading, *art* does better to the detriment of *mgrid* however under SMP the roles are reversed. Both have a high L1 miss rate but *art*'s is the highest of the pair. *art* has a high, and *mgrid* a fairly low L2 miss rate. Under Hyper-Threading *art* benefits most from the latency hiding offered by Hyper-Threading and causes harm to *mgrid* by polluting the L1-D cache. Under SMP there is no L1 interference so the *mgrid* outperforms *art* due to the lower L2 miss rate of the former.

When running against another copy of itself *vortex* has virtually no speedup running under Hyper-Threading compared to context switching. Under SMP there is almost no penalty which is due to the fairly low memory bus utilisation. *vortex* is punished under Hyper-Threading due to competition for trace-cache and I-TLB resource - this is not a factor in its performance under SMP.

vortex and *mcf* running under SMP take a notable (20% and 15% respectively) performance hit compared to running alone. This is due to moderate L2 miss rates causing increased bus utilisation. Performance under Hyper-Threading shows *vortex* suffering a large performance loss (20% lower than if it only had half the CPU time) while *mcf* does particularly well. The latter

Config	Physical Processor 0		Physical Processor 1	
	Hyper-Thread 0	Hyper-Thread 1	Hyper-Thread 0	Hyper-Thread 1
UP	Active	Disabled	Disabled	Disabled
SMP	Active	Disabled	Active	Disabled
1x2	Active	Active	Disabled	Disabled
2x2	Active	Active	Active	Active

Table 3.5: Configurations used for Linux kernel compilation experiments.

has a low IPC due to its high cache miss rates so benefits from latency hiding. *vortex* has a fairly low L1-D miss rate which is harmed by the competing thread.

gzip with its very low L2 and trace cache miss rates, moderate L1-D miss rate and large number of memory accesses always does well under SMP due to the lack of bus contention but has a moderate and mixed performance under Hyper-Threading. *gzip* is vulnerable under Hyper-Threading due to its high IPC and low L2 miss rate meaning it is already making very good use of the processor's resources. Any other thread will take away resource and slow *gzip* down.

3.3.3 Desktop Applications

The desktop application benchmarks described in Section 3.2.3.2 were used in a series of experiments on the Northwood processor.

These benchmarks, particularly *compile*, involve a lot more disk activity than the SPEC CPU2000 benchmarks. This means that there will be periods when IO requests are being waited on and the logical processor is idle. During these idle periods the other logical processor will be able to make use of all of the processor resources. This is an important realistic system effect that was not present in the above SPEC measurements.

The *compile* benchmark (Linux kernel build) was first tested to measure the effect of using parallel build processes. The `make` system managing the compilation can be instructed to aim for a specified number of parallel tasks - the `-j` option. Parallel builds of up to 8-way were performed on the configurations of logical and physical processors shown in Table 3.5. The results are shown in Figure 3.8. The results show that there is little to be gained from parallel builds on the uniprocessor *SMP*. This contradicts a common belief that a 3-way build of a Linux kernel on a uniprocessor is usually quicker than a 1-way build. With large memory sizes (and therefore large buffer-caches) and good disk prefetching, the disk latency hiding offered by the parallel build is not required

The two physical processor *SMP* shows a much reduced build time when a 2-way build is performed. This is the expected behaviour as the two build processes can make use of the doubled processing resource available. The single package, two Hyper-Threaded *1x2* shows a similar but much less pronounced behaviour with the reduction in build time of a 2-way over a 1-way build being about 10%. The 1-way build on the Hyper-Threaded configuration is itself about 7% quicker than a 1-way build on the same processor with Hyper-Threading disabled. This is due to the limited amount of parallelism available in the 1-way build (pipelined compilation) as well

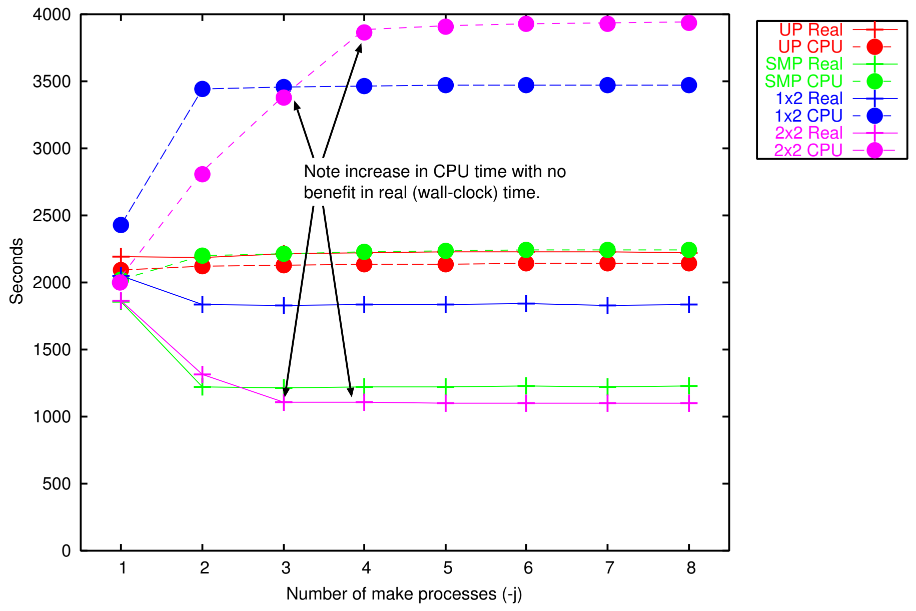


Figure 3.8: Build times ("Real" wall-clock time and CPU time) for two Linux kernels: comparing different parallel make options. CPU time is the integral of the number of logical processors active over time; in cases other than *UP* it can therefore be greater than the real time.

Metric	Number of make processes (-j)					
	1	2	3	4	5	6
Bus activity events/sec	14.2M	25.2M	33.2M	36.7M	37.1M	37.1M
L2 misses/sec	3.18M	5.72M	7.59M	8.41M	8.49M	8.50M
IPC	0.438	0.623	0.733	0.740	0.742	0.740

Table 3.6: Performance counter data for *2x2* builds of the Linux 2.6.7 kernel. Values are summed across all logical and physical processors.

as the Hyper-Threaded configuration's ability to simultaneously execute support and background tasks such as interrupt handlers and kernel threads. The two package, each of two logical processors, configuration *2x2*, shows the build time decreasing further as the parallelism is increased beyond 2. The reduction from 3-way to 4-way is only minor but the increase in CPU time shows that all four logical processors are being used. The system wide IPC was 0.733 for the 3-way build and 0.740 for the 4-way.

Processor performance counters were used to investigate the reason for this lack of gain. Bus activity, level 2 cache misses, and retired instructions were counted and the aggregated results are shown in Table 3.6. The bus activity appears to saturate at 37.1 million events per second, a utilisation of approximately 9%. Tests with `hdparm` shows that the maximum buffer-cache bandwidth that this machine-kernel combination is able to support is around 310MB/s which gives a bus activity of 37.7 million events per second. This suggests that buffer-cache bandwidth is the limiting factor for the kernel builds on 3 or 4 logical processors.

Both Hyper-Threaded configurations show a large increase in CPU time while the the wall-clock

		mp3	compile	glx	image
HT	**mp3**	0.605	0.681	0.471	0.829
	compile	0.721	0.579	0.436	0.667
	glx	0.845	0.743	N/A	0.773
	image	0.628	0.565	0.386	0.554
SMP	**mp3**	0.995	0.987	0.981	0.990
	compile	0.995	0.884	0.832	0.971
	glx	0.995	0.980	N/A	0.982
	image	0.994	0.980	0.946	0.986

Table 3.7: Performance ratios of the "desktop" applications running in multiprogrammed pairs on a Northwood Hyper-Threaded processor. A figure denotes the performance of the row benchmark while running with the column benchmark. The performance ratio metric is the same as that used for the SPEC experiments above where a ratio of 0.5 represents zero-cost context switching.

time decreases. This is due to the mutual slowing effect the threads have on each other. CPU time is a figure that has to be treated carefully in SMT systems because the amount of processing that can be done per unit time will depend upon the degree of sharing and contention between the threads.

For the purpose of the thread interaction experiments a single logical processor was used for each workload. Therefore the *UP* results are of interest. There is a negligible difference in performance between a 1- and 2-way build so a 1-way build was used for the further experiments.

A cross-product set of experiments was performed in a similar manner to the SPEC CPU2000 experiments above. The only exception was that *glx* could not be run with a second instance of the same benchmark because there is only one screen with one X server process. The results for pairs executed under Hyper-Threading and SMP configurations are shown in Table 3.7 using the performance ratio described in Section 3.2.4. The mean system speedup for the pairs was 1.33; this is considerably more than the 1.20 seen above which is due to the less compute-bound nature of these desktop applications.

The pairs involving *glx* run under Hyper-Threading show that the graphics benchmark takes more of the processor resource than the competing application. *mp3* coexists well with applications other than *glx*. It places a low demand on the cache hierarchy and its compute demand can easily take advantage of processor resources while its more IO-bound competitor is stalled. *compile* performs a large amount of file IO; its processor was idle for 3% of its execution time when running alone or with *mp3*, and 7% when running against another instance of itself. The benchmark's reduced performance when running against another instance of itself under SMP shows that the contention is partly system-wide, such as bus activity or disk IO/buffer cache contention.

3.3.4 Summary

I have measured the mutual effect of processes simultaneously executing on the Intel Pentium 4 processor with Hyper-Threading. The results show speedups for individual benchmarks of up to 30 to 40% (with a high variance) compared to sequential execution. I have expanded on these results to consider the bias between the simultaneously executing processes and shown that some pairings can exhibit a performance bias of up to 70:30. Using performance counters I have shown that many results can be explained by considering cache miss rates and resource requirement heterogeneity in general. Examination of performance counter data has shown that threads with high cache miss rates can have a detrimental effect on simultaneously executing threads. Those with high L1 miss rates tend to benefit from the latency hiding provided by Hyper-Threading. In the next section I investigate how the behaviour changes during the course of the programs' execution and whether a more formal correlation of performance counter data to observed performance exists.

3.4 Phases of Execution

The experiments described above illustrate a wide range of behaviour when independent processes are running on the two logical processors of a Hyper-Threaded processor. The results describe the behaviour of complete runs of each program. Most programs exhibit changing behaviour during their execution, often in a number of distinct phases [Sherwood99]. In order to see how the mutual effect of concurrently running processes changes as the programs move through different phases the following analysis was performed.

A subset of the SPEC CPU2000 benchmarks was used. The benchmarks were rank ordered based on their mean system speedup observed in the earlier experiments. Eight benchmarks were reasonably uniformly selected with care to maintain a balance of integer and floating point benchmarks. The benchmarks used were: *164.gzip*, *181.mcf*, *186.crafty* and *255.vortex* (integer); and *171.swim*, *177.mesa*, *179.art* and *200.sixtrack* (floating point).

Each benchmark was run alone on the first logical processor of the second physical package. The other logical processor of that package was idle. The Linux interrupt processor affinity facility (`/proc/irq/*/smp_affinity`) was used to force interrupts (except timer and inter-processor interrupts) to be handled by one physical processor while experiments were performed on the other.

During each run a set of performance counter values, including instructions retired on each logical processor, was collected at intervals of approximately 100ms. The performance counters were configured to count events from both user and kernel execution modes to allow system call activity of the benchmarks to be included. This configuration also causes the remaining interrupt activity to be counted. The activity is mainly the scheduler responding to timer interrupts and inter-processor interrupts for reading the performance counters; the effect of both are small and constant.

Interpolation of the performance counter samples was used to find the start and end retired instruction counts for each benchmark run (taking care to deal with counter overflow). The performance counter data was then split into 100 windows per benchmark run based on an equal number of retired instructions per window. The cycle counter and the remaining performance counters were interpolated from the samples to obtain the increment for each in each window.

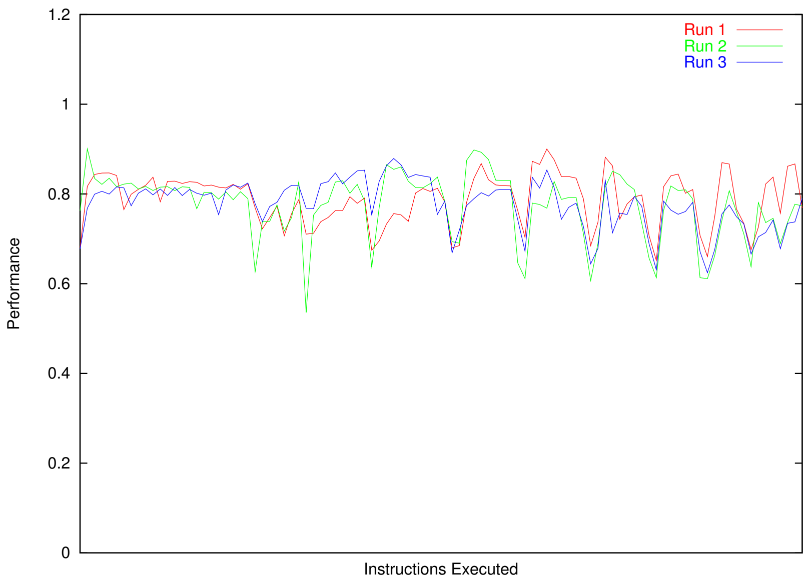


Figure 3.9: Performance ratio (relative to running alone on the processor) of 3 runs of *181.mcf* while competing with *164.gzip*.

The same pairwise simultaneous execution of benchmarks as performed in Section 3.3 was repeated, the only difference here being the use of the high frequency performance counter sampling. The 100 window data processing described above was performed on each benchmark of each pair. Because instructions rather than cycles were used to split the execution, the windows from a simultaneous run of a given benchmark correspond to the windows from its base run. There will be differences in the number of instructions retired because of nondeterministic effects and differing interrupt activity however the splitting into a fixed number of windows rather than a fixed window size minimises any error. It is possible to see the differing slowdowns for each window. For example, window *n* in a hypothetical benchmark's base run may have taken 355,000,000 cycles to execute; the same window from that benchmark while it ran simultaneously with a second benchmark may have taken 580,000,000 cycles - the result is a slowdown of 39% for this benchmark for this window (or a *performance ratio* of 0.61 using the metrics of Section 3.2.4). Figure 3.9 shows an example of how the performance ratio of a benchmark can change over time. The example is *181.mcf* running simultaneously with *164.gzip*; each plot is a different run with a different offset from the phase of the competitor's execution. Note that this graph says nothing about the competitor's own performance.

The analysis was originally performed using 1000 windows but a number of co-located peaks and troughs were seen with peaks showing a performance under Hyper-Threading of more than twice that when running alone. The co-located troughs suggest that the small localised changes in performance, such as interrupt handling and differing delays in I/O operations, were not being "averaged out". The analysis was repeated with a fresh base run of the subject benchmark. There were some correlation of the peak-trough pairs however it was clear that transient behaviour was being observed. The number of windows was reduced to 100 to try to smooth these variations. The result was much better with the traces based on the two different base runs showing a high correlation. The smaller number of windows is still sufficient to show the changing behaviour of the benchmark.

Figures 3.10 to 3.17 show the performance ratio (compared to running alone on the processor) of each of the subject benchmarks. Runs against different background benchmarks are shown in different colours. There are three runs with each background benchmark, each with a different offset in the phases of the two benchmarks. These graphs tell two stories: the effect of the subject benchmark on its own SMT performance, and the effect of the background benchmark on the SMT performance of the subject. The **shape** of the line is largely influenced by the subject benchmark while the **amplitude** (or vertical shift) is largely influenced by the background. It can be seen that when the performance ratio is generally good the shapes of the plots are largely determined by the subject benchmarks but as the effect of the background becomes greater and the subject performance ratio drops the definition of the shape is reduced.

For any given benchmark pair the graphs show three lines for successive runs of the subject benchmark. The background benchmark was running in a continuous loop in the background so will be out of phase with the subject. It is therefore notable that the three lines in most cases match so closely. In such cases the phase of the background benchmark is having little effect on the subject benchmark performance ratio. The choice of background does appear to decide on the average performance of the subject, shown as the vertical placement of the plots. The exceptions to this behaviour are generally when both benchmarks in a pair exhibit a high level of variation in their phases. Taking *164.gzip* and *255.vortex* as examples: both have definite phases and show less correlation on the subject lines when they compete against each other.

255.vortex exhibits a a periodic spiky pattern. Note that this pattern is evident in all the traces but is inverted for the traces with a lower over-all performance (when competing with *mcf*, *art* and *swim*). The reason for this is that *vortex* has a moderate L2 miss rate which follows the periodic pattern; while competing against benchmarks with low L2 miss rates the periods where *vortex* itself suffers a high L2 miss rate result in it having a low native IPC. During periods of low L2 miss rate its IPC is better. The result is that a benchmark with a low L2 miss rate, and therefore with a generally good IPC, can do more damage to *vortex* during the latter's periods of low L2 miss rate. When the competitor places a high demand on the L2 cache the miss rate of *vortex* is magnified and results in a slowdown. Since these competitors will have fairly low IPCs they do little harm to vortex during its low L2 miss rate periods.

The abrupt drops in performance ratio seen independently in the plots for *179.art* and *186.crafty* when competing against each other are coincident and occurred during the first of the three timed runs of the *art*. After this run of *art* completed the performance ratio of *crafty* and of subsequent runs of *art* returned to the higher level (0.4 to 0.5 for *crafty* and 0.8 to 0.9 for *art*); this appears to be a rare occurrence. Inspection of Figure 3.16 shows that the position of the abrupt drop with *art*'s execution is coincident with lesser, but repeatable, drops in the benchmark's performance ratio when executing with *255.vortex*.

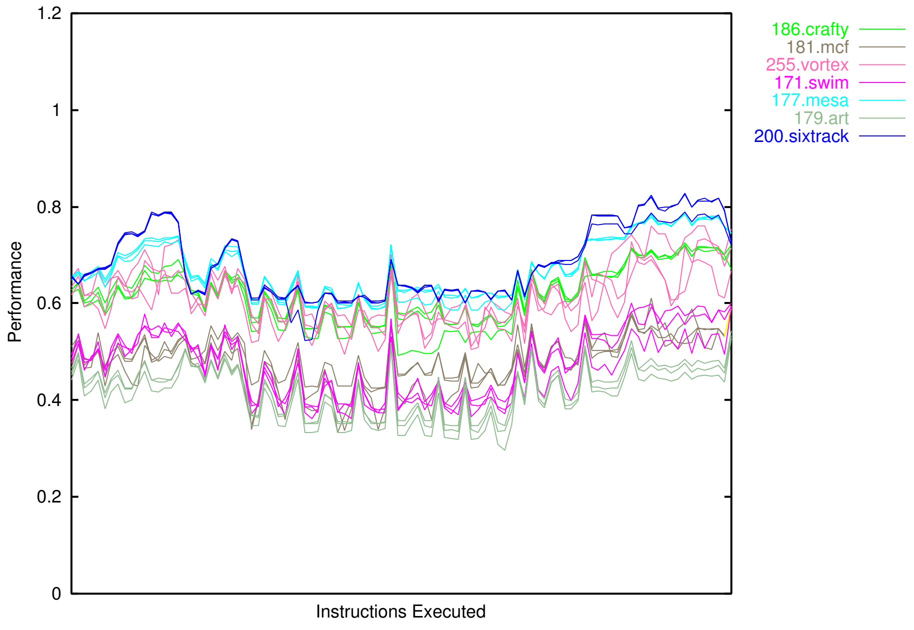


Figure 3.10: Performance ratio of *164.gzip* while running simultaneously with other benchmarks.

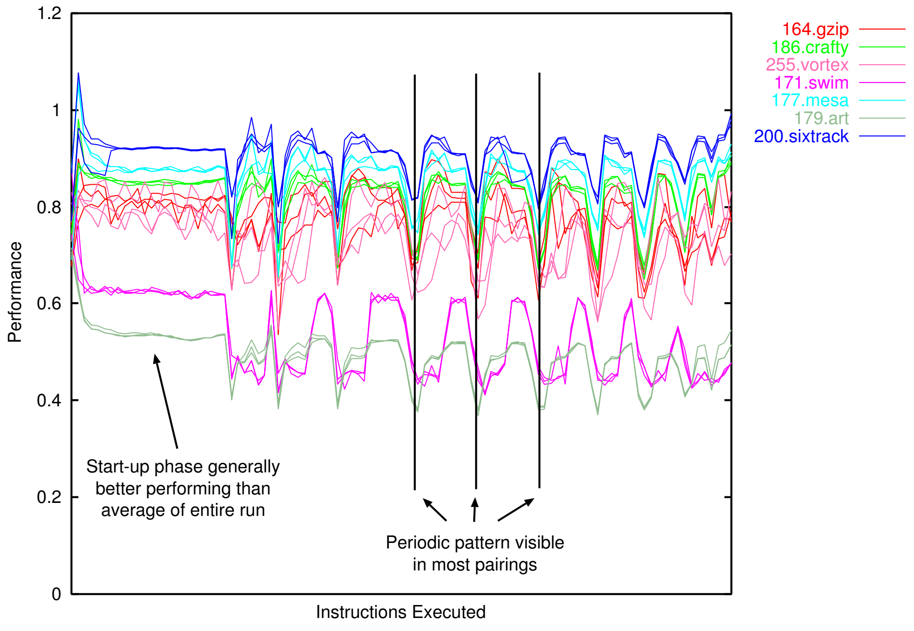


Figure 3.11: Performance ratio of *181.mcf* while running simultaneously with other benchmarks.

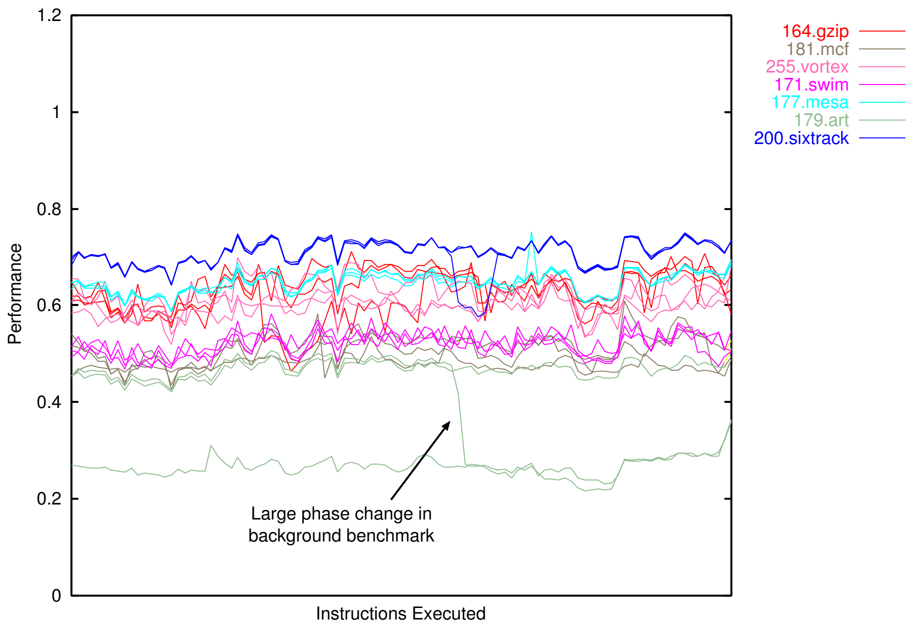


Figure 3.12: Performance ratio of *186.crafty* while running simultaneously with other benchmarks.

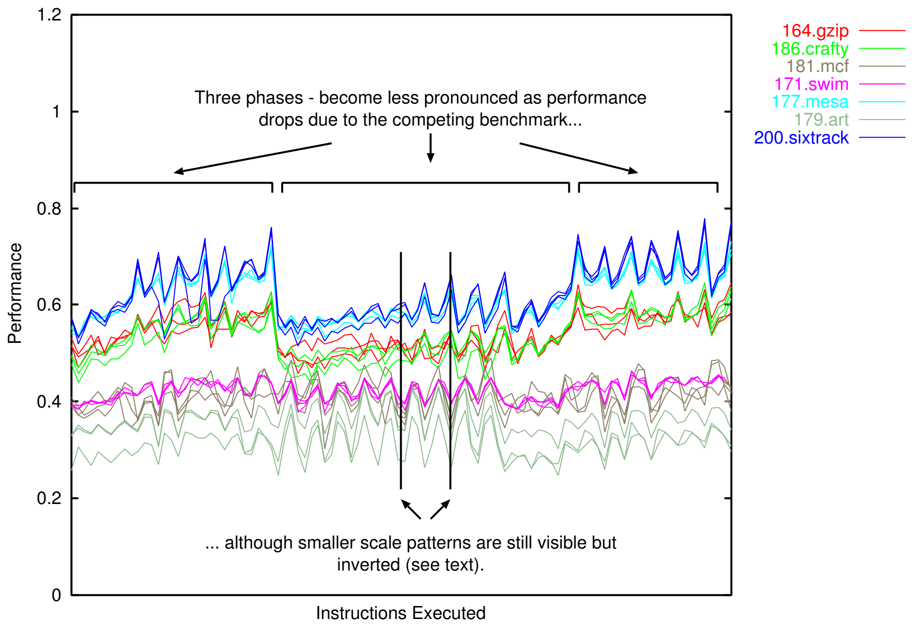


Figure 3.13: Performance ratio of *255.vortex* while running simultaneously with other benchmarks.

