DLL-Conscious Instruction Fetch Optimization for SMT Processors

A Thesis
Presented to
The Academic Faculty

by

Fayez Mohamood

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in Electrical and Computer Engineering

School of Electrical and Computer Engineering
Georgia Institute of Technology
May 2006

DLL-Conscious Instruction Fetch Optimization for SMT Processors

Approved by:

Dr. Hsien-Hsin S. Lee, Advisor
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Sung Kyu Lim
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Sudhakar Yalamanchili
School of Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved: April 7th, 2006

To my Parents...

ACKNOWLEDGEMENTS

There are many who have given me inspiration, guidance and provided me with professional and personal support. First of all, I would like to extend my heartfelt thanks to my parents for the unwavering support they provided me in every way possible, in order to enable me reach both educational and personal goals. I would also like to thank my brothers, for always being there whenever I needed their support. My family has been a critical part of my educational endeavors and their efforts will always be appreciated and remembered.

I would like to extend my appreciation to my advisor Dr. Hsien-Hsin Sean Lee for his strong support throughout the time I have worked with him and for helping me understand that there is always room for realizing higher potential in myself. My sincere thanks also goes to Dr. Sung Kyu Lim and Dr. Sudhakar Yalamanchili for serving on my thesis committee. In addition, I would also like to thank Mrinmoy Ghosh for providing me with valuable advice and suggestions to improve my work.

I would also like to thank everyone in the MARS (Microprocessor Architecture Research Society) group who have provided a stimulating environment to learn many things and foster new ideas. Last but not the least, I would like to extend my gratitude to many friends in my classes and outside, who have all influenced me in a constructive manner.

TABLE OF CONTENTS

DEDICATION . . . . **iii**

ACKNOWLEDGEMENTS . . . . **iv**

LIST OF TABLES . . . . **vii**

LIST OF FIGURES . . . . **viii**

SUMMARY . . . . **ix**

I INTRODUCTION **1**

II SHARED LIBRARIES IN MODERN OPERATING SYSTEMS . . . **4**

2.1 Virtual Memory Management in Modern Operating Systems 4

2.2 DLLs in Windows NT 6

2.3 Revisiting Simultaneous Multithreading 7

III DLL THRASHING AND DUPLICATION **9**

IV DLL CONSCIOUS INSTRUCTION FETCH **12**

4.1 VIVT Cache Optimization 13

4.2 VIPT Cache Optimization 14

4.3 Design Implications 15

V SIMULATION FRAMEWORK **18**

5.1 Bochs System Emulator 18

5.2 x86 Performance Simulator 21

VI EXPERIMENTAL RESULTS **24**

6.1 Analysis for a 2-Way SMT machine 28

6.2 Analysis for a 4-Way SMT machine 38

6.3 IPC Improvements 40

6.3.1 Observations on a 4-Wide Machine 40

6.3.2 Observations on an 8-Wide Machine 40

6.3.3 Sensitivity of Memory Latencies 42

6.3.4 Sensitivity under Perfect Branch Prediction 44

6.3.5 Sensitivity under Dynamic Resource Partitioning 45

VII RELATED WORK **46**

VIII CONCLUSIONS **48**

REFERENCES **49**

LIST OF TABLES

1 DLL Functionality 6
2 DLL Distribution in Windows Applications 10
3 Microarchitecture Parameters 22

LIST OF FIGURES

1 Virtual Address Space Layout for two distinct processes in Windows NT . . 5

2 A 2-Way SMT Pipeline 7

3 DLL-conscious Instruction Fetch - VIVT Cache 13

4 DLL-conscious Instruction Fetch - VIPT Cache 15

5 Lookup Process of a DLL instruction in a VIPT Cache 16

6 Enhanced TAXI Framework with SMT support 19

7 Distribution of DLL usage 25

8 DLL Misses per Thread for a 2-Way SMT 29

9 I-Cache Hit Rate for a 2-Way SMT 33

10 DLL Misses per Thread for a 4-Way SMT 37

11 I-Cache Hit Rate: 4-Way 39

12 IPC Improvements for 2-Way SMT 41

13 IPC Improvements for 4-Way SMT 43

SUMMARY

Simultaneous multithreading (SMT) processors can issue multiple instructions from distinct processes or threads in the same cycle. This technique effectively increases the overall throughput by keeping the pipeline resources more occupied at the potential expense of reducing single thread performance due to resource sharing. In the software domain, an increasing number of Dynamically Linked Libraries (DLL) are used by applications and operating systems, providing better flexibility and modularity, and enabling code sharing. It is observed that a significant amount of execution time in software today is spent in executing standard DLL instructions, that are shared among multiple threads or processes. However, for an SMT processor with a virtually-indexed based cache implementation, existing instruction fetching mechanisms can induce unnecessary false cache misses caused by the DLL-based instructions, which were intended to be shared. This problem is more conspicuous when multiple independent threads are executing concurrently inside an SMT processor.