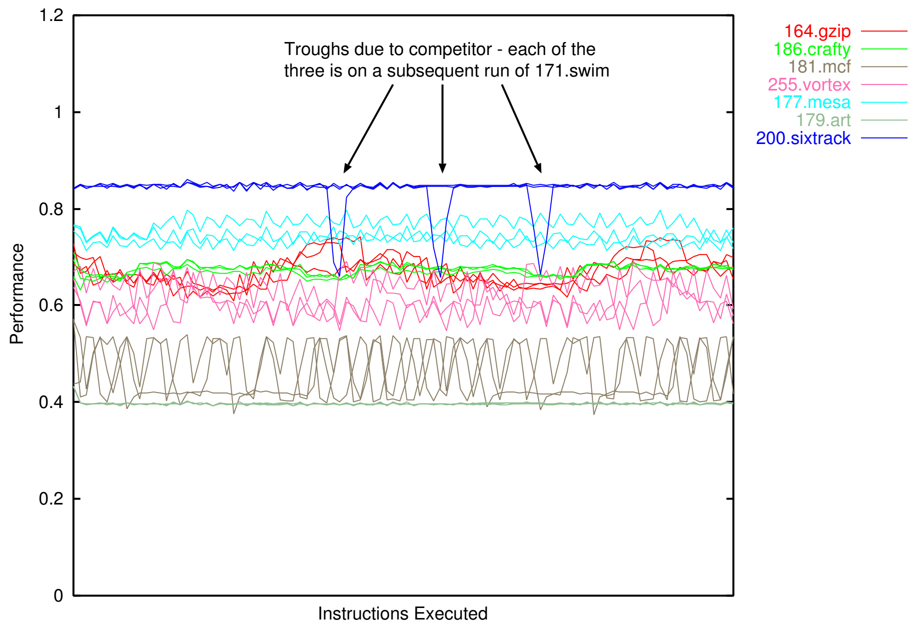


Figure 3.14: Performance ratio of *171.swim* while running simultaneously with other benchmarks.

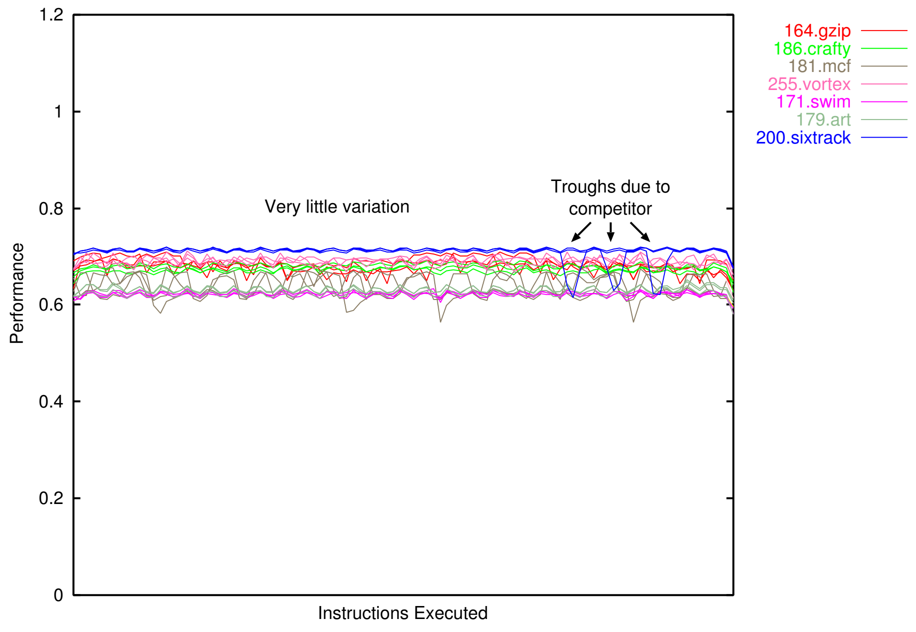


Figure 3.15: Performance ratio of *177.mesa* while running simultaneously with other benchmarks.

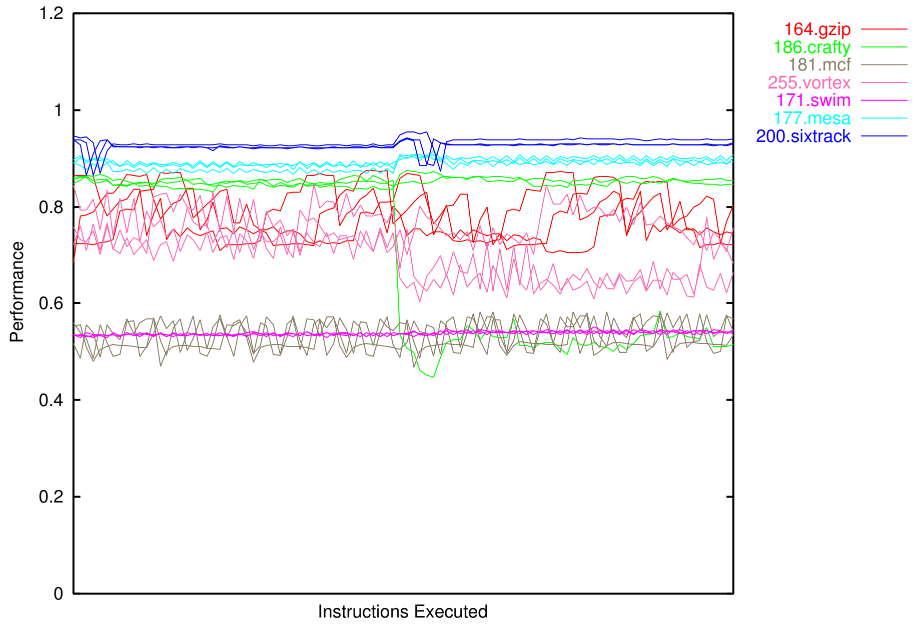


Figure 3.16: Performance ratio of *179.art* while running simultaneously with other benchmarks.

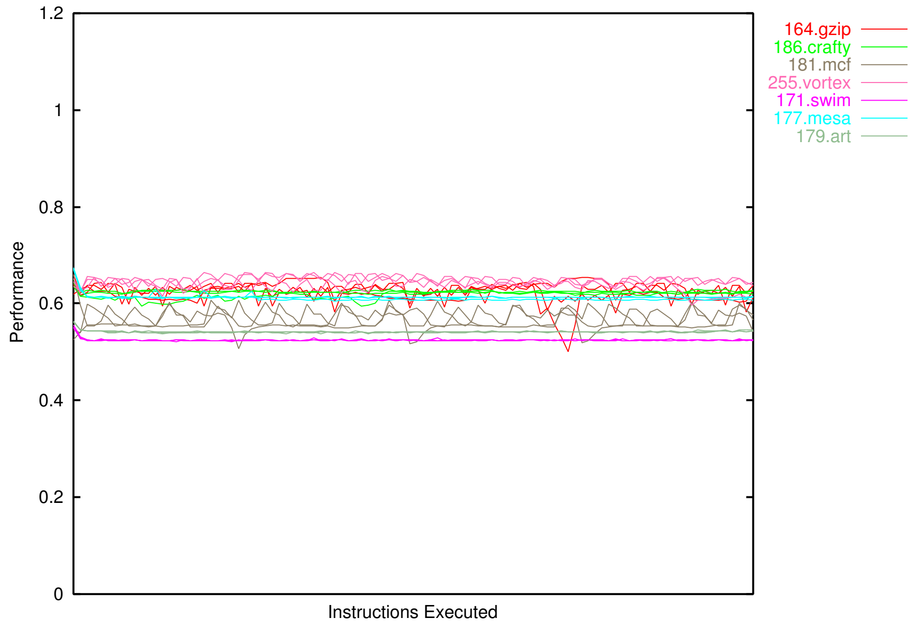


Figure 3.17: Performance ratio of *200.sixtrack* while running simultaneously with other benchmarks.

3.4.1 Performance Counter Correlation

The data collected in these experiments included performance counter increments for each window for level 1 (L1), level 2 (L2) and trace-cache (TC) misses, floating point (FP) operations and the total number of retired instructions (Insts). These metrics were counted independently for each logical processor. For each subject benchmark there were 3 runs with each of 7 background benchmarks giving a total of 16,800 samples. In order to assess whether there was any correlation between the performance counter increments and the subject benchmark performance ratio a multiple linear regression analysis was performed with the performance counter increments per cycle as the explanatory variables and the subject performance ratio as the dependent variable. The coefficient of determination (the R^2 value), an indication of how much of the variability of the dependent variable can be explained by the explanatory variables, was 66.5%. The coefficients for the explanatory variables are shown in Table 3.8 along with the mean number of counted events per cycle for each variable (shown to put the magnitudes of the variables into context). The fourth column of the table indicates the importance of each counter in the model by multiplying the standard deviation of that metric by the coefficient.

Counter	**Coefficient**	**Mean Events/1000 Cycles**	**Importance**
	(to 3 S.F.)	(to 3 S.F.)	(coeff x st.dev.)
(Constant)	0.4010		
TC-subj	29.7000	0.554	26.2
L2-subj	55.7000	1.440	87.2
FP-subj	0.3520	52.0	29.8
Insts-subj	-0.0220	258	-4.3
L1-subj	2.1900	10.7	15.4
TC-back	32.7000	0.561	29.0
L2-back	1.5200	1.43	2.3
FP-back	-0.4180	52.6	-35.3
Insts-back	0.5060	256	99.7
L1-back	-3.5400	10.6	-25.3

Table 3.8: Multiple linear regression coefficients for estimating the performance ratio of the subject benchmark. The performance counters for the logical processor executing the subject benchmark are suffixed “subj” and those for the background benchmark’s logical processor, “back”.

The most significant predictor is the IPC of the background task followed by the L2 miss rate of the subject benchmark. The negative coefficient for the L1 misses of the background task show that a high miss rate will reduce the performance ratio of the subject benchmark. A higher L1 miss rate for the subject benchmark itself will generally mean it has a better performance ratio. This apparent discrepancy is due to subject benchmarks having native high L1 miss rates will lose little when running under Hyper-Threading. The statistical significance of each variable can be tested by calculating the *p-value* for that variable. For a 95% confidence, any p-value greater than 0.05 will mean that the corresponding variable is insignificant. The L2 miss rate of the background process had a p-value of 0.100 so is insignificant (in practical terms this metric is covered largely by the IPC of the background process). The IPC of the subject process had a

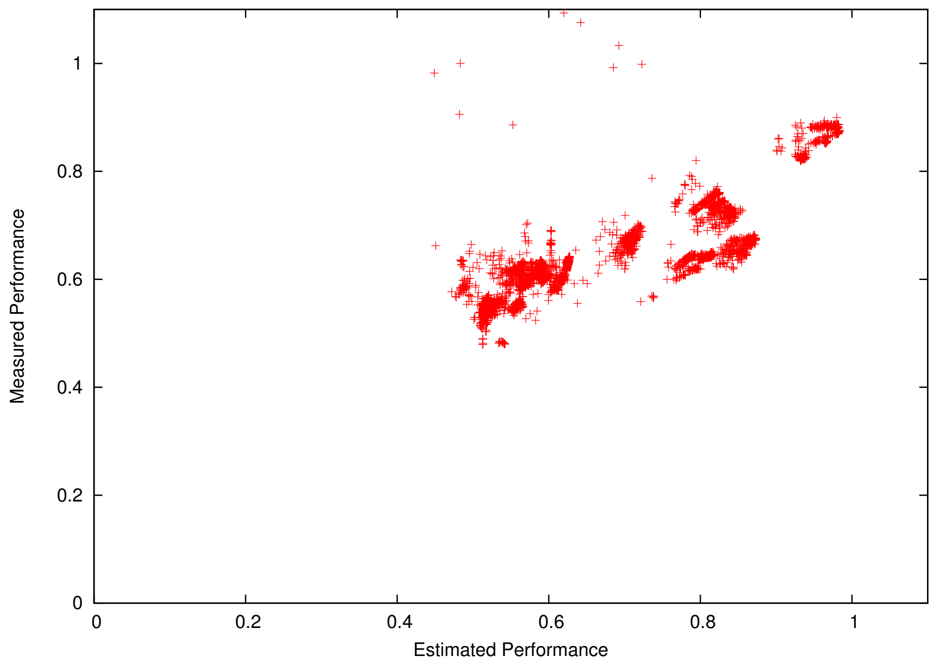


Figure 3.18: Testing the MLR estimation model on pairs from *175.vpr*, *252.eon*, *168.wupwise* and *183.equake*.

p-value of 0.0369 and has a small weighting on the final result. All other variables had negligible p-values.

The performance ratio of a benchmark can only be calculated directly when its standalone execution time is known; in these experiments each benchmark was run both standalone and in simultaneous execution competition. In a live system it is not possible to know what the execution time would have been for a given chunk of an application's execution. It would be useful to be able to read performance counters and estimate the current performance ratio. The model from the regression analysis above was tested on a different set of benchmarks to see how accurately it could estimate performance. The same experiments were performed using *175.vpr*, *252.eon*, *168.wupwise* and *183.equake*. The coefficient of correlation between the estimated and measured values was 0.853. Figure 3.18 shows a scatter plot of estimated and measured performance ratio figures for the each of the 100 windows for each pair.

3.5 Asymmetry

In this section I describe experiments performed to assess the importance of the difference between physical and logical processors when making scheduling decisions. I show that groups of four benchmarks scheduled to four logical processors belonging to two physical processors can exhibit a variety of system-wide performances depending on the particular allocation. The range of relative performance differences seen was up to 30%.

A subset of the SPEC CPU2000 benchmarks were used. The full set was not used because of the large number of possible combinations of groups of 4. The benchmarks were rank ordered

Run	Physical Processor 0		Physical Processor 1	
	Hyper-Thread 0	Hyper-Thread 1	Hyper-Thread 0	Hyper-Thread 1
1	A	B	C	D
2	A	C	B	D
3	A	D	B	C

Table 3.9: Allocations of benchmarks A, B, C and D to processors.

based on their mean system speedup observed in the earlier experiments. Eight benchmarks were reasonably uniformly selected with care to maintain a balance of integer and floating point benchmarks. The benchmarks used were: *164.gzip*, *181.mcf*, *186.crafty* and *255.vortex* (integer); and *171.swim*, *177.mesa*, *179.art* and *200.sixtrack* (floating point). All possible combinations of 4 benchmarks from this set were used in the experiments.

Each experiment consisted of a run for each of the three possible allocations of processes to processors (ignoring equivalent cases). Table 3.9 shows these allocations. In a similar manner to the earlier experiments each benchmark was run in a loop on its respective processor and timings were started after all benchmarks had completed at least one full run. Three timed runs were used and the per-benchmark performance ratio was calculated as its base run time divided by its simultaneous runtime. The system speedup in this experiment is the sum of all four benchmark performances ratios divided by the number of physical processors (2). This provides a speedup normalised to sequential processing on both physical processors (with no Hyper-Threading).

The Northwood machine used in the earlier experiments and described in Table 3.2 was used for this set of experiments. The machine had two physical processors each with two Hyper-Threaded logical processors. As in previous experiments the Linux 2.4.19 kernel was modified with a patch to allow explicit allocation of processes to processors.

It might be imagined that these experiments could be conducted on paper using the results from the earlier experiments by simply summing the performance ratio figures from the two pairs of benchmarks. This however does not take into account the extra system-wide interaction, for example on the memory bus.

For each group of four benchmarks the relative performance difference is recorded. The range of relative performances is shown in Figure 3.19. The largest difference is 1.30 and the mean value is 1.10. In 40% of cases the best schedule resulted in a greater than 10% speedup over the worst schedule.

These experiments were performed in order to assess the performance gains possible by careful allocation of processes to processors. The processes remain on the same processor for the entire experiment. As the benchmarks go through different phases of execution it is likely that there will be a more optimal allocation of processes to processor than the allocation that yields the best over-all performance. Furthermore, gains may be had by resorting to time-sharing (and idling one logical processor) when simultaneous processes interact so badly that their combined performance is worse than serialising them.

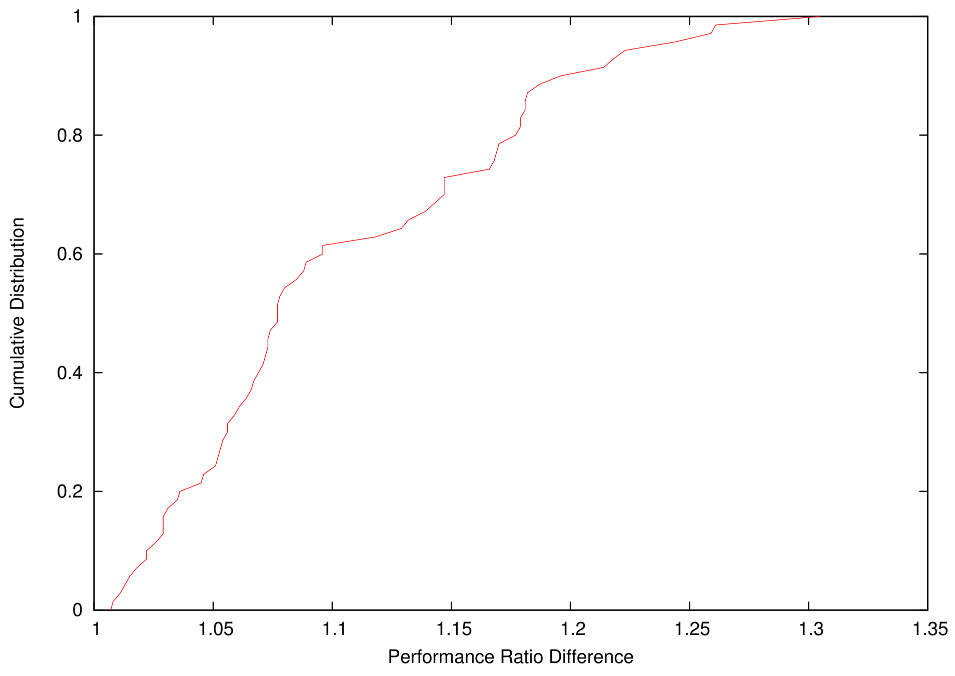


Figure 3.19: Cumulative distribution of relative performance differences between the alternative processor allocations for groups of four benchmarks.

3.6 Summary

In this chapter I have presented measurements of workloads running on Intel Hyper-Threaded processors in real, production systems. I have shown that the system-wide performance is highly variable and depends on the resource demands and characteristics of the simultaneously executing processes. I have shown that the share of performance between simultaneous processes is not always fair. I then showed how the performance of the workloads varied over the course of their execution and how the different phases of the simultaneously running processes affect the workloads. Using data from these experiments I have shown that the performance can be estimated on-line using data from performance counters with a reasonable confidence; this result is used in the following chapter to aide scheduling decisions.

I finished the chapter with a study to demonstrate the performance gains possible with SMT-aware scheduling. I showed that groups of four applications scheduled onto the four logical processors in a two physical package Hyper-Threaded system could exhibit performance differences of up to 30% depending on how they were allocated to the processors. The next chapter expands on this topic by adding SMT awareness to the operating system scheduler.

Chapter 4

A Process Scheduler for SMT Processors

In this chapter I describe the problems with using a traditional process scheduler with simultaneous multithreaded processors. I describe the characteristics of SMT that cause these problems and discuss techniques that could be used in a scheduler to avoid the problems. I present implementations of schedulers for the Linux 2.4 kernel that are sensitive to the characteristics of SMT processors and able to exploit them. I go on to evaluate my scheduler against a traditional scheduler and comment on how the techniques can be applied to other operating systems.

4.1 The Context

Intel originally marketed the Pentium 4 with Hyper-Threading as a way to get two processors in a single package. As described in Chapter 2, the two logical processors are abstracted to make them appear as two separate processors in an SMP arrangement. The operating system need not therefore be aware that a Hyper-Threaded processor is being used. Whilst this is sufficient for correctness, it is not ideal for performance. In Chapter 3 I showed that processes running on different logical processors could affect each other's performance with a large degree of variability.

There are two ways in which an SMT processor could be allocated by a scheduler:

- The processor could be treated as a set of individual, independent logical processors. This requires that the hardware threads are heavyweight, i.e. processes. This interface is provided by Intel's Hyper-Threaded processors.
- An SMT processor could be used to run true multithreaded workloads. In this scenario the physical processor is allocated as a single resource rather than separating out the logical processors. A multithreaded workload written or compiled for this particular SMT processor would be able to exploit its characteristics.

In this Chapter I choose to focus on the first scenario for the following reasons:

- Many common workloads are not written to be multithreaded, those that are tend to have a dominant main thread [Lee98] with other threads providing occasional support functions.
- A multithreaded workload compiled for one implementation of SMT may not work very well on another due to differing characteristics. This reduces the value of the second allocation strategy.

- With Hyper-Threading now available on standard desktop Pentium 4 processors, they will be used for a greater range of non-scientific workloads which tend not to be multithreaded or are not compiled for specific processor generations.

4.1.1 Problems with Traditional Schedulers

A scheduler that is completely unaware of the hierarchical nature of SMT processors using the logical processor abstraction will not know that resources are shared between "sibling" processors. As described in Chapter 2 this can lead to runnable tasks being scheduled on two logical processors of one package while another package remains idle. In addition, such a scheduler misses out on some scheduling flexibility: since logical processors on the same package share much state, particularly caches, a process can be migrated between logical processors with little loss compared to migrating between physical processors. This flexibility is useful when balancing load across processors.

Even if a scheduler is aware of the hierarchy and can avoid these problems then it is still not able to take the thread interactions into account in order to maximise the throughput of the system. Schedulers generally view all tasks equally in terms of their execution characteristics (with the exception of priority boosts for IO-bound tasks).

4.1.2 Scheduling for Multithreading and Multiprocessing Systems

In this section I present a brief overview of scheduling schemes and terminology used in multithreaded and multiprocessor systems.

The term *processor affinity* is used to describe a given process having a preference to be scheduled to execute on one or more specified processors. Processor affinity can be influenced by factors such as soft-state built up on one processor, the suitability of a processor for the task or the available resources in a heterogeneous system. Most multiprocessor schedulers use processor affinity in some form.

Cache affinity is a form of processor affinity where the process has an affinity for the soft-state it has built up in the processor caches. Whilst this may seem to be the same as basic processor affinity it is actually a dynamic scheme. If a given process it preempted by a second process that causes data belonging to the first process to be evicted from the cache then the first process' affinity for that cache (and therefore processor) reduces. The extreme is that all data belonging to the first process is evicted. In this case the process has no affinity for any cache/processor so the scheduler can assign it to any processor. Measuring the cache affinity of a process accurately would need potentially costly hardware support but an approximation can be made by counting cache line evictions to estimate process cache footprints [Squillante93].

The concept of cache affinity is relevant to SMT architectures because the caches are shared by the threads. The scheduler need not worry which logical processor within a given physical package it assigns a process to, the view of the cache will be the same.

When an affinity-based scheme is in use it is possible that the load will be unevenly shared between processors. There are a number of schemes that are used to re-balance load. A *work-*

sharing scheme is where newly created threads are migrated to a different processor from the one executing the thread that created them. This differs from the standard behaviour where child threads inherit the affinity data from the parent thread. *Work-stealing* is where an under-utilised processor "steals" threads from busier processors.

4.1.2.1 *NUMA Architectures*

The hierarchical nature of SMT systems places similar demands on a scheduler to non-uniform memory access (NUMA) systems. A NUMA system is a distributed shared-memory multiprocessor system where the latency of access to memory depends upon the location of the data. This is in contrast to the more familiar uniform symmetric (SMP) systems where the basic memory access latency is the same regardless of which processor and memory locations are involved. NUMA systems are structured as a number of processing nodes each containing one or more processors and a section of the memory. Any processor can access any word of memory but accesses from nodes other that the processor's own will take longer. To get the best performance from a NUMA system the scheduler and OS memory management system must arrange that processes run on the node hosting the bulk of the data they require. The processes have an affinity for the node.

Although SMT systems are symmetric in terms of memory access latency they draw a number of parallels with NUMA systems when cache contents are considered. An SMT physical processor package can be viewed in a similar manner to a processing node in a NUMA system: it has multiple (logical) processors and some shared storage (caches in the SMT case) equally accessible to all processors in the node. Processes are package (or node, or cache) affine.

In both SMT and NUMA systems the affinity of processes is more flexible than simple processor affinity. Schedulers for both types of system can exploit this to give greater scope for more efficient schedules.

4.2 Related Work

In this section I describe related work on hardware support for thread scheduling issues and scheduler support for SMT processors. The relatively recent availability of SMT systems means that most of the related work is simulation-based.

4.2.1 Hardware Support

Cazorla *et al* suggested that it is impossible to predict the performance of a given thread running with other threads on an SMT processor [Cazorla04]. They proposed an operating system interface to the processor to allow a number of high priority threads to be run with a guaranteed IPC. Lower priority threads could then use any remaining processor resources. The operating system established each thread's requirements by using a sampling phase with each thread running alone. The OS would then change processor parameters controlling the sharing of resources in order to achieve the promised IPC.

It should be possible to use this design to enable more general operating system control of per-thread resources. Intel's Hyper-Threaded Pentium 4 could benefit from the technique if it was to allow the various statically partitioned or duplicated resources to have their partitioning dynamically changed by the OS. This would allow the OS to improve the fairness of simultaneously executing threads by adjusting the amount of resource they each use.

4.2.2 SMT-Aware Scheduling

Parekh *et al* introduced the idea of thread-sensitive scheduling [Parekh00]. They evaluated scheduling algorithms based on maximising a particular metric for the set of jobs chosen in each quantum. The metrics included various cache miss rates, data memory access time and instructions per cycle (IPC). The algorithm greedily selected jobs with the highest metric; there was no mechanism to prevent starvation. The work was based on a simulation of up to 100 million retired instructions from each job. What is not clear from their work is how costly the scheduling algorithm is to run and how they collected and recorded the performance metrics used by it. The simulator did not model the operating system in detail but instead performed the required functions directly. The relatively short simulation window would have masked most of the phase change effects. Their average speedup over a simple round-robin scheme was 11%. They found that maximising the IPC was the best performing algorithm over all their tests.

Snavely's work on "symbiotic jobscheduling" [Snavely00, Snavely02] described the "SOS" (sample, optimise, symbios) scheduler which operates in phases. The sample phase involved the scheduler evaluating different combinations of jobs and recording a selection of performance metrics for each *jobmix*. With this data in hand, the scheduler chose permutations of jobs giving the best composite performance which were then used in the "symbios" phase. Snavely noted that as jobs are started and terminated and as the behaviour of individual jobs changes during execution the optimal jobmixes will change. To allow for this the scheduler returned back to the sample phase to collect fresh data. The variables in the algorithm included the length of time to stay in each phase. Snavely suggested a ratio of 10 to 1 for the time spent in the symbios and sample phases respectively. The length of the sample phase will depend on the number of possible job combinations and the size of the subset of these desired. The work used simulation and it is not clear how much, if any, of the operating system was modelled. The scheduler provided an average speedup relative to naïve scheduling of about 7%.

Both Parekh *et al* and Snavely focused on scenarios where there were a number of largely compute-bound batch jobs to be completed. Although both schedulers could deal with jobs coming and going, the effective churn rate associated with a system running interactive, IO-bound jobs would stretch this ability.

Kumar *et al* proposed a multi-core processor architecture with a heterogeneous mix of core implementations (but all using the same ISA) [Kumar04]. The example they used was the Alpha architecture: the newer EV6 (21264) core is 5 times as large as the EV5 (21164) but not 5 times as fast. They proposed a multi-core design with 3 EV6 and 5 EV5 cores and compared it to a 4 core EV6 design. In addition they considered an SMT version of the EV6. They found that careful allocation of workloads to processes and SMT threads could give them better system-wide throughput than the more simple CMP arrangements. They proposed a scheduler in a similar spirit to Snavely *et al* (Dean Tullsen is an author on both Snavely and Kumar's papers). They built on this by adding a trigger mechanism to cause the sampling phase to be re-entered. They suggested three triggering schemes. *Individual-event*: if a thread's IPC changed by more than 50%; *global-event*: if the sum of all IPCs changed by more than 100%; and *bounded-global-event*: as global-event but does not allow a re-sample until at least 50 million cycles have elapsed and forced a re-sample after 300 million cycles. They found the three mechanisms performed in the order given above with bounded-global-event being the best.

Nakajima and Pallipadi used data from processor performance counters to change the package affinity of processes in a two package, each of two Hyper-Threads, system [Nakajima02]. They aimed to balance load, mainly in term of floating point and level 2 cache requirements, between the two packages. Their "scheduler" was implemented as a user-mode application that periodically queried performance counters and swapped pairs of processes between packages by changing the processor affinity masks of the processes. They measured speedups over a standard scheduler of approximately 3% and 6% on two test sets chosen to exhibit uneven demands for resources. They only investigated workloads with four active processes, the same number as the system had logical processors. The technique can extend to scenarios with more processes than processors however the infrequent performance counter sampling can hide a particularly high or low load of one of the processes sharing time on a logical processor.

Fedorova *et al* argued that "chip multithreading" systems (those where the physical processor has multiple cores each running multiple hardware threads) would benefit from an intelligent scheduler [Fedorova04b]. In later work they implemented schedulers based on cache modelling and working-set techniques to reduce level 2 (L2) cache miss rates [Fedorova04a]. This work assumed an interleaved style of multithreading. While L2 miss rates are important to SMT scheduling the dynamic nature of the resource sharing means that other factors need to be taken into account.

4.3 Practical SMT-Aware Scheduling

I now describe the design, implementation and evaluation of a number of techniques to incorporate knowledge of an SMT processor into the scheduler. The common theme is the provision of heuristics, based on performance counter observations, to a general purpose scheduler.

For a scheduler design or modification to be practical and useful it must fit into the existing model of scheduler design. Since real schedulers are designed to work on many different systems, with and without SMT, a scheduler that is too SMT specific will be of little interest. Existing schedulers are generally of a dynamic priority nature: tasks with the highest priority at any given time are scheduled to run on the available processors; the priorities change over time to avoid starving a task of processor time. A sensible place to influence the scheduling decision in an SMT-sensitive way is in the calculation of dynamic priority. By doing this the existing facilities of static base priority, priority boosts (temporary increases in priority often given to tasks that have recently had I/O requests satisfied) and ageing (to avoid starvation) can be retained.

An alternative is to use SMT-specific knowledge to change the processor affinity and allow the scheduler to operate in the traditional manner. This method has a much longer-term effect than priority modification and makes responding to the different phases of a task's execution more difficult.

The SMT specific knowledge useful to scheduling will be of the interactions of processes with certain characteristics. The scheduler must therefore know about the characteristics of currently running, or candidate processes. Such knowledge could be acquired by static inspection of program text segments (the executable code) or dynamic measurement of the running processes. Static analysis of the programs is beneficial in its cost (a one time activity) but only provides a limited amount of information; effects such as mis-speculation and cache hit rates are important. These effects could only be obtained off-line through simulation/emulation; it would be just as well to run the code and measure it. Dynamic measurement of the running processes provides more information, not only on the process itself but on how it is interacting with the processes on the other logical processor(s). Dynamic measurement will incur an overhead for both sampling and evaluating the sampled data.

4.3.1 Design Space

The level of support for SMT-sensitive scheduling can vary from basic hierarchy awareness (as now provided in current operating systems) through to an attempt to create an optimal schedule ensuring the best processor throughput at all times. The cost and complexity of the scheme will generally increase as the number of features is increased. The increasing awareness of the SMT characteristics also makes a more complex scheduler less portable to other SMT and non-SMT systems. I implement a selection of points between SMT-unaware and a scheme fully aware of the particular characteristics of the SMT processor in use:

"naïve" A traditional multiprocessor scheduler that treats all logical processors across the system as equals.

"basic" A traditional multiprocessor scheduler modified to understand the logical/physical hierarchy.

"tryhard" A scheduler that uses measurement of the processes' execution to avoid the bad cases and try to move to better combinations of processes to co-schedule.

"plan" A scheduler that uses measurement of the processes' execution to try to obtain an optimal schedule in terms of processor throughput by periodically creating a scheduling plan.

"fair" This scheduler has the primary goal of ensuring fairness between processes. Processes that are not getting their fair share of resources when running under SMT are compensated with an extra share of processor time.

4.4 Implementation

I chose to implement the SMT-aware scheduler modifications on Linux because of the accessibility and ease-of-modification of that kernel. The modifications were made to a version 2.4.19 kernel, the latest stable version at the time the work was started and the same version as used for the measurements in Chapter 3. In Section 4.6 I discuss the applicability of these techniques to other operating systems and to the subsequent Linux 2.6 kernel.

4.4.1 The Linux Scheduler

The Linux scheduler in version 2.4 kernels is based around a single queue of runnable tasks [Bovet00]. The scheduling algorithm uses dynamic priorities based on static base priorities and a form of ageing to avoid starvation. Processes are preemptable meaning that a new runnable process with a higher priority than the current runnable process will force a reschedule. Scheduling decisions are made when a running process exhausts its current time-slice allocation or a higher priority process is woken up.

Scheduling works in a sequence of *epochs*. At the start of an epoch each process is given a time-slice of a number of *ticks* (a tick is usually 10ms) based on its static priority and the number of ticks it had left from the previous epoch. An epoch ends when there are no runnable processes, or all the runnable processes have exhausted their time-slices. The current process' tick counter is decremented during each timer interrupt. Each time a scheduling decision is made the dynamic priority is calculated for each runnable task and the one with the highest is selected to run. The dynamic priority, or *goodness*, of a task is zero if it has exhausted its time-slice otherwise it is the number of ticks remaining of its time-slice added to its static priority. The goodness is further modified upwards if the candidate task shares an address space with the previously running task.

Multiprocessor scheduling uses the same single queue and per-task time-slices. Each processor runs the scheduling function and goodness calculations itself. Processor affinity is supported by increasing the goodness of tasks when calculated by processor which they last ran on. Enforced processor affinity (such as the `cpus_allowed` feature used in Chapter 3) is implemented by only considering the goodness of tasks that are allowed to be execute on the processor running the scheduling function.

4.4.2 Extensible Scheduler Modifications

In order to be able to efficiently test scheduler modifications without time consuming machine reboots, the following modifications were made to the Linux scheduler. Hooks were put into the kernel to allow important functions to be overridden or callbacks to be received on certain events. A loadable kernel module (LKM) containing override functions was used to attach to these hooks. A LKM can be loaded and unloaded at run time facilitating a fast development and experimental cycle. With the LKM unloaded, the default Linux scheduler functions were used.

The kernel evaluates the "goodness" of each runnable process whenever its `schedule()` function is called. The `goodness()` function is called with the task metadata of the runnable process being considered and returns a weight value based on the current situation and task properties. The task with the largest weight value will be scheduled on that processor. The standard Linux `goodness()` function returns a weight that is higher for realtime tasks than non-realtime tasks. Higher priority realtime tasks result in a larger weight. Non-realtime tasks have a weight that depends on the following factors:

- number of time-slice ticks it has left in the current epoch,

- whether it was last run on the CPU that is currently performing the scheduling algorithm,
- whether it has the same address space as the task previously running on this CPU, and
- the "nice" value (static priority) the task has.

The standard `goodness()` function was modified to add a conditional call to an alternative function within the LKM.

The time-slice ticks for the currently running process are decremented during the timer interrupt handler's execution. A similar replacement function hook to the above was added to the `update_process_times()` function. This hook was used for the fractional tick technique described below.

Since my schedulers are designed to use data collected from performance counters they require notification of changes to running tasks so that the counters can be sampled and data attributed to the correct tasks. To this end, the context switching function, `__switch_to()`, was hooked to provide a callback to the LKM. `__switch_to()` is called just before a context switch; the module function is called with the task metadata of the task being switched out.

4.4.2.1 *Module Features*

The scheduler module is able to read performance counters but their configuration is handled externally for simplicity. The mechanism described in Section 3.2.1 was used for this.

The LKM provides a monitoring function via the `/proc` file-system which allows online querying of certain scheduler parameters. A context switch recording facility was built into the module to allow the time and details of all context switches to be recorded for debugging purposes. This was implemented in the `__switch_to()` callback and wrote the details into a circular buffer. This buffer was made available to a user-space graphical visualisation tool through a second `/proc` file. On each context switch the cycle-counter, logical processor number and the outgoing PID were logged along with up to two parameters according to the particular scheduler modification being used. This facility was only enabled during development of the schedulers and not during timed experimental runs.

4.4.3 Performance Estimation

The proposed scheduling designs require knowledge of each process' current performance ratio. The performance ratio of a process is defined in this dissertation as the rate of execution of the process when running in a given situation (simultaneously with another process in this case) compared to its rate of execution when running alone on the system. I showed in Chapter 3 how the performance ratio of a process running under SMT can change dramatically during the process' execution. In a live system it is not possible to know what the standalone execution rate of a process should be without actually running it alone for a period. The changing phases of a process' execution means that this technique is of limited value. Instead the performance can be estimated from parameters measured by the hardware performance counters. In Section 3.4.1 I developed a linear model for this prediction. This model is implemented here.

At a sample point (defined by the particular scheduler design) the performance counters were read. The elapsed cycle count since the last sample can be combined with the differences between samples of the counter values to provide a performance estimate. The performance counters are per-package but can be accessed from either logical processor. The code must know which logical processor it is running on so that it can know which counters refer to the current logical processor and which to the other logical processor. A record was kept of the previous sample along with the cycle counter "timestamp". Counter overflow was detected by observing a counter value as being less than its previous sample. The frequency of sampling was sufficiently high to allow this method. For the purpose of calculating the counter differences an overflowed value had its bit 40 set to account for the overflow (the counters are 40 bits wide).

The model from Section 3.4.1 is based on events per cycle, using performance counter events for trace-cache (TC), level 1 (L1) and level 2 (L2) cache misses and floating-point (FP) and all (Insts) instructions retired. The events are counted independently for each logical processor; in the calculation of the estimated performance ratio for one of the logical processors the counters corresponding to that processor are referred to here as ("subj") and those of the other logical processor as ("back"). The model is (note that *L2_back* is missing because it was shown to be statistically insignificant):

$$\begin{aligned} performance = {} & 0.401 + 29.7 * TC\_subj + 55.7 * L2\_subj + \\ & 0.352 * FP\_subj + {}^{-}0.022 * Insts\_subj + 2.19 * L1\_subj + \\ & 32.7 * TC\_back + {}^{-}0.418 * FP\_back + 0.506 * Insts\_back + \\ & {}^{-}3.54 * L1\_back \end{aligned}$$

It is desirable to perform the calculations using integer arithmetic because floating point registers are not saved when entering the kernel. The most effective technique is to use a fixed-point scheme where the low bits of an integer value are interpreted as the fractional value; the calculation was performed with 15 bits of fractional value which gives a resolution of approximately 0.00003, the final result was right-shifted by 5 bits giving a resolution of approximately 0.001. The higher resolution during the calculation was required to prevent the smaller factors from being discarded. The more compact representation for the result was used to allow 32 bit arithmetic to be used in subsequent calculations.

The overhead of performing frequent performance estimates will depend upon how often the particular scheduler implementation calls the function. The evaluation of the schedulers measures this overhead as well as the gain since the entire system is being measured. An indication of the overhead was be obtained by causing estimates to be performed without any other scheduler modifications in place; system speedups suffered approximately a 2% reduction when estimates for each logical processor were made during each timer interrupt. Determining the sensitivity of the frequency of calculating performance estimates is left to future work.

4.4.4 SMT-Aware Schedulers

In this section I describe the implementation of the different SMT aware scheduler designs introduced in Section 4.3.1.

4.4.4.1 *Scheduler: "basic"*

The only change required for the *basic* scheduler is the affinity of processes for physical packages rather than logical processors. This was implemented by changing the own-processor test when increasing the goodness. The logical processor numbering scheme and the number of logical processors per package means that simply ignoring the least significant bit in the comparison of the processor running the goodness function and the process' previous processor allocation is sufficient. Newer Linux kernels incorporate this modification as a more general (and elaborate) "sibling map".

```
if (PHYSICAL_PACKAGE(p→processor) = PHYSICAL_PACKAGE(this_cpu))
    weight += PROC_CHANGE_PENALTY;
```

4.4.4.2 *Scheduler: "tryhard"*

The *tryhard* scheduler aims to schedule a task on a logical processor that will provide the best total package performance when running with the task on the other logical processor in that package. To do this a matrix of task pair performances needs to be kept.

Performance estimates were calculated using the technique described in Section 4.4.3 each time one of the logical processors on a package performed a context switch. The sampling was initiated from the `__switch_to()` hook. Having obtained performance estimates for the two simultaneously running processes a rolling average for that pair's combined performance was kept. It is not feasible to store averages for every possible process pairing so a hash-table style scheme was used. This table needs to be fast to access for both updates and queries therefore an array indexed by a hash of the two PIDs was used. The array contained 64 entries and had a hash function designed to minimise the number of collisions in the common case of closely spaced PIDs:

$$((\text{pid2}\ \&\ 7) << 3 \mid (\text{pid2}\ \&\ 0\text{x}38) >> 3) \oplus (\text{pid1}\ \&\ 0\text{x3f})$$

where the PIDs were numerically ordered as `pid1` and `pid2`. Collisions were ignored since the data is used only as a heuristic.

The scheduler used the recorded performance data to influence the choice of processes to schedule by adjusting the goodness of a process depending on the recorded performance data for that process when simultaneously executing with the process currently on the other logical processor of the package. The result was that processes that would give a higher system speedup were favoured but the time-slice and other components of the standard dynamic priority were still respected.

Initially the time-slice (the dynamic part of the priority) was factored by the performance value. However, the limited range of possible values restricted the effect of the modification. Instead a value based on the performance sample was added to the dynamic priority. This value was the performance sample divided by 64 to give a value normalised to 16; this conforms to the order of magnitude of the other factors in the goodness calculation and achieved the desired effect.

4.4.4.3 *Scheduler: "plan"*

The *plan* scheduler attempts to produce an optimal schedule by choosing pairs of processes to simultaneously execute. It differs from *tryhard* in that it plans ahead what to execute rather than reacting to what the other logical processor is doing.

This "gang scheduling" based approach is common in related work which, being simulation based, has the luxury of tasks that are always runnable and the ability to easily perform the scheduling function for all processor at once. In a real system it will often be the case that one or more tasks are blocked, even if they are nominally compute-bound. Additionally the scheduling decisions are not made simultaneously on all logical processors because of unsynchronised timer interrupts and reschedules performed on the return from system calls.

There are three parts to the implementation: data recording, schedule planning and schedule realisation. These are described in turn.

Each time a context switch was performed the summed performance ratios for the two processes on that physical processor (called the "package performance") were used to update a record of "best partners". These recorded, for each process p, the PID of the process q that gave the best physical package performance when simultaneously running with p. The idle task was excluded because its use of the `HLT` instruction allowed the simultaneously executing process to utilise all the processor resources; this would give a false indication of the performance. The package performance was stored so that successive updates could check whether an improved pairing has been found. To allow for changes in phase and the termination of processes the recorded values were decayed over time. The best partner record was stored as an extra field in the task descriptor structure (`task_struct`) for the process p. This allowed quick and direct access when an update was performed. To avoid including inactive processes in scheduling plans a per-process activity counter was incremented during the `update_process_times()` callback; only processes with a non-zero counter were including in the plan, the counter was reset at each planning cycle.

A new scheduling plan was created periodically by a user-mode tool that read the recorded performance data from the LKM `/proc` interface. This data contained a record for each active process consisting of the best partner PID and the estimated package performance. The schedule planning program sorted the records by the package performance values and selected pairs of processes greedily, bypassing records containing processes already selected. This operation was performed at a relatively low frequency so the cost of the sort and search was acceptable. The plan was written back to the `/proc` interface for the scheduler to use.

The plan was realised by an approximation to round-robin gang scheduling of the pairs. The goodness function heavily biased the goodness of a process if it was a member of the current gang. It was not desirable to completely suppress other processes since one of the gang may block or terminate. Additionally the choice of pairs to schedule was based on samples from different pairings. By allowing other pairs to occasionally execute, these samples could be updated. This technique also allows for processes that were blocked during the last round of sample gathering and hence not in the plan.

The correct implementation of the plan was verified using the context switch recording function of the LKM.

4.4.4.4 *Scheduler: "fair"*

This scheduler attempts to bias processes' CPU time allocations by their performance under SMT. A thread that suffers a poor performance when simultaneously executing is given a greater share of CPU time. The Linux scheduler normally provides even sharing of processor time by allocating each process a time-slice (as a number of ticks) at the start of each scheduling epoch. Ticks are consumed as the process uses CPU time. This scheme was modified for the *fair* scheduler by scaling the magnitude of the time-slice decrement by a factor proportional to the ratio of the estimated performance to an assumed average performance. The average performance was declared to be 0.6 based on the measurements in Chapter 3. For an estimated performance of *p* the tick decrement was *p/0.6*. The magnitude was bounded in both directions to avoid problems with wildly inaccurate performance estimates.

Tick counters are integers with a scale of 1 tick per quantum (usually 10ms). In order to allow fractional decrements (without resorting to using floating point values which is not advisable in the kernel) the counter values were scaled by 256 (essentially a fixed-point representation).

The hook in the timer interrupt driven `update_process_times()` function was used to provide this alternative tick decrementing. The performance counter samples were used to estimate the current performance using the method described in Section 4.4.3.

This implementation allows static priorities to be used in the normal way because the per-process time-slice is still initialised by the static priority.

4.5 Evaluation

The effectiveness of the scheduler can be determined according to a number of metrics:

- the system throughput of a set of tasks,
- fairness across a set of tasks, and
- respect of static priorities.

In addition to evaluating my scheduling algorithms against a traditional scheduler, an algorithm from related work was implemented and measured.

The algorithm is based on Parekh *et al*'s "G_IPC" scheduling algorithm which they found to have the best all-round performance in their experiments [Parekh00]. At the start of each quantum G_IPC greedily schedules the set of threads with the highest individual IPCs. Parekh *et al* used a simulated system with no provision for starvation avoidance. Processes in their system did not block. My implementation, "*ipc*", approximates G_IPC by heavily biasing the goodness value of a process based on its rolling-average IPC: a higher IPC translates a higher dynamic priority. For the bulk of the Linux scheduling epoch this implementation provides the same outcome as G_IPC. However, my method avoids starvation of processes with natively low IPC.

4.5.1 Method

The method used to evaluate the schedulers was similar to the pairwise and asymmetry experiments in Chapter 3 except that the scheduler was given the freedom to migrate tasks between logical processors. Sets of tasks were defined and each task in the set was executed in an infinite loop with the duration of each run recorded. The SMT-aware schedulers reference processes by their PIDs so there is no memory of performance data between successive runs of the same benchmark or even of different instantiations of the same program with different data sets (such as *164.gzip*).

The tasks were started with a random staggered start. After all the tasks were started a nominal start time was declared. After a specified time the experiment ended. Only task runs which were entirely within the start-end window were included in the results although all tasks were running throughout the window. The execution time of each included task run was compared to its stand-alone run time to get a performance ratio for that task (as described in Section 3.2.4). The performance ratios were summed to give a system speedup. As with the experiments in Chapter 3 the system speedup is relative to zero-cost context switching on a single processor. This scheme is preferable to a purely IPC based scheme because it takes account of the varying baseline characteristics of the tasks. It is similar in spirit to Snavely *et al*'s weighted speedup. Each experiment was run three times and the mean speedups reported.

One problem with running many benchmarks simultaneously is the high memory usage. Pages having to be swapped to disk complicates the system behaviour by causing tasks to block on page faults, increasing the kernel activity by reading and writing pages and potentially reducing the performance of intended disk accesses by the benchmark tasks. While these effects are interesting in a full-system study it is important to be able to understand different effects in isolation before considering their combined effect. The experiments were performed with swap space disabled. In the event of the system running out of memory the Linux kernel kills a task according to an algorithm that considers the task's virtual memory size, CPU time and system importance (i.e. superuser or user processes). The test harness monitored the system for this behaviour to prevent it causing invalid test runs.