This work investigates an often-neglected form of contention between running threads in the I-TLB and I-cache caused by DLLs. To address these shortcomings, we propose a system level technique involving a light-weight modification in the microarchitecture and the OS. By exploiting the nature of the DLLs in our new architecture, we are able to reinstate physical sharing of the DLLs in an SMT machine. Using Microsoft Windows based applications, our simulation results show that the optimized instruction fetching mechanism can reduce the number of DLL misses up to 5.5 times and improve the instruction cache hit rates by up to 62%, resulting in upto 30% DLL IPC improvements and upto 15% overall IPC improvements.

CHAPTER I

INTRODUCTION

Dynamically Linked Libraries (DLL) provide an efficient and systematic way for software development and distribution. Although DLLs have the same internal structure as regular binaries, they are not independent programs, but are modules that provide a common set of services shared by different programs. DLLs can be classified into two categories based on their functionality— *System DLLs and Application DLLs.* System DLLs are provided along with the OS. Windows DLLs like KERNEL32 and NTDLL belong to this category and are used by all applications that run on Microsoft Windows platforms. Application DLLs refer to the DLLs that are distributed together with applications. Application DLLs enable portability, flexibility, and modularity by separating a whole program into easily manageable modules. The Microsoft Windows OS makes extensive use of DLLs for various services. Common Windows-based applications also rely on a major portion of the libraries to perform their functions. System DLLs are loaded into physical memory only once and are mapped to the same location in the virtual address space of each process.

The DLL usage of common desktop applications have very different characteristics from the traditionally used SPEC benchmarks. Studies by Lee et al. in [15] showed that typical desktop applications use an average of 30 DLLs, out of which more than 20 are shared among all applications[1]. It was also observed that instructions from DLL calls made up to 60% of the total application instruction count for certain applications. Clearly, with applications spending most of their time in DLLs, performance implications on the microarchitecture need to be better understood.

A Simultaneous Multi-Threading (SMT) processor [26, 30] is capable of issuing instructions from multiple threads or processes in a given cycle. Such an execution model, especially for a very wide superscalar machine, will lead to a better utilization of the resources that

[1]Based on shared DLLs on applications running on the Windows NT 4.0 OS.

might otherwise go unused as in a single-threaded processor. In an SMT processor, multiple architectural states are maintained for the concurrent execution contexts to make a physical processor appear as several distinct logical processors. From on-line transactional processing, scientific applications, to multi-tasking, multithreaded environments on desktop machines, SMT has been shown effective in achieving a better overall instruction throughput. Recently, commercial high-performance processor vendors began to embrace SMT in their implementations such as Compaq Alpha 21464 [7], Intel Pentium 4 Processor with Hyperthreading Technology [19], IBM Power5 [23], Clearwater's network processor [20], to name a few.

For meeting the decreasing cycle time requirement, many contemporary processors employ virtually-indexed first-level caches to avoid or hide the latency of each address translation incurred by physically-indexed caches. To eliminate address homonym[2] aliasing, an artifact of a virtually-indexed cache, the processor either needs to flush the entire cache contents upon every context switch, or a Process/Thread ID (or sometimes it is referred to as the Address Space ID — ASID[3]) [10, 12] needs to be stored with each memory page and/or cache line in the hardware to discern their respective owner. For an SMT processor with a virtually-indexed cache, a process ID (PID) tag is indispensable to enable multiple accesses from different threads.

In this work, we evaluate the extent of this issue by performing system-level simulations using Microsoft Windows applications and analyzing their instruction fetching behavior on an SMT processor in the presence of DLLs. We found that a considerable amount of performance gain can be achieved by eliminating DLL thrashing. To this end, we propose a cooperative microarchitecture and OS technique to eliminate this effect of DLL conflicts. Our basic idea involves propagating DLL information from the OS to the microarchitecture so that unnecessary conflict misses or duplication in SMT processors can be completely eliminated.

[2]Homonym is caused by the scenario when the same virtual address of different processes maps to different physical addresses.

[3]The terms PID, TID or ASID are used interchangeably to refer to an architecture level identifier for a thread in a multi-threaded or multiprocessing environment.

The rest of this thesis is organized as follows. Chapter 2 describes virtual memory management in modern operating systems and the DLL mapping process that gives rise to the DLL thrashing and duplication effect. Chapter 3 explains how DLL instructions unnecessarily thrash in the I-Cache. We explain our technique to overcome instruction thrashing between threads in Chapter 4. Chapter 5 describes our simulation methodology followed by a quantitative analysis in Chapter 6. Chapter 7 discusses related work and Chapter 8 summarizes our contributions and discusses the implications of our technique in future SMT processor and OS design.