The sets of benchmarks were chosen such that their total maximum memory requirement could be accommodated by the system taking into account memory used by the operating system and necessary support processes. Memory footprint sizes were obtained from Henning's descriptions of the SPEC 2000 benchmark suite [Henning00] and verified on the experimental machine. A budget of 750MB was allowed for each set.

The sets of benchmarks used were taken from those used in Chapter 3 and are shown in Table 4.1. These were chosen to provide a diverse range of workloads including integer, floating-point and memory intensive tasks and different number of processes.

The duration of the start-end window was chosen to allow all benchmarks to complete a sufficient number of times but without being so long as to be impractical. Times ranging from 2 to 6 hours were used. All tasks were given the same static priority (the default "nice" value of 0).

4.5.2 Throughput

The system throughput can be measured by the rate at which a given set of tasks progresses. A true measure of throughput can be obtained only while all of the tasks are running. The results presented here use the system speedup measure described above.

Figure 4.1 shows the system throughput for each test set running under each scheduler. Improvements over naïve scheduling of up to 3.2% are seen. To put this speedup into context, Nakajima

Set	Benchmarks
A	*164.gzip*, *186.crafty*, *171.swim*, *179.art*
B	*164.gzip*, *181.mcf*, *252.eon*, *179.art*, *177.mesa*
C	*164.gzip*, *186.crafty*, *197.parser*, *255.vortex*, *300.twolf*, *172.mgrid*, *173.applu*, *179.art*, *183.equake*, *200.sixtrack*
D	*mp3*, *compile*, *image*, *glx*

Table 4.1: Benchmark sets for evaluating scheduler designs.

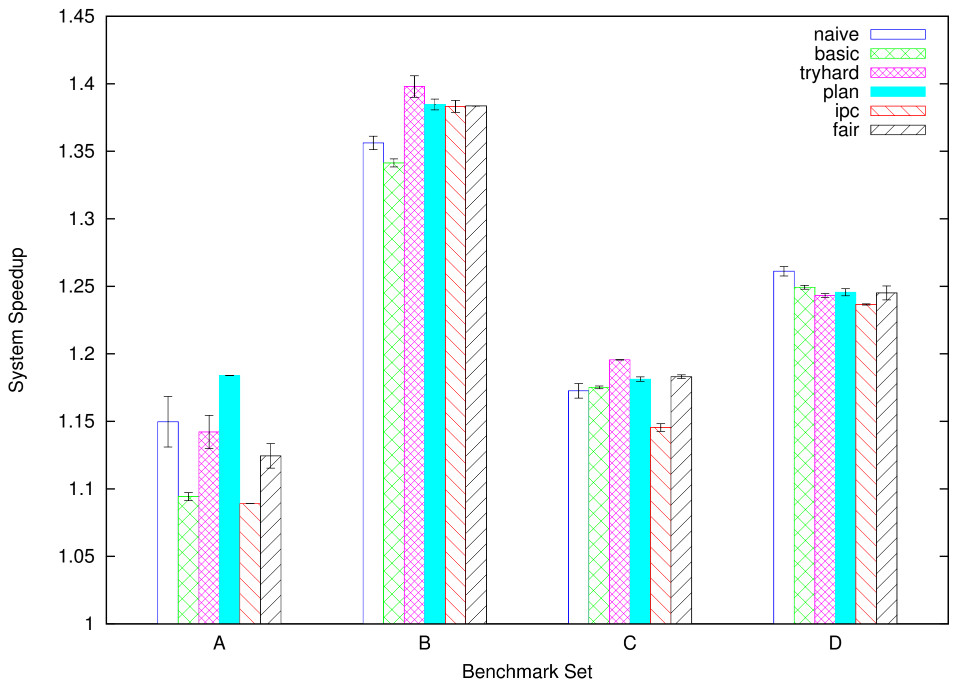


Figure 4.1: System speedups for the schedulers on the single package (*1x2*) configuration. Note the non-zero start for the y-axis.

and Pallipadi measured speedups of 3% and 6% on two task sets chosen to exhibit resource requirements that their scheduler could exploit.

The *tryhard* scheduler does reasonably well on benchmark sets B and C but results in a small slowdown (compared to *naïve*) on A and D. *plan* provides a speedup on sets A, B and C sets; its lack of an apparent speedup on set D is partly due to a false indication of high throughput when using the *naïve* scheduler (see below). The performance loss experienced by the *fair* scheduler is almost all due to the overhead of calculating the estimated performance ratio each time a tick counter is decremented. This algorithm is aimed solely at being fair; it makes no attempt to improve throughput.

Set A

This set consists of 4 processes which run on 2 logical processors. The system gets itself into a steady state where two processes on each logical processor get scheduled in a round-robin fashion, being preempted when they exhaust their time-slice. For *naïve* this steady state behaviour means that there is little chance of the simultaneous pairings changing so the system speedups obtained depend largely on the initial allocation of processes to logical processors. The high variance in the system speedups seen for these algorithms is because of the relatively large difference between performance ratios for the different possible pairings. The steady state reached in the experiments that gave the better throughput had *164.gzip* and *186.crafty* executing on one logical processor and *171.swim* and *179.art* on the other. In both cases the interaction effects of the two processes on a logical processor were similar (they have similar rows and columns in Figure 3.3(a)) so changing their relative phase has little effect.

tryhard provides little gain because it is hard for it to break the steady state cycle. The overhead of estimating performance causes a small net slowdown. *plan* does provide a speedup (compared to *naïve*) because it can cause migrations to occur and find better pairings. Set A under *ipc* causes the higher IPC benchmarks *164.gzip* and *186.crafty* to run together at the start of a scheduling epoch because they were greedily selected. When they exhaust their time-slices *171.swim* and *179.art* run together which results in a poor performance, particularly for *171.swim*. The result is a low average throughput.

Set B

Set B contains benchmarks with a generally good combined performance although *181.mcf* and *179.art* have both been shown to cause harm to *164.gzip* but not to *252.eon* or *177.mesa*. The odd number of processes in the set will cause frequent migrations of processes between the logical processors because there will always be a 60:40 split in load using the current affinity. This, and the changing number of processes affine to a logical processor, mean that there is no steady state as was observed with set A. With *naïve* all pairings will occur at some point so it would be expected that the bad pairings will occur reasonable often and hence reduce the average throughput. *tryhard* does well because it is able to reduce the number of times a bad pair is simultaneously executed. This algorithm reduces the goodness of a process that has been observed to yield a low performance when running with the process currently active on the other logical processor. The availability of many good pairings means that *tryhard* can usually find a better process to run.

Set C

The large number of benchmarks in set C gives a better choice for *tryhard* leading to an increase in performance compared to *naïve*. *ipc* performs poorly for benchmark set C because, as with set A, the pairs that have the highest IPC do not necessarily yield the best performance ratios.

plan shows a smaller speedup; this is due to a limitation of the planning algorithm: take a benchmark *P* that gives a good performance with other benchmarks *Q*, *R* and *S*; *P* will appear as "best partner" for those three benchmarks. When the planner greedily selects pairs it disregards pairs containing processes already selected, therefore it will select perhaps (*Q*, *P*) but will then have to disregard (*R*, *P*) and (*S*, *P*) because *P* has already been included in the plan. *R* and *S* will only be included in the plan if they appear as a "best partner" to some other process (a process not in the plan will still be run, usually towards the end of the epoch once the processes in the plan have exhausted their time-slices). An alternative behaviour would be to select pairs regardless of whether their members have already been selected. This method is not as unfair as it may seem because the time-slice of each process will still be respected; the disadvantage is that pairs containing the common process will lose the ability to be gang scheduled when the common process exhausts its time-slice, the remaining process in each pair will gain an arbitrary partner for the remainder of the epoch. A slightly more costly option would be to record a sequence of preferences[1] rather than just "best partners" (e.g. "second best", "third best" etc.). These two modifications were tried on this benchmark set: allowing pairs to contain processes already selected resulted in a system speedup of 1.183 (only marginally better than the original *plan* algorithm at 1.181); maintaining data on the "best three partners" gave 1.161 (slower than *naïve* at 1.173).

Set D

The *compile* benchmark consists of a series of many short-lived compiler and linker processes. These are hard for most of the SMT-aware schedulers to deal with; *plan* periodically samples data on "best partners" and calculates a sequence of pairs to gang schedule. The churn rate of processes means that this plan is almost always out-of-date and results in very little gain. *tryhard* and *ipc* work on a finer granularity but the very short lives of the processes limit the amount of useful characterisation that can be performed. A possible modification to these algorithms would be to maintain performance data across successive instances of the same program, perhaps by considering the program name (`argv[0]` in C).

The *naïve* scheduler appears to perform better than any of the other algorithms. However, this is at the expense of fairness — *compile* gets a small fraction of the total processor time with *glx* and *mp3* taking the bulk of the former's share (see Section 4.5.3 below). The latter two benchmarks general experience a better performance under Hyper-Threading than *compile* so cause the system speedup to be inflated. The other algorithms provide a much fairer share of processor time to *compile* which causes the system speedup to lose the artificial inflation it had from *glx* and *mp3*.

Summary

The *tryhard* scheduler yielded the best results over-all. It failed to produce an improvement on set A because it was unable to break a steady state pattern of execution. A combination of *tryhard* and some form of SMT-aware migration would be likely to perform well.

plan performed well in cases where processes were long-lived enough for their characteristics the be learned and a scheduling plan created and realised. Modification of this algorithm to store performance data between instances of the same program would be likely to provide a better performance. *plan* outperformed *tryhard* when the ability of the former to break a steady state schedule allowed optimal pairings to be found.

ipc generally performed poorly. This algorithm is based on related work. It favours processes with high IPC; this does not necessarily equate with high performance.

The *naïve* baseline scheduler is sometimes able to perform well but often the increased performance is down to a fortunate (but accidental) allocation of processes to logical processors. My SMT-aware scheduling algorithms generally provide a more reliable system throughput.

[1]Three "best partners" were recorded but not held in order; the periodic planning run performed the sorting.

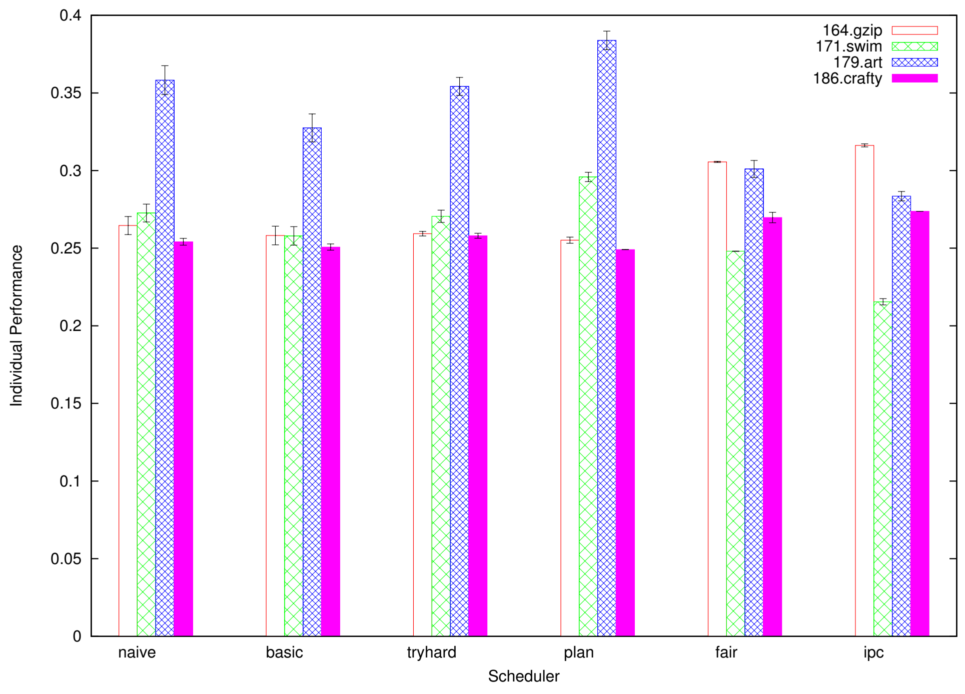


Figure 4.2: Individual performances of the benchmarks of set A running on the single package configuration (1x2)

4.5.3 Fairness

The throughput measurements described above give a speedup for the entire system. However, the goal of the scheduler is to provide a good speedup alongside fairness. All tasks were given the same static priority ("nice" value) so should exhibit similar individual performances. However, as seen in Chapter 3, having an equal allocation of processor time on a Hyper-Threaded processor does not guarantee an equal share of processor resources. The variation in the individual performances that were summed to give the system speedup can be used to gauge fairness.

Figures 4.2 through 4.5 show the individual performances of each benchmark running in the sets under each scheduler. A smaller spread of individual performances indicates a fairer schedule.

Set A

Figure 3.3(a) in Chapter 3 showed that *179.art* generally detriments the performance of the task it runs simultaneously with but rarely suffers a reduced performance itself. It is for this reason that *naïve* and some of the SMT-aware schedulers show this benchmark achieving such a good individual performance.

171.swim usually exhibits a periodic change approximating a sine function in its performance ratio when running simultaneously. Its estimated performance follows the same basic shape and predicts the peaks very well but over-estimates the troughs. The net result is an over-estimation of the benchmark's performance which causes the *fair* scheduler to over-penalise its tick counter. This effect is observable in the results as a lower then average performance ratio for *171.swim*.

ipc, which greedily schedules tasks with high IPCs, causes a reduction in performance for the

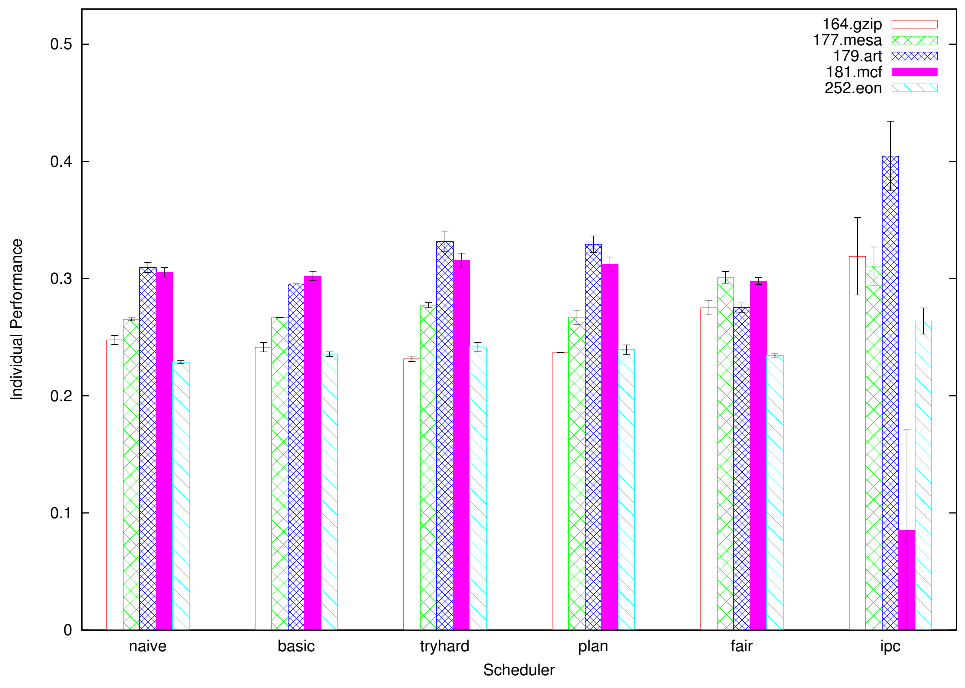


Figure 4.3: Individual performances of the benchmarks of set B running on the single package configuration (1x2)

low-IPC benchmarks *181.mcf* and *179.art* and an increase for *164.gzip* and *186.crafty* which both have high IPCs.

Set B

As with set A, *179.art* achieves a relatively high performance. *181.mcf* has a similar, although slightly less pronounced, thread interaction behaviour so shows a similar bias here. *ipc*'s preference for high-IPC tasks is shown clearly with *181.mcf* getting little processor time and therefore a poor over-all performance.

Set C

The previously seen effect of *179.art* is clearly visible again in set C. This set also contains *172.mgrid* which Figure 3.3(a) shows as having opposite properties to *179.art*: it gets hurt by many competing workloads but rarely causes a performance detriment to others. This effect is seen here as a generally low individual performance for this benchmark.

Set D

In most cases *glx* yields a better performance than the other benchmarks. In a similar manner to *179.art* in the other benchmark sets, Table 3.7 in Chapter 3 shows how the benchmark does harm to others but is not harmed itself.

naïve shows a particularly unfair spread of performance with *glx* having an individual performance ten times higher than *compile*. This is due to the combination of *compile*'s frequent forking and *glx*'s (and *mp3*'s to a limited extent) compute-bound nature. When a process forks the time-slice of the parent process is divided equally between the parent and child (although each get

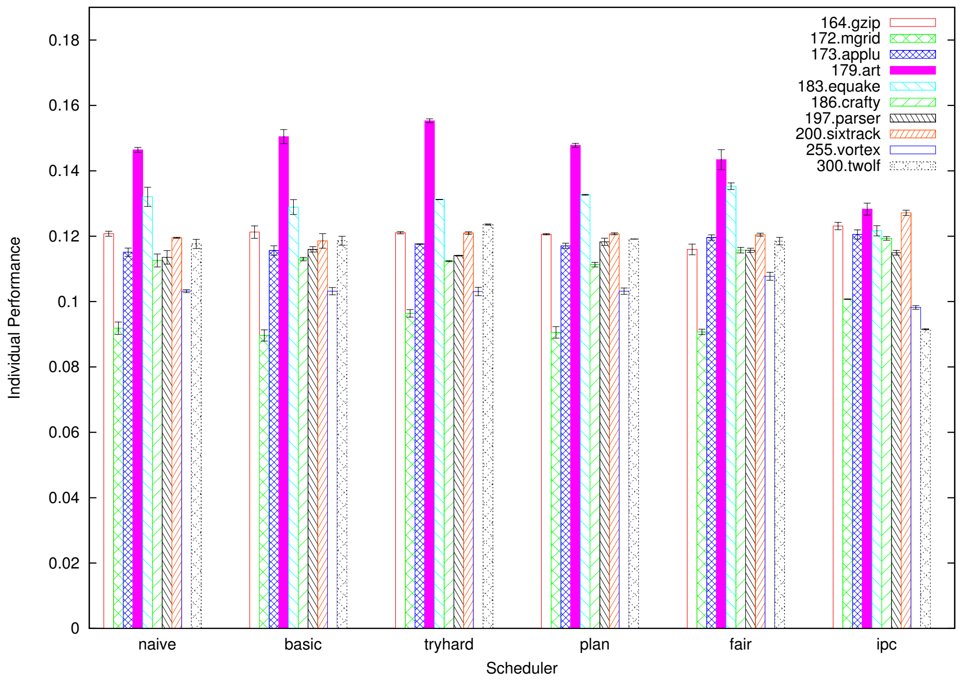


Figure 4.4: Individual performances of the benchmarks of set C running on the single package configuration (1x2)

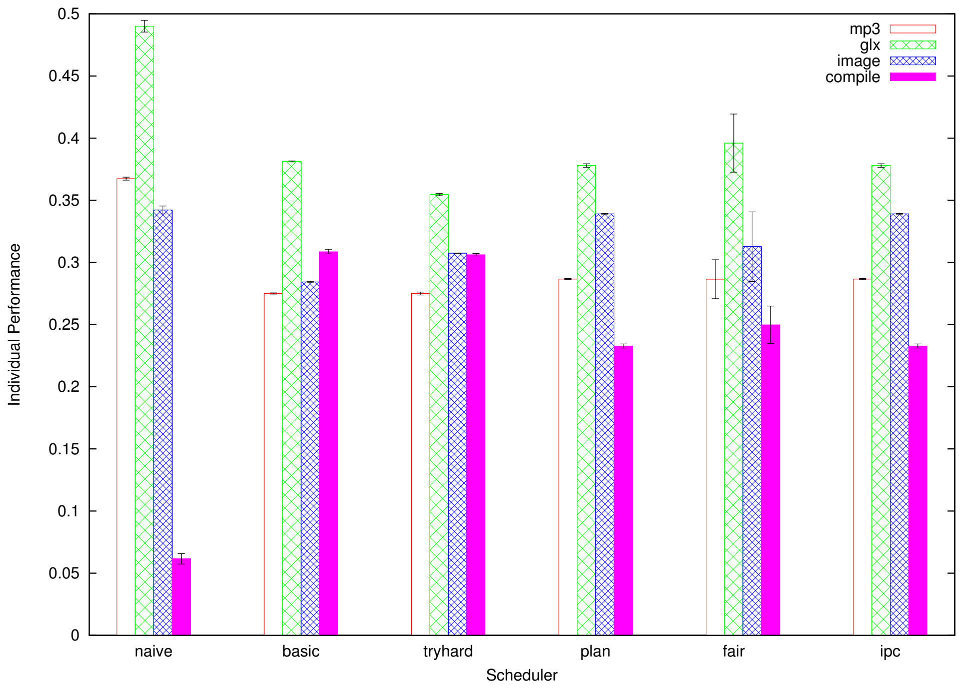


Figure 4.5: Individual performances of the benchmarks of set D running on the single package configuration (1x2)

Benchmark	"Nice" value	Time-slice milliseconds
252.eon	0	60
179.art	3	50
181.mcf	7	40
177.mesa	11	30
164.gzip	15	20

Table 4.2: Static priorities given to the benchmarks of set B.

a full allocation at the start of the next epoch); the current processor number is inherited by the child so it maintains the same affinity as the parent. When a family of processes, such as *compile*, is time-sharing a logical processor with a compute-bound task such as the X-server associated with *glx*, the recently forked processes, which now have low goodness, will often be waiting for the compute-bound process to exhaust its time-slice. The large amount of I/O performed by *compile* means that it often yields the processor back to the compute-bound process. The other logical processor is also busy so there will be very little migration of tasks between the logical processors. The effect of changing affinity to physical package rather than logical processor is clearly seen with *basic*; the processes making up *compile* are now able to freely migrate to the other logical processor so can take advantage of blocking-induced reschedules on that processor.

Summary

My SMT-aware schedulers generally provide fairness between threads that is as good as, or better than, that provided by a traditional scheduler. The algorithms that attempt to optimise for throughput respect the processes' time-slices so should provide an approximately equal allocation of processor time to the processes. Programs that generally perform well under Hyper-Threading will still show a higher individual performance in many cases.

The scheduler based on related work, *ipc*, usually provides an unfair share of performance. This is because it favours processes with high IPC. The original algorithm would have completely starved low-IPC programs such as *181.mcf*; my implementation prevents this by utilising the Linux scheduler's own starvation avoidance functionality.

The *fair* scheduler, which allows a poorly performing process more processor time, can improve the fairness in some cases but it limited by the accuracy of the performance estimation model.

4.5.3.1 *Static Priorities*

The scheduling algorithms were designed to adjust the dynamic priority of processes in order to improve throughput or fairness. By doing this the static priority of the processes should be respected. To check this behaviour a benchmark set was run with the benchmarks each given a different priority. The benchmarks of set B were given static priorities as shown in Table 4.2. The ordering of the benchmarks was chosen to give the longest running benchmarks the higher priorities in order to keep the experiment duration reasonably short.

Figure 4.6 shows the individual performance of each benchmark in the set running under each scheduler. The bars are ordered such that the highest priority is on the left and lowest on the

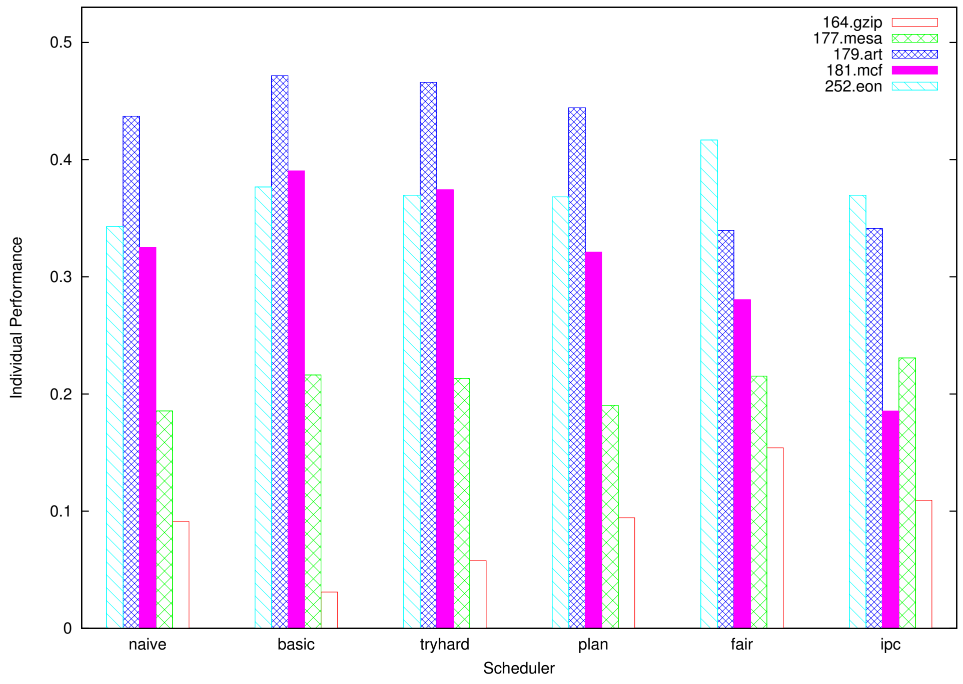


Figure 4.6: Individual performances of the benchmarks of set B running on the single package configuration (1x2). The benchmarks are given static priories resulting in time-slices of 60, 50, 40, 30 and 20ms in order left to right.

right. It can be seen that *naïve*, *basic*, *tryhard* and *plan* all allow *179.art* to achieve a performance greater than its priority would suggest. This is due to the benchmark generally performing well while always harming the simultaneously executing process. The apparent incorrect behaviour of these scheduling algorithms is therefore an artifact of the workloads involved. The *fair* algorithm attempts to correct this type of behaviour and the results show clearly how the ratios of achieved performances match the ratios of time-slices well. *181.mcf* suffers under the *ipc* scheduler because of the benchmark's low native IPC.

A second experiment representing a more common use of static priorities was performed. Benchmark set D contains an application that makes use of the X server. The default behaviour for the system under test is to start the X server with a higher static priority than most other applications. This was altered for the throughput experiments above but retained here. Figure 4.7 shows the individual performance data for the benchmarks of set D running under each scheduler with the X process (represented by the result for *glx*) having a higher static priority (by 5 "nice" units) than the other tasks. This can be compared to Figure 4.5 above where all processes were given equal static priorities.

As described above, *glx* generally gets a better individual performance in the equal-priority experiments. However, the impact of the static priority increase on the X server is clearly visible. The time-slice given to each task at the start of each scheduling epoch is based on the task's static priority added to half of any remaining time-slice from the last epoch. For the system under test the base time-slice for the X server was 6 ticks (60ms) and 4 ticks (40ms) for the other processes; this should result in a ratio of 3:2 of the individual performances of *glx* to each of the

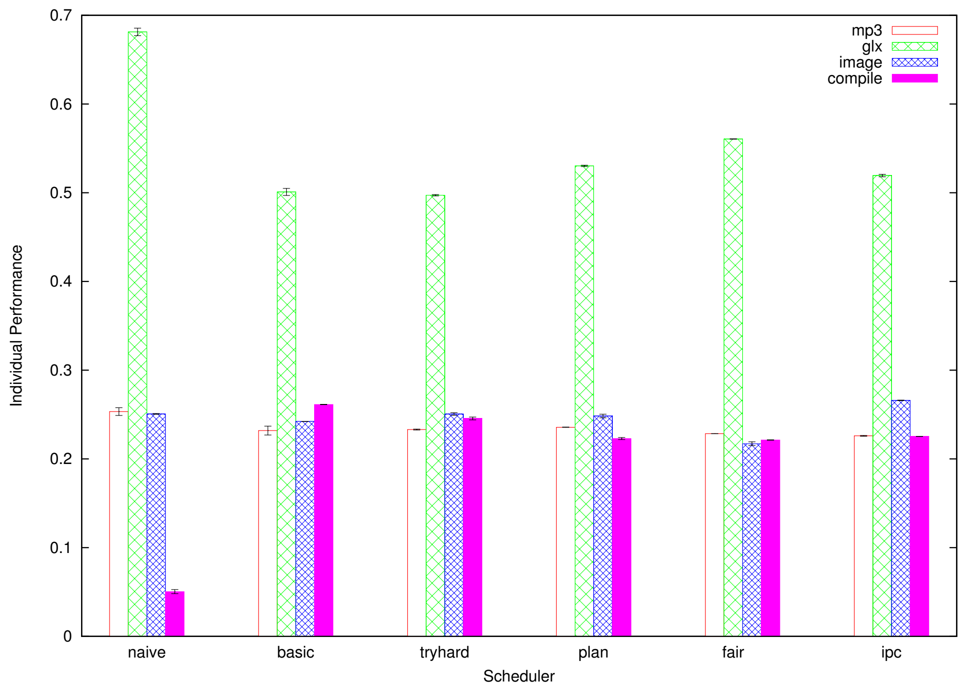


Figure 4.7: Individual performances of the benchmarks of set D running on the single package configuration (1x2). The X server process, the main part of the *glx* benchmark, has an increased static priority.

other benchmarks. The ratio is higher than this for all the schedulers which is due to the better Hyper-Threaded performance of *glx*.

These results show that the proposed scheduling algorithms are respecting static priorities.

4.6 Applicability to Other Operating Systems

The scheduling algorithms presented in this chapter were based around modifications to the calculation of dynamic priority. Many general-purpose operating system schedulers use some form of dynamic priority therefore it should be possibly to incorporate SMT-aware heuristics. The monitoring functions were based on callbacks from the context switch and timer interrupt functions; these are features found in general-purpose operating systems.

The use of processor hardware performance counters is a potential barrier to common acceptance of a scheduler of the type described here. The processor has only one set of performance counters and there may be many potential users of the counters. For application use there are counter virtualisation packages available that can "context switch" the counter configurations and values. For kernel use this is not sufficient and my schedulers require constant measurements of realised performance.

Linux 2.6 has a new implementation of the process scheduler. It differs from the scheduler in the 2.4 (used for my experiments) and 2.2 kernels mainly in its allocation of the time-slice for a process being per-processor rather than system-wide. In addition the run-queues are maintained independently for each processor. The result is that processor affinity has more of an impact and migration becomes a more explicit operation. Since there is no common run-queue, there is no need for a system-wide lock; processors can perform context switches in parallel. The goodness function has been removed, its functionality being replaced by the ordering of the run-queue and the implied affinity. It would still be possible to incorporate SMT-aware scheduling into this model. For example, careful modification of the run-queues would permit more reliable gang scheduling than was possible in the 2.4 kernel.

The scheduler in recent versions of the Windows operating system uses a single-queue preemptive scheme [Solomon00][2]. Static priorities are used with temporary priority boosts for starvation avoidance. Boosts are also given to threads associated with the foreground window and threads that have recently had a wait operation complete. Multiprocessor scheduling utilises soft-affinity data for each thread; when picking a task for a processor the scheduler searches the run-queue and executes the first (highest priority) thread that either ran last on that processor, has a static affinity for that processor, is experiencing starvation or is of a very high priority. There is ample scope for adjusting priorities and affinities to implement the various SMT-aware schedulers.

4.7 Applicability to Other Processors

The scheduling algorithms presented here relied on processor hardware performance counters. Only a small set of the available event types were counted: cache miss rates and instruction throughput. These event types are often found on other processors with less sophisticated counters than the Pentium 4.

The performance estimation function in Section 4.4.3 is specific to the particular system. The coefficients will differ with different processor cores (such as Prescott instead of Northwood) and cache arrangements. It is likely, however, that the relative magnitudes of the coefficients would be similar across similar processors. The coefficients were produced by performing a multiple linear regression analysis on data from test runs of a set of benchmarks. This is a mechanical (albeit time consuming) process which could be used to tune the model for different systems or even different workloads. Further investigation of this model is left for future work.

My work was performed using a “Northwood” version of the Pentium 4 processor. As shown in Chapter 3 the recently released “Prescott” version experiences fewer cases where the thread interactions cause a significant loss of performance. This suggests that the software techniques presented here are not as important on the newer processor. However, there are still a number of poorly performing simultaneous pairings which it would be advantageous to avoid. There is also scope for simplifying the microarchitecture of future SMT processors by dealing with resource allocation fairness in software, including scheduling techniques similar to those presented here, rather than hardware. This strategy could allow the processor to run faster, consume less energy or use less silicon.

It is likely that future SMT processors will support more threads, with four-thread units rumoured to be coming in the near future. The techniques presented here are likely to become more useful in

[2]See also Microsoft Developer Network: *Platform SDK: DLLs, Processes, and Threads*.

these processors as they may benefit from the possibility of disabling one or more threads during periods of poor performance due to unfortunate thread interactions.

4.8 Summary

I have presented a number of different scheduler implementations that are aware, to different degrees, of the characteristics of the SMT processor and try to improve throughput and/or fairness. Unlike much previous work in the area these schedulers have to deal with the complexity of a real scheduler, such as processes blocking, and preserve existing facilities such as starvation avoidance. The speedups obtained, up to 3.2%, are similar to speedups reported in related work which uses a less flexible scheme with workloads chosen to benefit from that scheme. My scheduling algorithms generally outperform an IPC-based scheme (found to be the best all-round performing scheme studied) from the simulation-based literature modified to work within the constraints of a practical general-purpose operating system scheduler.

Chapter 5

Alternatives to Multiprogramming

In the previous chapters I have focused on the scenario where a simultaneous multithreaded (SMT) processor is presented as a set of distinct logical processors which can be allocated independently. I have shown how a naïve use of this abstraction can cause a loss of performance. I have presented a design and evaluation of a range of process scheduling algorithms that try to avoid this performance loss. In this chapter I discuss alternative ways of using SMT processors which expose and exploit their multithreaded nature rather than trying to hide it. I present some preliminary experiments to assess the value of dedicating a hardware thread to system functions such as interrupt handling — a technique possible using current SMT hardware.

The techniques described here fall into two broad categories. Firstly those that present the processor as a multithreaded processor and secondly those that use the threads to improve the performance of the main thread therefore presenting the processor as a traditional uniprocessor. Some work possesses elements of both categories by using multiple processor threads to work on a single software thread but having more than one of these groups existing on the same multithreaded processor. The second category is motivated by the desire to improve the performance of an individual thread, something that SMT in a multiprogramming context is very unlikely to do. The speedups must be considered alongside the system-wide improvements offered by multiprogramming on SMT.

5.1 A Multithreaded Processor as a Single Resource

To avoid the operating system having to be concerned with the thread interactions experienced on an SMT process the problem could be given to the applications by allocating the entire processor as a single resource rather than trying to allocate threads independently. It would be up to the application to make best use of the available hardware threads. Applications would be better placed to know their own demands and, given enough information about the processor, should be able to make a more informed decision about what to simultaneously execute.

Modern compilers are able to optimise code for different models of processors implementing the same instruction set. This is done by building knowledge of the operation of the processor into the compiler. This scheme could be extended to parallelising compilers; the compiler would be designed with knowledge of the thread interactions. As with existing machine-specific optimisations, the benefits are limited or lost when the compiled code is executed on a different processor (of the same instruction set).

A possible middle-ground is to locate the processor-specific knowledge in a *thread package*. This software typically sits between the operating system and the application code and provides a common abstraction for threads. For example, the *POSIX threads* standard specifies an interface for applications to create multiple threads [IEEE04]. The threads package implements this standard by mapping the application *p-threads* to kernel or user (or a combination of the two) threads. During a scheduling quantum the operating system could allocate the entire SMT processor to the thread package and allow the package to decide on the optimal allocation of p-threads to simultaneous threads.

Capriccio is a user-level thread package aimed at highly concurrent network servers [von Behren03]. The application threads are scheduled onto operating system threads using a resource-aware scheduler; the package monitors resources such as memory and file descriptor usage and attempts to balance load over time. Adding processor resources and SMT-awareness to this package has the potential to increase throughput; the large choice of threads to run increases the chance of an optimal schedule.

5.1.1 Program Parallelisation

Any multiprocessor or multithreading system on which it is desired to run a single workload will benefit from that workload being multithreaded. Single-threaded workloads can sometimes be parallelised by a compiler. On SMT processors extra threads do not always mean better performance so the nature and extent of the parallelisation need to be chosen carefully. Puppin and Tullsen investigate the parallelisation of Livermore loops on a simulated SMT processor [Puppin01]. They note that traditional parallelisation techniques work well for SMT processors but the optimal number of hardware threads to use may not be the processor full complement. They develop a model to predict the best number of threads to use based on the contention for functional unit types and the length of dependent chains of instructions.

5.2 Threads for Speculation and Prefetching

In this section I describe proposed mechanisms that use simultaneous threads for speculation of data and control-flow. The closely related topic of *pre-execution* is also described, where a thread executes ahead of the main thread in order to prefetch data or precompute branch directions.

5.2.1 Data Speculation

Programs written in a single-threaded manner can sometimes be successfully parallelised by a compiler. Numeric applications tend to be the easiest to auto-parallelise because it is straightforward to track the data dependencies. Programs that are more reliant on memory accesses are harder to parallelise due to the potentially much larger number of dependencies through the memory locations. Superscalar processors can extract instruction level parallelism from such programs by hoisting and speculatively executing loads and checking for subsequent writes to the locations referenced. This mechanism is *instruction-level data speculation*. A natural extension to this is *thread-level data speculation* where various parts of a program, particularly loop iterations, can be executed in parallel based on speculative use of loaded values. As with instruction-level speculation, speculative threads have to be replayed if the data values used turn out to be incorrect. The main difference from instruction-level speculation is the greater degree of parallelism than can be extracted but at the expense of a higher cost in replaying incorrect executions. Thread-level speculation has also been proposed as a tool to assist manual parallelisation of programs [Prabhu03].

Multiscalar processors use a combination of hardware and software to split a single-threaded process into multiple "tasks" which are farmed out to a set of processing units within the processor [Sohi95]. Each unit fetches and executes independently and keeps a copy of the global register file. Register values are routed between units to keep the copies consistent. *Implicit-Multithreading* (IMT) [Park03] applies the Multiscalar concept to SMT. The authors found that a simple replacement of the Multiscalar execution units with SMT threads gained little advantage over single-threaded superscalar. They proposed some microarchitectural optimisations to improve the performance:

- a fetch policy biased towards threads processing earlier tasks,
- task threads multiplexed onto SMT contexts, and
- a faster thread start-up mechanism to support the large number of thread start-ups.

The result was a 20 to 30% speedup over superscalar.

Thread-level data speculation (TLDS) as proposed by Steffan and Mowry speculatively breaks a single-threaded process into "epochs" which are similar to Multiscalar's tasks [Steffan98, Steffan00]. TLDS relies on the compiler to identify regions of the code that are suitable for speculation and to perform the parallelisation. The multithreaded processor (traditional or simultaneous) is responsible for forwarding data values between threads (which can cause epochs to execute in an overlapped fashion) and to check for data violations when an epoch is completed and attempts to commit its work. Any data dependency violating epoch has to be executed again.

Marcuello and González proposed to use a modified SMT processor to provide a similar facility to that provided by implicit-multithreading described above but without the need to modify application binaries [Marcuello98, Marcuello99]. They focus particularly on having different threads executing different iterations of loops. Loop closing branches are highly predictable which allows the hardware to perform the decomposition into threads. They speculate on register values by looking at the difference in live register values over successive iterations of the loop. They report a 25% speedup for integer codes.

Akkary and Driscoll describe *dynamic multithreading* (DMT) in which the processor speculatively spawns threads on procedure and loop boundaries [Akkary98]. The goals of this work and some of the microarchitectural mechanisms are similar to Marcuello *et al*'s speculative multithreading but Akkary and Driscoll focus on spawning at procedure boundaries.

Thread-level speculation (TLS) has been used as the basis for other work. Zhou *et al*'s *iWatcher* is a an extension to traditional "watchpoints" (memory locations that when referenced cause the processor to trap into an exception for debugging purposes) [Zhou04]. The authors noted that watchpoints are generally used to check constraints on a set of variables. In normal operation with no bugs existing, the check will always succeed and execution will resume following an expensive exception. iWatcher's main contribution is the use of TLS to allow the main thread to continue speculatively while the watchpoint handler runs in another thread context. The original thread is only allowed to become non-speculative when and if the handler declares the check successful. As well as the reduced impact on the throughput of the main thread, this design avoids expensive privilege level changes in a similar manner to Zilles' work described below.

5.2.2 Pre-execution

Pre-execution and *precomputation* are similar techniques used to reduce the latency of load instructions or find the outcome of a branch in advance of where it would normally be known. This is done by a second computation proceeding alongside the main thread speculatively executing some subset of instructions. The effect is either that the value to be loaded is available in the cache for when the main thread executes the load, or the branch outcome is known when the main thread comes to predict that branch. Mechanisms have been proposed to use dedicated hardware, such as a "precomputation engine", for the secondary computation. Combining pre-execution and SMT should yield good results; spare thread contexts can be used to run the speculative secondary computations(s) which can share the contents of the cache and enjoy fast register sharing or copying. Pre-execution differs from data speculation described above in that the actual architecturally important execution is carried out by a thread in the traditional way with that thread being itself speculative at the thread level.

Collins *et al* use precomputation to prefetch "delinquent" loads (the small number of static loads that generate the majority of cache misses) [Collins01b]. The compiler identifies the delinquent loads and for each, generates a sequence of instructions called a "p-slice" which is able to compute the address and prefetch the data. The processor spawns the p-slice when it encounters a designated trigger instruction in the main thread. They base their simulation on an SMT implementation of the Intel IA64 instruction set. In later work Collins *et al* remove the reliance on the compiler by moving the identification of delinquent loads and the construction of p-slices to hardware to allow unmodified binaries to benefit from the technique [Collins01a]. They also move architecture to the Alpha-based SMTSIM. The authors report a wide range of speedups in both papers with about 30% being the most reasonable figure in both and up to 169% in the compiler-assisted version when speculative threads are allowed to spawn further speculative threads.

Intel implemented a version of precomputation on a pre-production Pentium 4 Xeon with Hyper-Threading. Since this processor has no hardware support for speculative precomputation a separate heavyweight operating system thread was forked at initialisation of the main thread and the Windows XP event mechanism was used for the main thread to communicate trigger events to the speculative thread [Wang02]. The program binary was statically modified based on performance data obtained using Intel's VTune Performance Analyser. Speedups of up to 45% were observed on pointer-chasing synthetic benchmarks and between 7% and 40% on a selection of application benchmarks expected to benefit from precomputation. Perhaps the most interesting aspect of this work is the demonstration that the techniques of precomputation can be employed without any hardware modifications.

Speculative slices as proposed by Zilles and Sohi [Zilles01] are similar to p-slices. They are created manually in order to prefetch loads or precompute hard-to-predict branches. Speculative slices differ from p-slices in that each execution of a slice is tied to a particular dynamic branch/load (i.e. a particular instance of the instruction). The authors argue that this is necessary, particularly for branches, as the behaviour will change between dynamic instances.

In follow-up work to their Hyper-Threaded study described above, Intel use some of the techniques of Zilles and Sohi's work in order to provide a prefetching helper thread [Kim04]. They discuss the problem of dynamic runtime tuning of the helper threads including how to remove unnecessary prefetches causing wasted processing. They note that the Intel Hyper-Threaded processors do not currently have sufficiently fine-grained and lightweight synchronisation to allow such tuning.

Software-controlled pre-execution differs from Collin *et al*'s precomputation in that it executes a copy of the original code spawned from compiler/profiler or programmer inserted points [Luk01]. The pre-execution thread proceeds without changing any global state, other than causing data to be prefetched into the cache, by not committing stores or generating exceptions. This thread continues for a determined number of instructions or until a determined point in the code. The author reports an average speedup of 24% across a set of "irregular" applications (those that would most benefit from this technique).

Speculative Data-Driven Multithreading uses spawned speculative threads to work ahead in the program code to execute "critical computations" [Roth01]. The *data-driven thread* executes only those instructions from the main program that are required to perform the particular computation therefore allowing it to proceed quicker than the main thread. The result and input used are recorded in an "integration table" to allow the main thread to use the generated result if the speculation was correct thereby saving on duplicate work. Simulation on an 8-way Alpha based SMT architecture yielded execution time savings of up to 45%.

Slipstream processors run a shortened copy of the application speculatively. This shortened copy contains only the instructions that the processor believes are required for correct forward progress [Sundaramoorthy00]. The design is based on the observation that there is redundancy in program code which includes unreferenced writes, non-modifying writes and correctly predicted branches. A full execution (the "R-stream") of the program runs in chase with the shortened version (the "A-stream") passing control-flow and dataflow outcomes back to it for checking. The A-stream is functioning as a prefetch thread for the R-stream with the latter executing all instructions and producing the definitive results. The design was originally evaluated on a simulated chip multiprocessor (CMP) and later work considered the use of SMT [Purser00]. A 2-core CMP design was shown to give an average improvement of 12% over a single core and SMT provided a speedup of 10 to 20% compared to an equivalant non-SMT architecture.

A good example of a use of pre-computation that would be applicable to the SMT-based designed described above is Roth *et al*'s virtual function call ("v-call") target prediction [Roth99]. V-calls are typically used in object oriented programs to dynamically select the function to call at runtime based on type information. Unlike static calls, the jump target is hard to predict using conventional mechanisms but easy to predict using the pre-computation methods described above.

5.2.3 Multiple Path Execution

The technique of multiple path execution involves the processor proceeding along both possible execution paths from a branch until the outcome of that branch is known. This reduces the cost of a mispredicted branch at the expense of wasted execution effort. The technique was used, to a limited extent, in the IBM System/360 Model 91 where the first few instructions from both branch paths were fetched (but not decoded) [Anderson67]. The hardware cost of multiple path execution can be high; the use of spare processing capacity in a multithreaded processor would allow some of the benefit of multiple path execution to be obtained without the hardware overhead.

Wallace *et al* propose a technique for making use of spare thread contexts within an SMT processor, *Threaded Multiple Path Execution* [Wallace98]. When the execution of a thread reaches a hard-to-predict branch the processor performs an internal "fork" operation with the two threads following the different paths from the branch. Once the outcome of the branch is known, the wrong-path thread can be squashed and the thread context reused for further speculation. Clearly the processor must be careful about when it chooses to fork to prevent an exponential increase in the number of required contexts very quickly consuming the fixed number of available contexts. Wallace *et al* denote one path from each fork as the *primary path* and only allow further forks from that path. This not only removes the problem of exponential growth but also simplifies the implementation. They also limit forks to branches with a low *confidence* by using a branch confidence predictor which tracks how well prediction is working.

The architectural to physical register mapping used by SMT lends itself to lightweight forking making the rollback of the register state of a wrong-path thread easy due to only having to discard mappings and update the physical register free list. As with instruction level speculation, it is memory accesses that complicate the rollback. Wallace *et al* propose two elaborate schemes to track load-store dependencies using a combination of store buffers and tagging of speculative stores.

Threaded multiple path execution could be applied to other forms of hardware multithreading processors but SMT is particularly suitable as the speculative paths are using spare capacity rather than dedicated resources.

5.3 Threads for Management and Monitoring

In this section I describe proposals to use simultaneous threads, executed at a lower hardware priority, for purposes other than application threads. Uses include monitoring for optimisation and error-checking.

5.3.1 Mini-Threads

Redstone notes that a multi-context SMT processor would require a large number of registers [Redstone02]. With a large instruction window and a large register set (such as the Alpha), the size of the register file would cause either a long cycle time or an increase in the pipeline depth. Redstone proposes to extend SMT to support *mini-threads* [Redstone03]. In this architecture the thread context is split into the architectural registers and the remaining (mainly control-flow) state such as the program counter, return stack, store buffers and so on. A mini-thread has its own control-flow state but it shares the architectural registers with the other mini-threads within the context. This approach reduces the number of physical registers required and provides a cheap inter-mini-thread communication mechanism. The disadvantage of the reduced number of architectural registers per mini-thread is generally outweighed by the increase in throughput due to the extra thread-level parallelism. The allocation of architectural registers to mini-threads is left to the compiler which could choose to partition the threads statically or dynamically or to share registers where appropriate.

A problem with this design is that calls to the kernel could be made by any mini-thread and in the general case the kernel would not know which registers were being used by that mini-thread at that time so would not know which registers could be saved for its own use. A reasonably straight-forward fix described by Redstone is to force static partitioning of registers and have the instruction decode stage rewrite the register numbers for the particular mini-context being fetched (i.e. for a two mini-thread scenario the second mini-thread has the top bit of each register number set by the decode stage). This rewriting also applies to kernel code so an unmodified kernel can be used. Redstone suggests an alternative for when static partitioning is not in use; when a mini-thread enters the kernel, all other mini-threads are blocked and all registers are saved and restored when the control returns to the user-mode code.

Redstone evaluated the architecture using parallel benchmarks and applications and showed a mean speedup of 38% on a 2-context SMT processor. Processors with higher numbers of contexts yielded less speedup but are the ones that are is most need of a smaller register file. A use not considered was executing subordinate threads such as profiling or monitoring functions. This would be a suitable application for mini-threads as they can be written to use only a small number of the registers and would benefit from being able to read (and maybe write in some circumstances) the registers of the main thread.

5.3.2 Subordinate Threads

Subordinate (helper) threads utilise simultaneous execution in a biased manner to provide some form of support to the main thread. The main thread is generally allowed to use all the processor resources it needs while the subordinate threads use whatever remains. Subordinate threads can perform functions such as profiling, optimisation and prefetching. Subordinate threads differ from pre-computation in that they are explicitly provided by either the application or the operating system rather than being automatically extracted from a program.

Dubois and Song propose *nanothreads* as a mechanism to assist the execution of the main thread [Dubois98]. Nanothreads share all the resources, including most of the registers, of the main thread. Additionally, each nanothread has its own small number of architectural registers. Nanothreads can be spawned explicitly by the main thread to provide facilities such as prefetching. Alternatively a *nanotrap*, a lightweight trap based on selectable hardware events such as a cache miss or invalidation, can spawn a nanothread. The authors suggest that this mechanism may be useful in circumstances where an exception would traditionally be the only suitable mechanism but too expensive for high frequency events. A nanotrap can be asynchronous (the main thread continues) or synchronous (the main thread blocks until the nanothread servicing the nanotrap completes).

Chappell *et al* note that SMT is of little use to single-threaded programs [Chappell99]. To provide a benefit to an individual thread they propose *Simultaneous Subordinate Microthreading* (SSMT) in which an SMT-like processor hosts a single program thread and a number of lower priority subordinate microthreads used to perform hardware optimisations on behalf of the program thread. The microthreads are spawned either explicitly by the program thread using a new processor instruction or automatically by certain events taking place within the processor. The microthreads' instructions are stored in a "MicroRAM" within the processor which reduces interference with the main thread's instruction fetching. Chappell *et al* suggest possible uses for these microthreads which include branch prediction optimisation, prefetching and software cache management. They suggest that such software based schemes can outperform purely hardware based equivalents because more sophisticated algorithms can be implemented in software than in hardware and there is greater flexibility with the ability to tune the microthread implementations for different applications. Microthreads are preferable to normal system threads for the optimisation purpose because they do not contribute to cache utilisation and can be given greater access to processor internals without being constrained to the processor's instruction set architecture.

Chappell *et al* focus on increasing the performance of a single program. An SSMT system wishing to execute more than one program would have to context switch between them (which would also involve context switching the microthreads) whereas an SMT system with sufficient thread contexts would be able to simultaneously execute the programs. The performance tradeoff between SSMT and SMT is therefore individual programs running faster but only having a share of the processor time each, versus individual benchmarks running continuously but slower due to sharing the processor. The SSMT designers do not address this issue. It is likely that some implementations and workloads would benefit more from SSMT and some more from SMT.

Chappell *et al*'s work requires specific hardware. It is worth exploring the feasibility of using some kind of subordinate threads for optimisation and system functions on current hardware. The main disadvantage of Intel's Hyper-Threading in this context is the lack of any form of prioritisation. This means that a "subordinate" helper thread could very easily cause a performance loss to the main program thread while trying to optimise it or perform some other service. The design of such a thread would have to be very careful and would have to avoid actions that are known to affect the performance of the other thread. A major problem would be instruction and data cache interference, including cache flushing due to self modifying code (something an optimisation thread may well want to do). In order to provide any optimisation service the subordinate thread would have to have suitable access to appropriate processor internal information. The current processor performance counters could be useful but a richer, lower level of access would be required to be able to replicate the work of Chappell *et al*. A further complication is forking and joining. Hyper-Threading originally had no support for any form of inter-thread communication other than the use of shared memory as would be used by multiple processors. The Intel "Prescott" core, the second implementation of Hyper-Threading, introduces the MONITOR and MWAIT instructions as described in Chapter 2. These instructions provide a mechanism to synchronise threads.

Dorai *et al*'s *Transparent Threads* are a general purpose variation of subordinate threading. Various priority mechanisms are added to an SMT processor to allow background ("transparent") threads to run without affecting the performance of the foreground thread [Dorai02]. The priority is implemented by a collection of mechanisms. Instruction fetch and issue slots are prioritised; in each cycle, a slot is only allocated to a background thread once the foreground thread has taken all the slots it can. The instruction buffers (reorder buffer and pre-issue instruction queues) are difficult to allocate according to priority because the effect on resource contention is delayed. The paper describes a technique called "background thread instruction window partitioning" which imposes an ICOUNT (hardware rate-limiting of each thread's fetching) limit on the background threads' fetching in order to limit their population of the instruction buffers. This technique does not completely prevent the foreground thread from suffering resource contention but does limit the damage that a background thread can do. The authors also describe "background thread flushing" where the most recently fetched instructions from the background threads can be flushed from the reorder buffer and the program counters wound back. This action will be carried out when the foreground thread is unable to obtain a slot in the reorder buffer. The work focuses on pipeline resources but suggest that access to caches could be biased by limiting the amount of each cache that a background thread can use by modifying the address hashing function in use.

Dorai *et al* suggest the following uses for transparent threads.

- Downgrading the priority of non-interactive threads when an interactive thread on that processor is receiving an event. The interactive thread gets all of the resources it needs while the other threads can continue in the background using any available resources.
- Subordinate multithreading, in a manner similar to SSMT described above or for prefetching and pre-execution.
- Performance monitoring for profiling the foreground thread, without the need to introduce invasive instrumentation code.

In their simulated prototype the authors found that the average performance degradation to foreground tasks while running some transparent prefetching threads was only 3%. They suggest that two-thirds of this value is due to cache effects which their design does not deal with. The background prefetching thread ran at 77% of the speed it would have done with an equal priority scheme and its prefetching functionality gave the main thread a speedup in the region of 10%, a net gain.

Oplinger and Lam propose using an SMT-based architecture to allow programmers to write code monitoring and recovery functions that can be speculatively executed in parallel with the main thread [Oplinger02]. Examples of these functions include profiling and run-time error checking such as checking the stack for buffer overruns. The design supports efficient transactions. A transaction can be implemented by the processor making a local copy of the register state before speculatively proceeding with the thread. If the thread decides to commit then the buffered stores are performed and execution continues, otherwise the registers are restored and execution continues from the specified address.

5.4 Fault Tolerance

In recent years a body of research has focused on techniques to support hardware that may contain or experience faults. Traditionally digital hardware is assumed to perform perfectly if operated with specifications. With modern processors having very tight margins on voltage levels, feature sizes and noise, the risk of a transient fault, caused by random effects such as cosmic rays, is becoming greater (albeit very small at the moment). Research on how to handle this potential future problem generally uses redundancy in some form to check the results of executed code. Space redundancy (the use of replicated hardware) has obvious financial and energy costs, time redundancy (re-executing again on the same hardware) takes longer, and data redundancy (the use of check bits, CRCs, etc.) introduces extra complexity.

Rotenberg [Rotenberg99] proposes a modification of SMT called *AR-SMT* which allows two copies of a program to be run approximately at the same time. There is a notion of an active thread, the "A-stream", followed shortly by the redundant thread, the "R-stream". The R-stream lags the A-stream in time providing time redundancy. A delay buffer is used to feed results from the A-stream to the R-stream for comparison — this technique is similar to slipstreaming described above. The R-stream can enjoy perfect control and dataflow prediction because it uses these details as passed back by the A-stream (this does not reduce any of the redundancy as the instructions are still executed). This design is aimed at general purpose processors where traditional fault tolerant systems are too costly or inefficient. The technique would suffer somewhat on a processor using the Intel Hyper-Threading style of multithreading where homogeneity can cause a notable reduction in the combined throughput of the threads (see Chapter 3).

The technique is further refined by Reinhardt and Mukherjee's *simultaneously and redundantly threaded* (SRT) processor by considering where redundancy needs to be provided in more detail and allowing more slack in the fetch rates of the two threads [Reinhardt00]. SRT is only a detection mechanism; software checkpointing must be used alongside it in order to be able to backtrack to a point before the fault. Vijaykumar *et al* modify SRT to build in the recovery functionality [Vijaykumar02]. SRTR (SRT with recovery) will prevent a faulty operation from changing global state by ensuring that instructions from the leading thread can only commit when the trailing redundant thread agrees on the correct result. This method avoids the need for software checkpointing which improves performance when the rate of faults is low and allows unmodified software to be used.

5.5 Operating System Functions

In this section I describe related work and present some preliminary investigations into using simultaneous threads to improve the performance of operating system functions. The particular theme is the avoidance of changing privilege levels, a mechanism known to be expensive on modern processors.

5.5.1 Exception Handling

Zilles *et al* propose the use of multithreading, using SMT as the particular example, for exception handling [Zillies99]. When executing on a speculative superscalar processor a thread experiencing a recoverable exception, such as a software translation look-aside buffer (TLB) miss, will undergo a sequence of operations similar to the following.

- Instructions up to and beyond the faulting instruction will be fetched.
- Some non-dependent instructions beyond the faulting instruction may have been speculatively executed.
- The faulting instruction is executed and the exception noted to be raised when the instruction is retired.
- On retirement of the faulting instruction the exception is raised by changing the control flow to the exception handler and flushing the fetched, and possibly speculatively executed, instructions that followed the faulting instruction.
- The exception handler executes and its return-from-exception instruction again causes a control flow change back to the originally faulting instruction.
- Fetching continues from the originally faulting instruction and execution continues.

Note that there is duplicated work in that instructions following the faulting instruction are fetched and some speculatively executed twice. During the two control flow changes there will be periods of idleness for the core execution units while the new instruction sequences propagate through the fetch, decode and rename stages.

Zilles *et al* suggest that, for software TLB miss exceptions at least, the non-dependent instructions beyond the faulting memory access could proceed and instructions dependent upon the memory access could wait in the normal dataflow manner. If the exception was to be handled in an out-of-band manner then on the handler's completion the faulting instruction would be allowed to proceed. The execution would then carry on as normal with the originally faulting instruction's completion allowing the dependent instructions to be scheduled. Zilles *et al* propose the use of a new hardware thread context for the exception handler with a small amount of extra hardware to communicate completion of the handler back to faulting instruction. This mechanism removes the duplicate work described above and allows the main thread to continue with any non-dependent work.

Should the exception be non-recoverable then the handler communicates the fact back to the faulting instruction which then causes the thread to terminate when the faulting exception is retired. As with control flow misprediction any speculatively executed instructions beyond the faulting instruction are not retired so no architectural state is incorrectly affected.

5.5.2 Privilege Level Partitioning

Interrupts can be quite disruptive to applications running on aggressive out-of-order superscalar processors. This is due in part to the high cost of (re)filling the instruction windows for both the interrupt handler and the user mode application. The time spent in the interrupt handler is of course taken away from the application's CPU time allocation. The net effect is that providing any sort of quality-of-service guarantee to an application is difficult or impossible.

Muir describes an alternative arrangement as part of his *Piglet* architecture [Muir00] One processor in a two way SMP system is dedicated to operating system functions including interrupts handling. Applications execute on the other processor. Whilst this mechanism provides a speedup to individual applications it comes with a high hardware and energy consumption cost. The same technique applied to the two logical processors of a Hyper-Threaded processor would be less costly to implement as no extra physical processor or other hardware would be needed. A Hyper-Threaded implementation would not provide as much isolation as Muir's SMP implementation because of the dynamic sharing of resources by the interrupt handler and the user level application. When not servicing an interrupt the system Hyper-Threaded must sleep using the `HLT` instruction in order to give all shared resources to the application Hyper-Thread.

Piglet also provided for asynchronous system calls. The user level process running on one processor would used shared memory message passing to cause the system call to be executed on the OS processor.

In his design for a multithreaded processor Moore proposes the use of a remote procedure call interface for user threads to communicate with the operating system [Moore96].

The *Xen* virtualisation infrastructure [Barham03] divides a computer between multiple "guest" operating systems (OSs). Xen abstracts the hardware interfaces by providing virtual interfaces to guest OSs and hosting device drivers within the *hypervisor* or a per-machine device driver domain. Interrupts taken by the hypervisor or device driver domain while a guest OS is scheduled will consume CPU time allocated to the guest OS. In an effort to reduce this effect Fraser *et al* propose that interrupts are handled by one logical Hyper-Threaded processor while the guest runs on the other logical processor [Fraser04].

5.5.3 Interrupt Handling Partitioning on Linux

In order to assess the impact of the interrupt handling partitioning technique on a commodity operating system I performed some preliminary experiments using the Linux kernel.

Every process metadata structure contains two useful bitmasks, `cpus_allowed`, denoting processors it is allowed to be executed on, and `launch_policy` which initialises the `cpus_allowed` field for child processes. The "`init`" process was given a `launch_policy` specifying only the first logical processor of each physical package. As all other processes as descendants of `init`, all were created with this restriction. The scheduler is able to migrate processes between allowed

Config	Physical Processor 0		Physical Processor 1	
	Hyper-Thread 0	**Hyper-Thread 1**	**Hyper-Thread 0**	**Hyper-Thread 1**
UP	Both	Disabled	Disabled	Disabled
SMP	Both	Disabled	Both	Disabled
1x2	Both	Both	Disabled	Disabled
2x2	Both	Both	Both	Both
PGLT	User app.	Disabled	IRQ	Disabled
1xHTP	User app.	IRQ	Disabled	Disabled
2xHTP	User app.	IRQ	User app.	IRQ

Table 5.1: Configurations for interrupt/application separation experiments. "Both" denotes the traditional arrangement of a processor serving both user level processes and interrupt handlers; "user app." and "IRQ" denote processors dedicated to the task of running user-level processes and handling interrupts respectively.

processors (i.e. the first logical processor on each package in a multi-package system). Linux also supports *IRQ affinity* (subject to a cooperative IRQ controller) using a similar bitmask arrangement for each IRQ number. IRQ handling was limited to the second logical processor on each package by setting the `smp_affinity` bitmask for each interrupt through the `/proc/irq/` tree. The round-robin allocation of interrupts to allowed processors is still able take place but only to the second logical processor in each package. Since no user-level process were allowed to execute on the second logical processors these processors executed the idle task when not handling interrupts. Linux uses the `HLT` instruction for idling which causes the process to dedicate all previously shared and partitioned resources to the active thread.

A series of workloads were executed on seven machine configurations as shown in Table 5.1. *UP* is a baseline traditional uniprocessor configuration. *SMP*, *1x2* and *2x2* represent the useful combinations of SMP and Hyper-Threading. *PGLT*, *1xHTP* and *2xHTP* apply application/interrupt separation to *SMP*, *1x2* and *2x2* respectively. *PGLT* represents the original Piglet configuration.

5.5.3.1 *Linux Kernel Compilation*

A commonly used application benchmark is a build of the Linux (or BSD) kernel. A large compilation like this contains compute, memory and disk-bound elements so is a good all-round test. Unlike standardised benchmarks such as the SPEC suite [SPEC], the variation in build and kernel configurations means that the results are only meaningful as trend indicators on the actual system and configuration tested.

Two different kernels were built. The first was the Linux 2.4.19 kernel and its modules as used on the test machine. This kernel was configured with only the facilities and drivers needed for that particular machine therefore was fairly small. Each experimental run of this build was performed with warm caches. The second kernel was Linux 2.6.7, the newest Linux kernel at the time of the experiment. This kernel was configured with almost all of the available facilities and drivers and the experiments were performed with cold caches. The two different configurations were used to provide different example workloads, one more processor and memory bound and the other more disk bound. It was not intended to compare the performance of the two different configurations against each other.

For each kernel a build of the kernel itself and the modules was performed on each processor configuration. Each test was performed with single-process and parallel builds (`-j` option to `make`). For the 2.4.19 build the caches were initially warmed with a build and each subsequent experiment was separated by a `make clean`. The build of the 2.6.7 kernel was large enough that it flushed the processor and OS block caches by itself. Each experiment was separated by a `make mrproper` (and a reinstatement of the kernel configuration file) to provide a clean build tree.

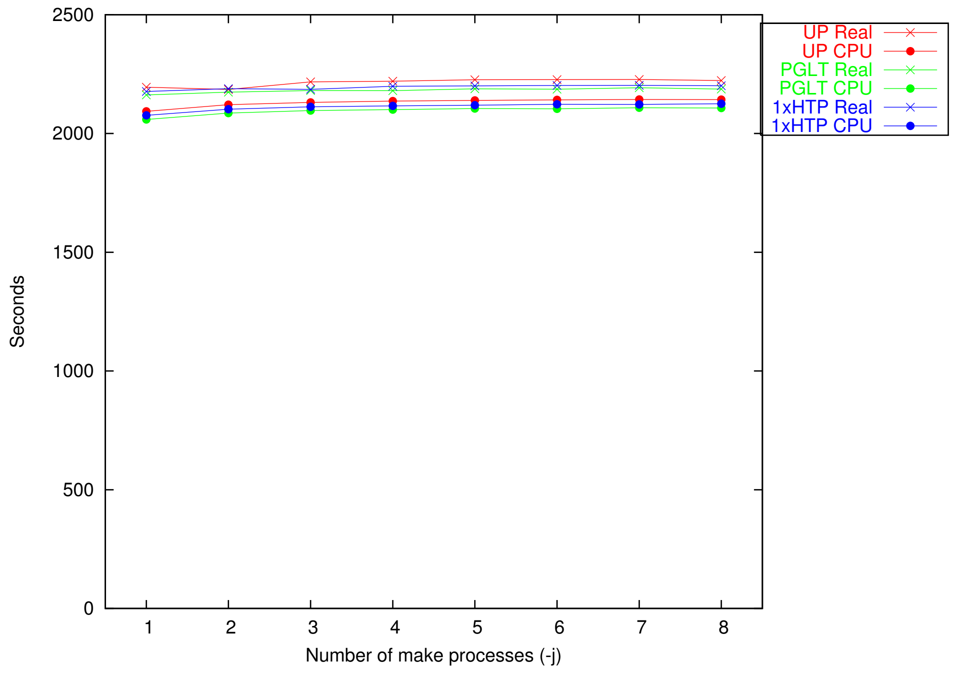
The results of the experiments can be interpreted from three points of view, relating to the invariant parameter: the effect of using parallel builds, the effect of removing interrupt processing from the application processor(s), and the tradeoff between using a given number of processors for multiprogramming or for partitioned application and interrupt processing. The first of these was described earlier, in Section 3.3.3.

The second interpretation of the data fixes the number of processors that are used to execute applications and adds a logical or physical processor to handle interrupts. Figure 5.1(a) shows the build times for the larger Linux 2.6.7 kernel for the basic uniprocessor *UP* and for two configurations where there is a single processor for applications with interrupt processing partitioned to another physical processor (*PGLT*) and to another logical processor on the same package as the application processor (*1xHTP*). These results show that there is little difference between the configurations although the Hyper-Threaded partitioned configuration does slightly better than the uniprocessor configuration and the multiprocessor partitioned configuration does slightly better than the Hyper-Threaded partitioned configuration. The differences are only in the order of 1%. Figure 5.1(b) shows the comparison of a dual processor system (*SMP*) with a dual processor, each of two logical processors system with interrupts and applications partitioned on each physical processor (*2xHTP*). This graph shows almost no different between the build times of each configuration.

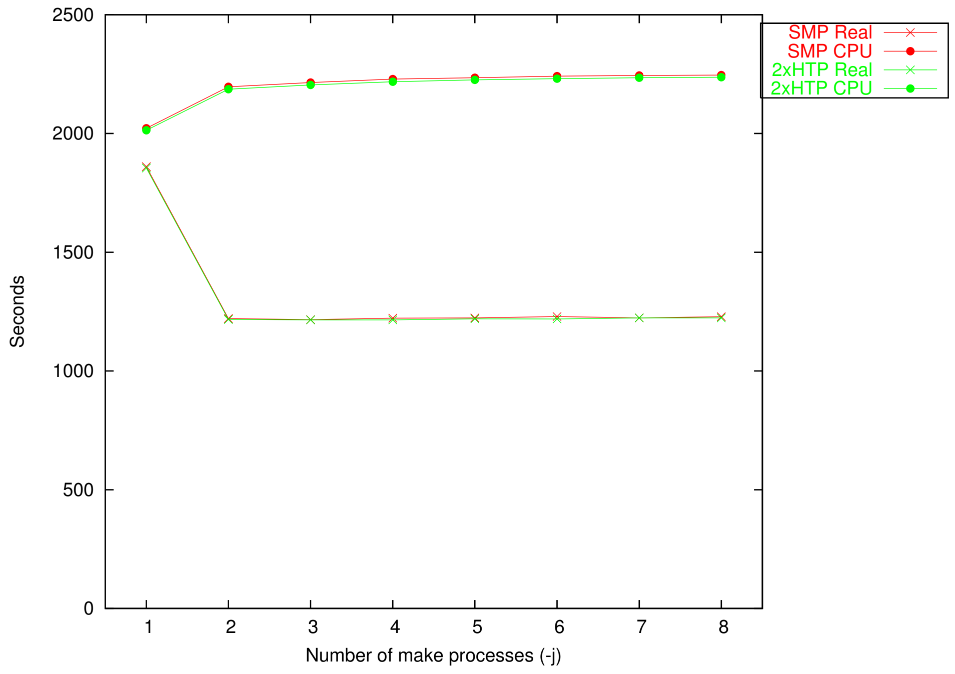
These results do not take into account the cost of providing the partitioning (which would be considerable in the multiprocessor *PGLT* case) or the loss of any multiprocessing benefit by not using the other logical/physical processors for application processing. This second aspect is considered next.

The final view of the data uses the invariant of the number of logical and physical processors. This view allows a comparison of the performance of partitioning compared to using the processors for multiprogramming. Figure 5.2 shows comparisons of the build times for Linux 2.6.7 for three different processor counts: two physical processors, two logical processors on a single physical processor, and two logical processors on each of two physical processors. All three scenarios show multiprogramming winning over partitioning. Therefore, for this particular workload, the gain from multiprogramming is greater than that from partitioning. The two physical processor case unsurprisingly shows the largest difference. The two Hyper-Threaded scenarios, particularly the two package *2xHTP*, show less of a difference. This is due to the fairly similar speedups seen for each technique independently.

These results suggest that the form of interrupt/application partitioning being investigated is not



(a) Single application processor.



(b) Two application processors.

Figure 5.1: Build times ("Real" wall-clock time and CPU time) for the Linux 2.6.7 kernel: the effect of partitioning interrupt processing to other logical or physical processors.

Config	SPEC Run time seconds	TCP rate MB/second
UP	10165	99.0
PGLT	8507	92.2
1xHTP	8842	90.4

Table 5.2: Results for the SPEC CPU2000 runs while sinking a high bandwidth TCP stream.

worthwhile for this workload but would not cause much of a performance loss if partitioning was found to be useful for other workloads. Although a reasonable amount of time is spent executing in the kernel most of this is "top-down" rather than in the interrupt context. The latter is mainly restricted to notifications of the completion of direct-memory-access (DMA) transfers — a relatively low cost.

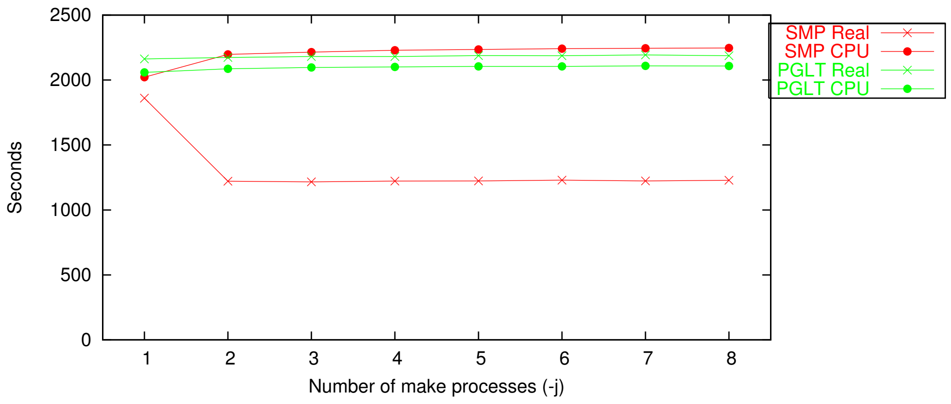
5.5.3.2 *Compute and Network Workloads*

The partitioning of interrupt and application processing should be most effective when the system is running mixed workloads of compute bound processes and interrupt-heavy I/O. To demonstrate this the SPEC CPU2000 benchmark suite was run while receiving data over a TCP stream with `ttcp`. The SPEC run consisted of running each benchmark once (with cold caches) in sequence (the Fortran-90 programs were not used due to the lack of an available compiler).

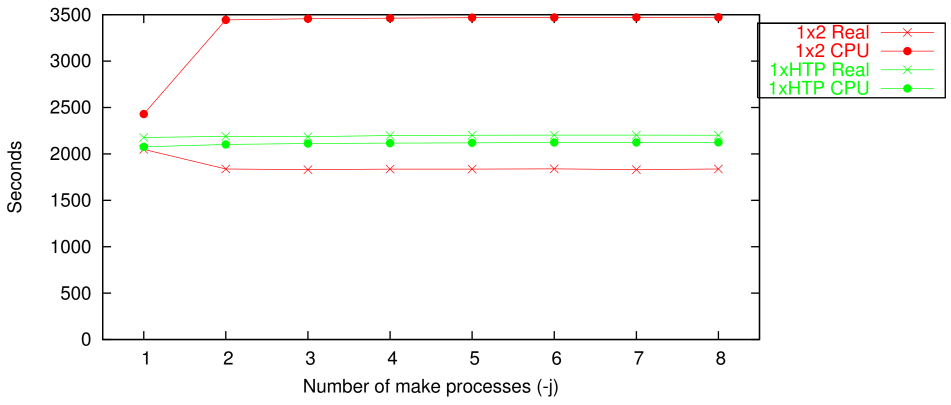
`ttcp` is a utility that can source or sink TCP streams. A receiver was run on the test machine and a transmitter on an identical machine, both connected by gigabit Ethernet to the same switch. Default parameters were used with the exception of the number of buffers to be transmitted which was set to a large value to ensure traffic flowed for the entirety of the SPEC run. The transmitter was forcefully stopped when the SPEC run completed. The `ttcp` receiver was executed on the same application processor as the SPEC benchmarks. In the partitioned configurations the kernel-mode I/O processing would have been split between the processors with "top-down" system call context processing occurring on the application processor and "bottom-up" interrupt context processing on the interrupt processor. This highlights the main difference between these experiments and Muir's Piglet — the latter uses asynchronous message passing between the processors to perform system calls. Linux aims to minimise the amount of work performed by a non-preemptable interrupt handler. Instead work is deferred to a bottom-half handler ("tasklet" or "soft IRQ"). Tasklets are scheduled on the processor that executed the (interrupt handler) code that created them. The result is that soft IRQs are executed on the interrupt processor in Piglet-style configurations.

To assess the impact of application/interrupt partitioning on the single-threaded SPEC run configurations with one application processor were tested. These are *UP*, the basic uniprocessor configuration, *PGLT*, the Piglet-style configuration and *1xHTP*, the Hyper-Threaded version of Piglet. The results for the SPEC run time and TCP transfer rate are shown in Table 5.2.

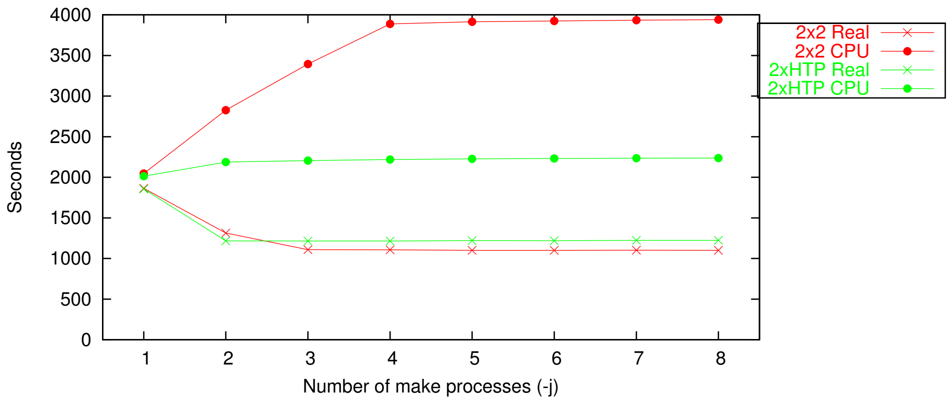
The results show that partitioning improves the performance of the SPEC execution. For *PGLT* this improvement is primarily due to the significant amount of CPU time used for TCP reception being moved to the other physical processor therefore giving the SPEC programs a greater share of their processor. This configuration shows a reduced rate of TCP transfer than *UP*; this is most



(a) Two physical processors.



(b) Two logical processors on a single physical processor.



(c) Two logical processors on each of two physical physical processors.

Figure 5.2: Build times ("Real" wall-clock time and CPU time) for the Linux 2.6.7 kernel: comparing the performance of a given set of processors used in multiprogrammed and partitioned configurations.

	httperf		SPECweb99
	Response time	Transfer time	Throughput
Config	milliseconds	milliseconds	kbits/sec
UP	1.20 ±0.10	44.8 ±1.8	348.8
SMP	1.00 ±0.00	37.0 ±1.4	396.7
1x2	1.10 ±0.00	35.1 ±0.3	378.2
2x2	1.00 ±0.00	34.4 ±0.5	395.5
PGLT	0.90 ±0.00	41.2 ±0.6	355.2
1xHTP	0.95 ±0.05	36.5 ±0.6	352.7
2xHTP	0.90 ±0.00	35.1 ±0.5	396.9

Table 5.3: Response and transfer times for fetches of a 1MB file using HTTP.

likely due to `ttcp` and its associated I/O processing being spread over two processors and therefore suffering cache-line ping-pong. The Hyper-Threaded *1xHTP* also shows an improvement over the basic uniprocessor case. The improvement is less than that achieved by the dual package *PGLT* because the interaction between the interrupt processing and the SPEC execution reduces the effective processor throughput for each Hyper-Thread. *1xHTP* shows a lower TCP transfer rate. This is caused by `ttcp` suffering the same thread interaction performance reduction.

5.5.3.3 *HTTP Serving*

The *httperf-0.8* tool [Mosberger98] was used to measure the performance of the Apache web server version 1.3.27 running on the test machine. A 1 megabyte file was repeatedly requested at a rate found to saturate the gigabit network connection between the identical client and server test machines. The request rate was 112 requests per second giving a network throughput of 939 megabits per second. Each test was performed with 2000 HTTP connections made to the server at a fixed rate of 112 requests per second.

Table 5.3 shows the initial response and total transfer times for the 1 megabyte file with the server running on each machine configuration. The results show that any hardware parallelism improves both times. The response time benefits from all Piglet-style configurations although the magnitude of this gain is very small compared to the total transfer time. Both *PGLT* and *1xHTP* configurations show a reduction in transfer time over *UP*: 8.0% and 18.5% respectively. The Hyper-Threaded version performs better than the original Piglet version because of the shared caches. Comparing the performance of configurations using one physical processor shows the multiprogrammed *1x2* slightly outperforming the Hyper-Threaded Piglet-style *1xHTP*, both being much better than *UP*. With two physical processors multiprogramming again performs marginally better than the equivalent Hyper-Threaded Piglet *2xHTP*. The traditional SMP arrangement performs better that the wasteful *PGLT* configuration.

Throughput was tested using the SPECweb99 benchmark [SPEC]. This benchmark tests the performance of the web server hardware and software. It consists of a generated set of files to be served statically and a CGI script to be used dynamically for both GET and POST HTTP methods. The benchmark provides the client software which chooses the type of requests made according to a prescribed distribution. SPECweb99 results are normally presented as the number of simultaneous HTTP connections that the server was able to maintain with at least 95% of the connections having an average bit-rate of at least 320 kbits/sec. Time constraints did not permit a full evaluation therefore tests were performed using 60 simultaneous connections (a number found to provide good bit-rates) and the achieved bit-rate reported. The full test is left for future work.

The bit-rates achieved are shown in Table 5.3. It can be seen for the single physical processor configurations that a small increase in throughput is possible with both *PGLT* and *1xHTP* over the uniprocessor *UP*. The multiprogrammed *1x2* configuration provides the highest bit-rate — the dynamic content generation used by SPECweb99 means that there is plently of work to keep both logical processors busy. The two physical processor configurations all show a similar throughput because they are constrained by system-wide resources such as disk IO and the buffer-cache.

Both sets of results show interrupt/application partitioning provides an advantage over traditional uniprocessor or SMP configurations but does not outperform the Hyper-Threaded processor being used in a multiprogrammed (or “virtual multiprocessor”) configuration.

5.6 Summary

I have described various ways to use SMT processors other than as a virtual multiprocessor. Treating an SMT processor as a single resource avoids the problem of non-cooperating threads detrimenting each others’ performance. This approach allows the programmer and/or compiler to choose the threads that will run best together. The difficulty is that a multithreaded application optimised for a particular SMT processor may not be optimal for another SMT processor of the same architecture but with a different number of threads or a different microarchitecture. Techniques such as precomputation and multi-path execution utilise the extra threads to speed up a single thread. Subordinate threads are a promising technique but are limited by the currently available hardware, particularly the lack of thread priority and inter-thread synchronisation.

I have described preliminary experiments to assess the value of partitioning application and interrupt handler processing across different logical processor (threads) of an SMT processor. The results suggest that the technique is useful for some workloads and warrants further investigation.

Chapter 6

Conclusions

In this dissertation I have described experiments performed to analyse the behaviour of a simultaneous multithreaded processor, the Intel Pentium 4 with Hyper-Threading. The dynamic sharing of processor resources causes a wide variety of interactions between simultaneously executing processes often resulting in poor per-thread, or system-wide performance. I presented a series of process scheduling algorithms that are sensitive to the characteristics of the Hyper-Threaded processor and showed that they can improve the system throughput and/or fairness compared to using a traditional scheduler. In this chapter I summarise the work and my contributions and describe some directions for further research.

6.1 Summary

In Chapter 1 I described the motivation for studying the performance of SMT processors, particularly the effect of the mutual interaction of simultaneously executing threads. The current use of SMT processors, “virtual multiprocessors”, provides a simple interface for operating systems but hides the performance effect of threads sharing the processor. I presented my thesis, that SMT processors can be used more effectively with an operating system that is aware of their characteristics.

In Chapter 2 I described simultaneous multithreading and the commercial implementations of it. I described areas of operating system support that are necessary or desirable for SMT processors. I highlighted the process scheduler as an area that currently lacks detailed support.

In Chapter 3 I presented a series of experiments measuring the performance of Hyper-Threading. I showed how pairs of processes simultaneously executing can exhibit a wide range of performance. The experiments used processor hardware performance counters to investigate and explain the observed behaviour and develop a model to estimate the current performance based on those counters. I demonstrated how the performance of processes can change dramatically during their lifetime as they move through different phases of execution. I ended the chapter with a study that showed how different allocations of processes to logical processors in a two-package system could result in quite different system throughputs.

In Chapter 4 I presented a number of process scheduling algorithms that are able to take SMT thread performance interactions into account in order to improve the throughput and/or fairness of the system. I demonstrated how these techniques could be incorporated into a traditional scheduler so as to maintain the existing facilities of priority and starvation avoidance. The schedulers yielded improvements of up to 3.2% over a traditional scheduler; this figure is comparable with related work.

Finally, in Chapter 5 I discussed ways to exploit SMT processors other than the "virtual multiprocessor" model studied in the earlier chapters of this dissertation. Many of the alternative uses for simultaneous multithreading try to use the extra thread(s) to improve the throughput of a single thread. I presented preliminary experiments to assess the value of using two threads to process application and interrupt handling code separately. These initial results are encouraging and suggest the technique warrants further research.

In conclusion, my thesis — that SMT processors can be used more effectively with an operating system that is aware of their characteristics — is justified as follows. I showed in Chapter 3 that there is a large variability in thread and system performance when different workloads are executed simultaneously on a real SMT processor. I showed how rearranging the assignment of processes to logical processors could improve performance by up to 30%. An operating system without any knowledge of SMT processors would not be able to perform such a rearrangement and would not be able to tell that a given arrangement was yielding poor performance. Secondly I presented SMT-aware process scheduling algorithms in Chapter 4 that were able to improve the throughput and/or fairness, compared to a scheduler unaware of the characteristics of an SMT processor, of a set of processes.

6.2 Further Research

The work presented in this dissertation could be taken further in a number of ways.

The performance estimation model developed in Chapter 3 was useful and accurate enough to provide scheduler heuristics. However, there is scope for further refinement by considering more performance counter metrics. A similar study on other SMT processors would allow models to be compared with the possibility of a parameterisable general model.

The thread interaction study in Chapter 3 showed how some workloads regularly caused the simultaneously executing thread to experience a low performance. There were workloads whose performance was difficult to impact and some that experienced a reduced performance when running with almost any other workload. Analysis of the hardware performance counters allowed many of these effects to be explained. A more formal classification model would be useful, perhaps based on Bayesian Classification. Being able to classify a running process in terms of the harm it can do to other processes and the degree to which it suffers itself would enable a simpler algorithm with the sole aim of avoiding simultaneous execution of processes that have been classified as likely to reduce performance.

The SMT-aware scheduling algorithms presented in Chapter 4 used dynamic measurement of running processes to provide feedback for scheduling. Processes were referenced by their identifiers (PIDs) with subsequent, or forked, instances of the same program having different PIDs and therefore being treated separately from the previous, or parent, process. It is likely that these subsequent instantiations would perform similarly to their predecessors so investigating the incorporation of data from the predecessor would be worthwhile.

In common with many scheduling algorithms, my SMT-aware algorithms have a number of tuneable parameters. Tuning these parameters could yield further speedups. The parameters include:

- the frequency of the performance estimate calculation,
- the scaling of the dynamic priority increases,
- the frequency of the *plan* offline planner's execution, and
- the size of the *tryhard* performance matrix.

The experiments described in Chapter 5 showed how a system configured to partition application and interrupt processing on to different Hyper-Threads could yield speedups over traditional configurations. The performances observed were similar to those obtained when using Hyper-Threading as a "virtual multiprocessor" but without the hard-to-predict interaction between application threads. A scheme such as this would be useful in situations were predictability is important such system providing quality-of-service or resource accounting.

This topic could be taken further with a study of more workloads, such as file or database servers. Additionally the *Piglet* concept of asynchronous system calls could be implemented using SMT. This would have the advantage of avoiding expensive privilege level changes but without the potentially costly problem of inter-cache traffic of the original Piglet system. The current generation of Intel Hyper-Threaded processors provide a lightweight inter-thread synchronisation mechanism (the `MONITOR` and `MWAIT`) instructions which could be used for this. However, current implementations only allow use of this facility in kernel-mode; this permits a user-mode program to asynchronously signal the kernel but the user-mode program is not able to use the same lightweight mechanism to wait on a reply. An evaluation of the technique could be performed by executing programs in kernel mode. It is certain that future processors will extend this synchronisation mechanism to user-mode.

SMT processors are likely to be commonplace for a good while. SMT provides an effective way to extract more throughput from a processor without incurring a high implementation overhead. With Intel now producing SMT processors as standard and IBM starting to produce their SMT Power5 processor, the use of SMT will become more widespread. It is likely that future SMT processor will support more threads than current implementations because the overhead of adding the extra state to the processor is fairly low. However, although early SMT research described processors with up to 8 threads and hinted at even larger numbers, practical considerations such as the size and complexity of the register file limit the scalability of SMT. Multicore processors, which have now been available for a few years, with a shared level 2 cache are likely to have their scalability limited by the connection and bandwidth between the cores and the cache. Both IBM and Sun Microsystems have announced processors which combine multithreading and multiple cores. It is likely that this combination will become more common in the future.

One problem with extra threads is the increased demand for bottleneck resources such as the caches. This problem is compounded on multicore systems where the cores share a level 2 cache. A suitable schedule of software threads to hardware threads is likely to become more important.

Further study of scheduling and processor allocation would be useful. Additionally, the use of hardware threads for system purposes, such as optimisation and performance monitoring, would be useful as the threads can be designed to minimise their load on bottleneck resources.

A focus of this dissertation has been the evaluation of performance and investigation of techniques on real commodity systems. At the time of writing, multicore processors (chip multiprocessors – CMP) and multicore-multithreaded processors (often referred to as chip multithreading – CMT) are only beginning to appear in commodity systems. Many of the areas covered in this dissertation are relevant to multicore processors. CMP systems often have a shared level 2 cache and per-core level 1 caches. The sharing of the level 2 cache was found to be a major contributor to the performance interactions between Hyper-Threads, it will be the main cause of performance interactions for CMP systems. CMT systems introduce a three level hierarchy of processors: multiple packages, each of multiple cores, each of multiple logical processors (threads). The varying degrees of sharing at the different levels presents an interesting challenge to efficiently utilise the processors without losing performance through resource contention. The techniques of feedback-directed scheduling explored in this dissertation could be extended to CMT systems.

Bibliography

[Akkary98] H. Akkary and M. A. Driscoll. *A Dynamic Multithreaded Processor*. In Proceedings of the 31st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-31), pages 226–236. IEEE Computer Society, December 1998. (p 97)

[Alverson90] R. Alverson, D Callahan, D. Cummings, D. Koblenz A. Porterfield, and B. J. Smith. *The Tera Computer System*. In Proceedings of the 4th International Conference on Supercomputing, pages 1–6. ACM Press, June 1990. (p 20)

[Anderson67] D. W. Anderson, F. J. Sparacio, and F. M. Tomasulo. *The IBM System/36O Model 91: Machine Philosophy and Instruction-Handling*. IBM Journal, 11:8–24, January 1967. (p 100)

[Barham03] P. R. Barham, B. Dragovic, K. A. Fraser S. M. Hand, T. L. Harris, A. C. Ho R. Neugebauer, I. A. Pratt, and A. K. Warfield. *Xen and the Art of Virtualization*. In Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03), pages 164–177. ACM Press, October 2003. (p 106)

[Boggs04] D. Boggs, A. Baktha, J. Hawkins, D. T. Marr J. A. Miller, P. Roussel, R. Singhal, B. Toll and K. S. Venkatraman. *The Microarchitecture of the Intel Pentium 4 Processor on 90nm Technology*. Intel Technology Journal, 8(1):1–17, February 2004. (pp 24, 32)

[Borozan02] H. Borozan. *Microsoft Windows-Based Servers and Intel Hyper-Threading Technology*. Technical Article, Microsoft Corporation, April 2002. (p 27)

[Bovet00] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly and Associates, October 2000. (p 77)

[Bulpin04] J. R. Bulpin and I. A. Pratt. *Multiprogramming Performance of the Pentium 4 with Hyper-Threading*. In Third Annual Workshop on Duplicating, Deconstruction and Debunking (at ISCA'04), pages 53–62, June 2004. (p 45)

[Cazorla03] F. J. Cazorla, E. Fernandez, A. Ramirez, and M. Valero. *Improving Memory Latency Aware Fetch Policies for SMT Processors*. In Proceedings of the 5th International Symposium on High Performance Computing (ISHPC), pages 70–85. Springer-Verlag, October 2003. (p 51)

[Cazorla04] F. J. Cazorla, P. M. W. Knijnenburg, R. Sakellariou E. Fernandez, A. Ramirez, and M. Valero. *Predictable Performance in SMT Processors.* In Proceedings of the first ACM International Conference on Computing Frontiers (CF04), pages 433–443. ACM Press, April 2004. (p 73)

[Chappell99] R. S. Chappell, J. Stark, S. P. Kim S. K. Reinhardt, and Y. N. Patt. *Simultaneous subordinate microthreading (SSMT).* In Proceedings of the 26th International Symposium on Computer Architecture (ISCA '99), pages 186–195. IEEE Computer Society, May 1999. (p 102)

[Chen02] Y-K. Chen, M. Holliman, E. Debes, S. Zheltov A. Knyazev, S. Bratanov, R. Belenov, and I. Santos. *Media Applications on Hyper-Threading Technology.* Intel Technology Journal, 6(2):47–57, February 2002. (pp 35, 39)

[Collins01a] J. D. Collins, D. M. Tullsen, H. Wang, and J. P. Shen. *Dynamic Speculative Precomputation.* In Proceedings of the 34th Annual International Symposium on Microarchitecture (MICRO-34), pages 306–317. IEEE Computer Society, December 2001. (p 98)

[Collins01b] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes Y-F. Lee, D. Lavery, and J. P. Shen. *Speculative Precomputation: Long-range Prefetching of Delinquent Loads.* In Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA '01), pages 14–25. IEEE Computer Society, July 2001. (p 98)

[Diefendorff99] K. Diefendorff. *Compaq Chooses SMT for Alpha.* Microprocessor Report, 13(16), December 1999. (p 21)

[Dorai02] G. K. Dorai and D. Yeung. *Transparent Threads: Resource Sharing in SMT Processsors for High Single-Threaded Performance.* In Proceedings of the 11th International Conference on Parallel Architectures and Compilation Techniques (PACT '2002), pages 30–41. IEEE Computer Society, September 2002. (p 103)

[Dorozhevets92] M. Dorozhevets and P. Wolcott. *The El'brus-3 and MARS-M: recent advances in Russian high-performance computing.* The Journal of Supercomputing, 6(1):5–48, March 1992. (p 20)

[Dubois98] M. Dubois and Y. H. Song. *Assisted Execution.* Technical Report 98-25, University of Southern California, October 1998. (p 101)

[Eggers97] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo R. L. Stamm, and D. M. Tullsen. *Simultaneous Multithreading: A Platform for Next-Generation Processors.* IEEE Micro, 17(5):12–19, October 1997. (pp 20, 37)

[Fedorova04a] A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum. *Throughput-*

Oriented Scheduling on Chip Multithreading Systems. Technical Report TR-17-04, Harvard University, August 2004. (p 75)

[Fedorova04b] A. Fedorova, C. Small, D. Nussbaum, and M. Seltzer. *Chip Multithreading Systems Need a New Operating System Scheduler.* In 11th ACM SIGOPS European Workshop, September 2004. (p 75)

[Fraser04] K. A. Fraser, S. M. Hand, R. Neugebauer, I. A. Pratt A. K. Warfield, and M. A. Williamson. *Reconstructing I/O.* Technical Report UCAM-CL-TR-596, University of Cambridge Computer Laboratory, August 2004. (p 106)

[Glaskowsky04] P. N. Glaskowsky. *Prescott Pushes Pipelining Limits.* Microprocessor Report, February 2004. (p 23)

[Greenwald96] M. Greenwald and D. Cheriton. *The Synergy Between Non-blocking Synchronization and Operating System Structure.* In Proceedings of the USENIX 2nd Symposium on Operating Systems Design and Implementation (OSDI '96), pages 123–136. The USENIX Association, October 1996. (p 28)

[Grunwald02] D. Grunwald and S. Ghiasi. *Microarchitectural Denial of Service: Insuring Microarchitectural Fairness.* In Proceedings of the 35th Annual International Symposium on Microarchitecture (MICRO-35), pages 409–418. IEEE Computer Society, November 2002. (p 38)

[Hennessy03] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan-Kaufman, third edition, 2003. (p 18)

[Henning00] J. L. Henning. *SPEC CPU2000: Measuring CPU Performance in the New Millenium.* IEEE Computer, 33(7):28–35, July 2000. (p 83)

[Hinton01] G. Hinton, D. Sager, M. Upton, D. Boggs D. Carmean, A. Kyker, and P. Roussel. *The Microarchitecture of the Pentium 4 Processor.* Intel Technology Journal, 5(1):1–13, February 2001. (p 21)

[Hirata92] H. Hirata, K. Kimura, S. Nagamine, Y. Mochizuki A. Nishimura, Y. Nakase, and T. Nishizawa. *An Elementary Processor Architecture with Simultaneous Instruction Issuing from Multiple Threads.* In Proceedings of the 19th International Symposium on Computer Architecture (ISCA '92), pages 136–145. IEEE Computer Society, May 1992. (p 20)

[IEEE04] IEEE. *IEEE 1003.1-2004: Standard for Information Technology — Portable Operating System Interfaces (POSIX).* IEEE Computer Society, 2004. (p 96)

[Intel01a] Intel Corporation. *Intel Architecture Software Developer's Manual. Volume 2: Instruction Set Reference*, 2001. (p 29)

[Intel01b] Intel Corporation. *Intel Architecture Software Developer's Manual. Volume 3: System Programming Guide*, 2001. (pp 27, 40)

[Intel01c] Intel Corporation. *Introduction to Hyper-Threading Technology*, 2001. (p 21)

[Intel03] Intel Corporation. *Prescott New Instructions Software Developer's Guide*, June 2003. (pp 24, 29)

[Kalla04] R. Kalla, B. Sinharoy, and J. M. Tendler. *IBM Power5 Chip: a Dual-Core Multithreaded Processor*. IEEE Micro, 24(2):40–47, March 2004. (pp 24, 29, 31)

[Kaxiras01] S. Kaxiras, G. Narlikar, A. D. Berenbaum, and Z. Hu. *Comparing Power Consumption of an SMT and a CMP DSP for Mobile Phone Workloads*. In Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES '01), pages 211–220. ACM Press, November 2001. (p 35)

[Kessler92] R. E. Kessler and M. D. Hill. *Page Placement Algorithms for Large Real-Indexed Caches*. ACM Transactions on Computer Systems, 10(4):338–359, November 1992. (p 33)

[Kim04] D. Kim, S. S. Liao, P. H. Wang, J. del Cuvillo X. Tian, X. Zou, D. Yeung, M. Girkar and J. P. Shen. *Physical Experimentation with Prefetching Helper Threads on Intel's Hyper-Threaded Processors*. In Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO '04), pages 27–38. IEEE Computer Society, March 2004. (p 99)

[Koufaty03] D. Koufaty and D. T. Marr. *Hyperthreading Technology in the Netburst Microarchitecture*. IEEE Micro, 23(2):56–64, 2003. (p 21)

[Krewell02] K. Krewell. *Intel's Hyper-Threading Takes Off*. Microprocessor Report, December 2002. (p 21)

[Kumar04] R. Kumar, D. M. Tullsen, P. Ranganathan N. P. Jouppi, and K. I. Farkas. *Single-ISA Heterogeneous Multi-Core Architectures for Multi-Threaded Workload Performance*. In Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA '04), pages 64–75. IEEE Computer Society, June 2004. (p 74)

[Lee98] D. C. Lee, P. J. Crowley, J-L. Baer, T. E. Anderson and B. N. Bershad. *Execution Characteristics of Desktop Applications on Windows NT*. In Proceedings of the 25th International Symposium on Computer Architecture (ISCA '98), pages 27–38. IEEE Computer Society, June 1998. (pp 20, 71)

[Limousin01] C. Limousin, J. Sebot, A. Vartanian, and N. Drach-Temam. *Improving 3D Geometry Transformations on a Simultaneous Multithreaded SIMD Processor*. In Proceedings of the 15th International Conference on Supercomputing, pages 236–245, May 2001. (p 26)

[Lo97a] J. L. Lo, S. J. Eggers, H. M. Levy, S. S. Parekh and D. M. Tullsen. *Tuning*

Compiler Optimizations for Simultaneous Multithreading. In Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-30), pages 114–124. IEEE Computer Society, December 1997. (p 33)

[Lo97b] J. L. Lo, J. S. Emer, H. M. Levy, R. L. Stamm D. M. Tullsen, and S. J. Eggers. *Converting Thread-Level Parallelism to Instruction-Level Parallelism via Simultaneous Multithreading.* ACM Transactions on Computer Systems, 15(3):322–354, August 1997. (p 20)

[Lo98] J. L. Lo, L. A. Barroso, S. J. Eggers K. Gharachorloo, H. M. Levy, and S. S. Parekh. *An Analysis of Database Workload Performance on Simultaneous Multithreaded Processors.* In Proceedings of the 25th International Symposium on Computer Architecture (ISCA '98), pages 39–50. IEEE Computer Society, June 1998. (p 32)

[Loikkanen96] M. Loikkanen and N. Bagherzadeh. *A Fine-Grain Multithreading Superscalar Architecture.* In Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques (PACT '96), pages 163–168. IEEE Computer Society, October 1996. (p 21)

[Luk01] C-K. Luk. *Tolerating Memory Latency through Software-Controller Pre-Execution in Simultaneous Multithreading Processors.* In Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA '01), pages 40–51. IEEE Computer Society, July 2001. (p 99)

[Magro02] W. Magro, P. Peterson, and S. Shah. *Hyper-Threading Technology: Impact on Compute-Intensive Workloads.* Intel Technology Journal, 6(2):58–66, February 2002. (p 38)

[Marcuello98] P. Marcuello, A. González, and J. Tubella. *Speculative Multithreaded Processors.* In Proceedings of the 12th International Conference on Supercomputing, pages 77–84. ACM Press, July 1998. (p 97)

[Marcuello99] P. Marcuello and A. González. *Exploiting Speculative Thread-Level Parallelism on a SMT Processor.* In Proceedings of the 7th International Conference on High Performance Computing and Networking Europe 1999, pages 141–150. Springer-Verlag, April 1999. (p 97)

[Marr02] D. T. Marr, F. Binns, D. L. Hill, G. Hinton D. A. Koufaty, J. A. Miller, and M. Upton. *Hyper-Threading Technology Architecture and Microarchitecture.* Intel Technology Journal, 6(2):1–12, February 2002. (pp 21, 29, 46)

[McDowell03] L. K. McDowell, S. J. Eggers, and S. D. Gribble. *Improving Server Software Support for Simultaneous Multithreaded Processors.* In Proceedings of the ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '03), pages 37–48. ACM Press, June 2003. (p 33)

[Moore96] S. W. Moore. *Multithreaded Processor Design*. Kluwer Academic Publishers, 1996. (pp 19, 106)

[Mosberger98] David Mosberger and Tai Jin. *httperf: A Tool for Measuring Web Server Performance*. In First Workshop on Internet Server Performance, pages 59–67, June 1998. (p 112)

[Muir00] S. J. Muir and J. M. Smith. *Piglet: A Low-Intrusion Vertical Operating System*. Technical Report MS-CIS-00-04, University of Pennsylvania, January 2000. (p 106)

[Nakajima02] J. Nakajima and V. Pallipadi. *Enhancements for Hyper-Threading Technology in the Operating System — Seeking the Optimal Scheduling*. In Proceedings of the 2nd Workshop on Industrial Experiences with Systems Software. The USENIX Association, December 2002. (p 75)

[Oplinger02] J. Oplinger and M. S. Lam. *Enhancing Software Reliability with Speculative Threads*. In Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '02), pages 184–196. ACM Press, October 2002. (p 103)

[Parekh00] S. S. Parekh, S. J. Eggers, H. M. Levy, and J. L. Lo. *Thread-Sensitive Scheduling for SMT Processors*. Technical Report 2000-04-02, University of Washington, June 2000. (pp 74, 82)

[Park03] I. Park, B. Falsafi, and T. N. Vijaykumar. *Implicitly-Multithreaded Processors*. In Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA '03), pages 39–50. IEEE Computer Society, June 2003. (p 97)

[Prabhu03] M. K. Prabhu and K. Olukotun. *Using Thread-Level Speculation to Simplify Manual Parallelization*. In Proceedings of the ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '03), pages 1–12. ACM Press, June 2003. (p 97)

[Preston02] R. P. Preston, R. W. Badeau, D. W. Bailey, S. L. Bell L. L. Biro, W. J. Bowhill, D. E. Dever, S. Felix R. Gammack, V. Germini, M. K. Gowan, P. Gronowski D. B. Jackson, S. Mehta, S. V. Morton J. D. Pickholt, M. H. Reilly, and M. J. Smith˙ *Design of an 8-wide Superscalar RISC Microprocessor with Simultaneous Multithreading*. In Proceedings of the 2002 IEEE Solid-State Circuits Conference (ISSCC 2002), pages 334–335. IEEE Solid-State Circuits Society, February 2002. (p 21)

[Puppin01] D. Puppin and D. M. Tullsen. *Maximizing TLP with loop-parallelization on SMT*. In Workshop on Multi-Threaded Execution, Architectures and Compilers (MTEAC-5), December 2001. (p 96)

[Purser00] Z. Purser, K. Sundaramoorthy, and E. Rotenberg. *A Study of Slipstream Processors*. In Proceedings of the 33rd Annual International Symposium on

Microarchitecture (MICRO-33), pages 269–280. IEEE Computer Society, December 2000. (p 99)

[Redstone00] J. A. Redstone, S. J. Eggers, and H. M. Levy. *An Analysis of Operating System Behaviour on a Simultaneous Multithreaded Architecture.* In Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '00), pages 245–256. ACM Press, November 2000. (p 38)

[Redstone02] J. A. Redstone. *An Analysis of Software Interface Issues for SMT Processors.* PhD thesis, University of Washington, December 2002. (pp 28, 30, 101)

[Redstone03] J. A. Redstone, S. J. Eggers, and H. M. Levy. *Mini-Threads: Increasing TLP on Small-Scale SMT Processors.* In Proceedings of the nineth International Symposium on High Performance Computer Architecture (HPCA-9), pages 19–30. IEEE Computer Society, February 2003. (p 101)

[Reinhardt00] S. K. Reinhardt and S. S. Mukherjee. *Transient Fault Detection via Simultaneous Multithreading.* In Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA '00), pages 25–36. IEEE Computer Society, June 2000. (p 104)

[Rotenberg99] E. Rotenberg. *AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors.* In Proceedings of the 29th International Symposium on Fault-Tolerant Computing (1999), pages 84–91. IEEE Computer Society, June 1999. (p 104)

[Roth01] A. Roth and G. S. Sohi. *Speculative Data-Driven Multithreading.* In Proceedings of the seventh International Symposium on High Performance Computer Architecture (HPCA-7), pages 37–48. IEEE Computer Society, January 2001. (p 99)

[Roth99] A. Roth, A. Moshovos, and G. S. Sohi. *Improving Virtual Function Call Target Prediction via Dependence-Based Pre-Computation.* In Proceedings of the 13th International Conference on Supercomputing, pages 356–364. ACM Press, May 1999. (p 99)

[Sazeides01] Y. Sazeides and T. Juan. *How to Compare the Performance of Two SMT Microarchitectures.* In Proceedings of the 2001 IEEE International Symposium on Performance Analysis of Systems and Software, pages 180–183, November 2001. (p 44)

[Seng00] J. S. Seng, D. M. Tullsen, and G. Z. N. Cai. *Power-Sensitive Multithreaded Architecture.* In Proceedings of the 2000 IEEE International Conference on Computer Design, pages 199–206. IEEE Computer Society, September 2000. (p 35)

[Serrano94] M. J. Serrano, W. Yamamoto, R. C. Wood and M. D. Nemirovsky. *A Model for Performance Estimation in a Multistreamed Superscalar Pro-*

cessor. In Proceedings of the 7th International Conference on Computer Performance Evaluation, Modelling Techniques and Tools, volume 794 of *Lecture Notes in Computer Science*, pages 213–230. Springer-Verlag, May 1994. (p 20)

[Sherwood99] T. Sherwood and B. Calder. *Time Varying Behaviour of Programs*. Technical Report CS99-630, University of California, San Diego, August 1999. (p 59)

[Snavely00] A. Snavely and D. M. Tullsen. *Symbiotic Jobscheduling for a Simultaneous Multithreading Processor*. In Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '00), pages 234–244. ACM Press, November 2000. (pp 37, 74)

[Snavely02] A. Snavely, D. M. Tullsen, and G. Voelker. *Symbiotic Jobscheduling with Priorities for a Simultaneous Multithreading Processor*. In Proceedings of the 2002 International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '02), pages 66–76. ACM Press, June 2002. (pp 37, 74)

[Snavely99] A. Snavely, N. Mitchell, L. Carter, J. Ferrante and D. M. Tullsen. *Explorations in Symbiosis on two Multithreaded Architectures*. In Workshop on Multi-Threaded Execution, Architectures and Compilers (MTEAC '99), January 1999. (pp 37, 44)

[Sohi95] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. *Multiscalar Processors*. In Proceedings of the 22th International Symposium on Computer Architecture (ISCA '95), pages 414–425. IEEE Computer Society, June 1995. (p 97)

[Solomon00] D. A. Solomon and M. E. Russinovich. *Inside Windows 2000*. Microsoft Press, third edition, June 2000. (p 93)

[SPEC] *The Standard Performance Evaluation Corporation, http://www.spec.org/*. (pp 41, 45, 107, 112)

[Squillante93] M. S. Squillante and E. D. Lazowska. *Using Processor-Cache Affinity Information in Shared-Memory Multiprocessor Scheduling*. IEEE Transactions on Parallel and Distributed Systems, 4(2):131–143, February 1993. (p 72)

[Steffan00] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. *A Scalable Approach to Thread-Level Speculation*. In Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA '00), pages 1–12. IEEE Computer Society, June 2000. (p 97)

[Steffan98] J. G. Steffan and T. C. Mowry. *The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization*. In Proceedings of the fourth International Symposium on High Performance Computer Architecture (HPCA-4), pages 2–13. IEEE Computer Society, February 1998. (p 97)

[Sundaramoorthy00] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. *Slipstream Processors: Improving both Performance and Fault Tolerance*. In Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '00), pages 257–268. ACM Press, November 2000. (p 99)

[Tomasulo67] R. M. Tomasulo. *An Efficient Algorithm for Exploiting Multiple Arithmetic Units*. IBM Journal of Research and Development, 11(1):25–33, January 1967. (p 18)

[Tuck03] N. Tuck and D. M. Tullsen. *Initial Observations of the Simultaneous Multithreading Pentium 4 Processor*. In Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT '2003), pages 26–34. IEEE Computer Society, September 2003. (pp 39, 41, 46)

[Tullsen01] D. M. Tullsen and J. A. Brown. *Handling Long-latency Loads in a Simultaneous Multithreading Processor*. In Proceedings of the 34th Annual International Symposium on Microarchitecture (MICRO-34), pages 318–327. IEEE Computer Society, December 2001. (p 51)

[Tullsen95] D. M. Tullsen, S. J. Eggers, and H. M. Levy. *Simultaneous Multithreading: Maximizing On-Chip Parallelism*. In Proceedings of the 22th International Symposium on Computer Architecture (ISCA '95), pages 392–403. IEEE Computer Society, June 1995. (pp 20, 37)

[Tullsen96a] D. M. Tullsen. *Simulation and Modeling of a Simultaneous Multithreading Processor*. In 22nd Annual Computer Measurement Group Conference, pages 819–828. Computer Measurement Group, December 1996. (p 37)

[Tullsen96b] D. M. Tullsen, S. J. Eggers, J. S. Emer, and H. M. Levy. *Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor.* In Proceedings of the 23th International Symposium on Computer Architecture (ISCA '96), pages 191–202. IEEE Computer Society, May 1996. (pp 20, 37)

[Tullsen98] D. M. Tullsen, S. J. Eggers, and H. M. Levy. *Retrospective: Simultaneous Multithreading: Maximizing On-Chip Parallelism*. In 25 Years of the International Symposia on Computer Architecture (Selected Papers), pages 115–116. ACM Press, August 1998. (p 20)

[Tullsen99] D. M. Tullsen, J. L. Lo, S. J. Eggers, and H. M. Levy. *Supporting Fine-Grained Synchronization on a Simultaneous Multithreading Processor*. In Proceedings of the fifth International Symposium on High Performance Computer Architecture (HPCA-5), pages 54–58. IEEE Computer Society, January 1999. (p 29)

[Ungerer03] T. Ungerer, B. Robič, and J. Šilc. *A Survey of Processors with Explicit Multithreading*. ACM Computing Surveys, 35(1):29–63, March 2003. (p 20)

[Vianney03] D. Vianney. *Hyper-Threading speeds Linux*. IBM developerWorks, January 2003. (p 38)

[Vijaykumar02] T. N. Vijaykumar, I. Pomeranz, and K. Cheng. *Transient-Fault Recovery Using Simultaneous Multithreading*. In Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA '02), pages 87–98. IEEE Computer Society, May 2002. (p 104)

[von Behren03] R. von Behren, J. Condit, F. Zhou, G. C. Necula and E. Brewer. *Capriccio: Scalable Threads for Internet Services*. In Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03), pages 268–281. ACM Press, October 2003. (p 96)

[Wallace98] S. Wallace, B. Calder, and D. M. Tullsen. *Threaded Multiple Path Execution*. In Proceedings of the 25th International Symposium on Computer Architecture (ISCA '98), pages 238–249. IEEE Computer Society, June 1998. (p 100)

[Wang02] H. Wang, P. H. Wang, R. D. Weldon, S. M. Ettinger H. Saito, M. Girkar, S. S. Liao, and J. P. Shen. *Speculative Precomputation: Exploring the Use of Multithreading for Latency*. Intel Technology Journal, 6(2):22–35, February 2002. (p 98)

[Yamamoto95] W. Yamamoto and M. D. Nemirovsky. *Increasing Superscalar Performance Through Multistreaming*. In Proceedings of the 1995 Conference on Parallel Architectures and Compilation Techniques (PACT '95), pages 49–58. IEEE Computer Society, June 1995. (p 20)

[Zhou04] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas. *iWatcher: Efficient Architectural Support for Software Debugging*. In Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA '04), pages 224–235. IEEE Computer Society, June 2004. (p 98)

[Zilles01] C. B. Zilles and G. S. Sohi. *Execution-based Prediction using Speculative Slices*. In Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA '01), pages 2–13. IEEE Computer Society, July 2001. (p 99)

[Zillies99] C. B. Zillies, J. S. Emer, and G. S. Sohi. *The Use of Multithreading for Exception Handling*. In Proceedings of the 32nd Annual International Symposium on Microarchitecture (MICRO-32), pages 219–229. IEEE Computer Society, November 1999. (p 105)

Appendix A

Monochrome Figures

This appendix contains monochrome versions of colour figures which when rendered in black and white are not easily understandable.

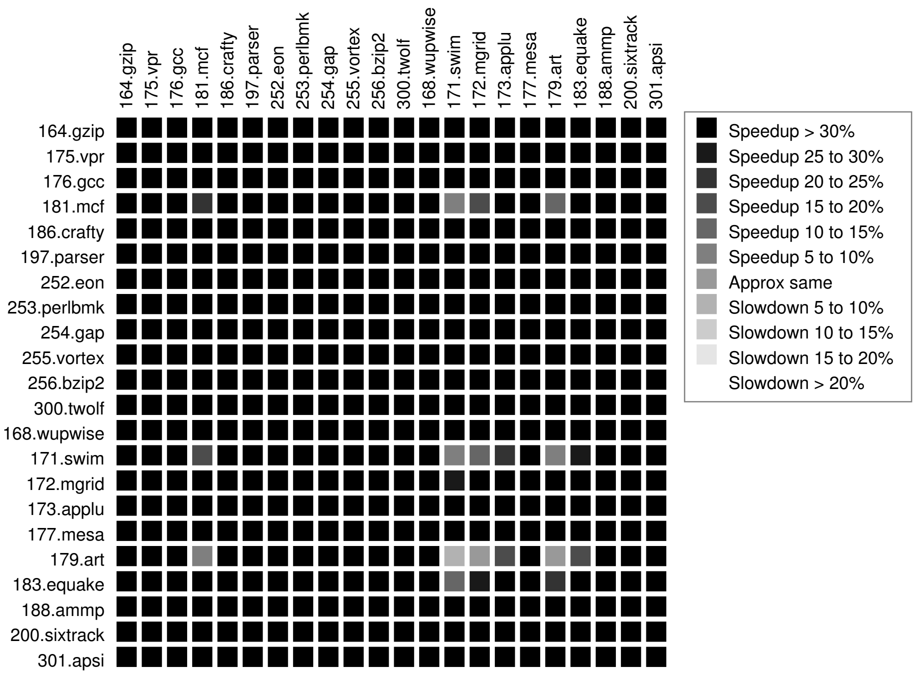
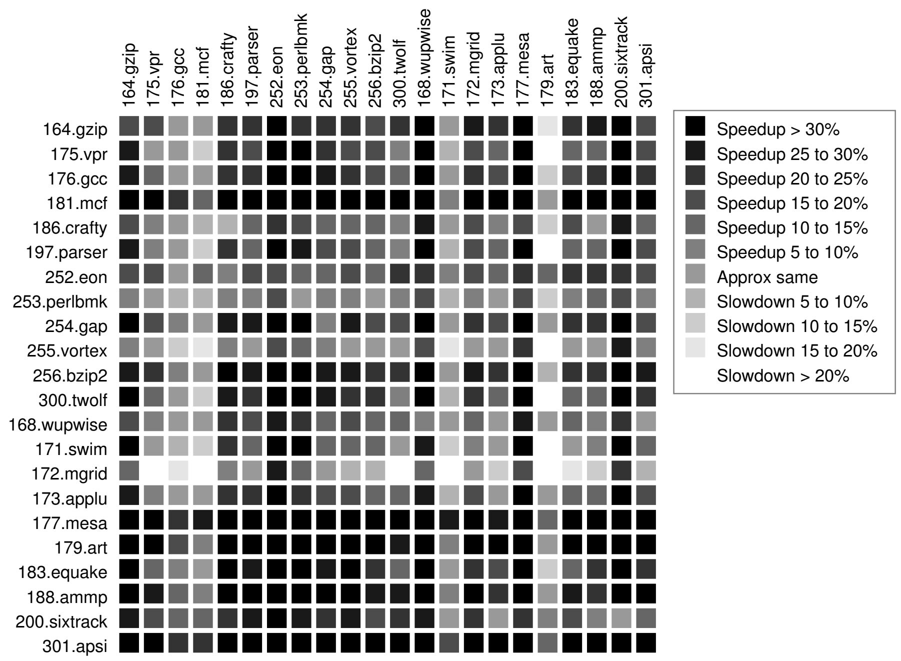
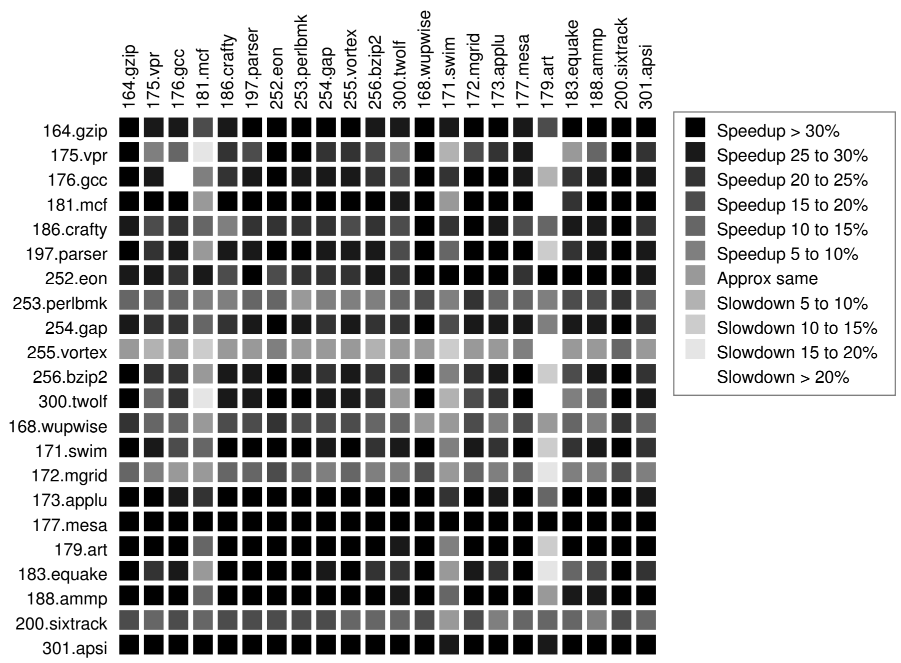


Figure A.1: *Monochrome version of Figure 3.7.* Effect on each SPEC CPU2000 benchmark in a multiprogrammed pair running on an SMP configuration. A black square represents a good performance ratio for the subject benchmark and a white square denotes a bad performance ratio (relative to "perfect" SMP).



(a) Northwood



(b) Prescott

Figure A.2: *Monochrome version of Figure 3.3*. Effect on each SPEC CPU2000 benchmark in a multi-programmed pair running on a Hyper-Threaded processor. A black square represents a good performance ratio for the subject benchmark and a white square denotes a bad performance ratio.