CHAPTER II

SHARED LIBRARIES IN MODERN OPERATING SYSTEMS

2.1 Virtual Memory Management in Modern Operating Systems

Memory management is one of the primary tasks of a modern operating system. Virtual memory requires explicit support from the operating system for a variety of reasons including address space protection and provision for shared memory. In addition to mandatory support, virtual memory managers can also provide fine-grained memory protection, support for sparse address spaces and capability for mapping superpages [11].

Since DLLs are prevalent on the Microsoft Windows platform, we will briefly describe the VMM (Virtual Memory Manager) found in Windows NT. Windows NT supports 32-bit virtual addresses that realizes a 4GB linear address space for each process. However, out of the 4GB address space, only the lower 2GB is available for the application to use[1]. The upper half of the address space is reserved for system use and cannot be accessed by user code. Hence, regardless of the system that the process is running on, the virtual address space is identical, and, similar pages of system memory are mapped into each process at the same relative location for efficiency.

System DLLs are typically loaded when the OS initializes and different processes link to the library functions upon startup. In fact, in the Microsoft Windows implementation there is a Prototype Page Table Entry that points to shared system DLLs for efficiency. For this reason, access to an instruction that belongs to a system DLL will always point to a single address that is valid in the virtual address space of any given running process. All processes eventually address these DLLs through the same Prototype PTE. In addition

[1]Newer generations of the Windows OS support for larger application address spaces

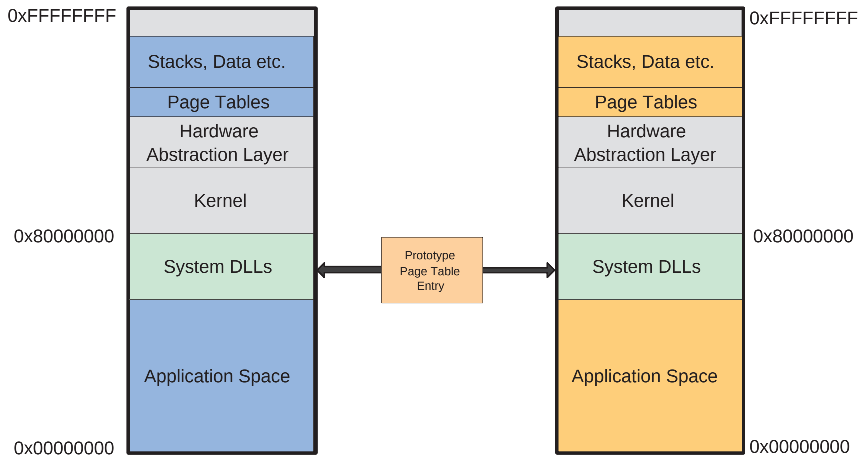


Figure 1: Virtual Address Space Layout for two distinct processes in Windows NT

to this, Windows also loads DLLs on page boundaries, hence the OS can identify DLLs at the granularity of pages since it is guaranteed that DLLs do not straddle page boundaries with regular application code. Figure 1 illustrates the virtual address space of two processes in Windows NT and gives a sense of how system DLLs are mapped at the same relative location in each address space.

The basic concept of shared libraries is similar across different operating systems. In Linux, shared libraries are referred to as Shared Objects (.so). Shared Objects in Linux are also mapped into identical locations in the virtual address space of all processes that use them. However, there is one minor difference between the manner in which shared libraries are loaded in Windows and Linux. As described earlier, Windows has predetermined locations for system DLLs. Hence, no matter what physical system the OS runs on, a given System DLL will be mapped into the same virtual page. In Linux, the first time a shared library is loaded by a process, it will be placed into an available (but not necessarily fixed) virtual address location. However, from this point on, all processes will link to these Shared Objects at the same virtual addresses. In other words, although the virtual address locations of Shared Objects will vary depending on which process loads them first, on a given OS session, they will still be mapped into the same relative location in every process that use them. The fact that shared libraries are mapped into the same relative location in all processes gives rise to DLL thrashing and duplication in the I-Cache. These issues are

Table 1: DLL Functionality

DLL Name	Functionality
KERNEL32	Memory, IO and Interrupt functions
NTDLL	Core operating system functionality
USER32	User interface support like window handling
GDI32	Creation of 2-D graphics
RPCRT4	Remote Procedure Call APIs
WININET	Internet access functionality
ADVAPI32	Supports security and registry calls

explained in detail in the following section.

2.2 DLLs in Windows NT

Dynamically Linked Libraries (DLL) provide an efficient and systematic way for software development and distribution. Although DLLs have the same internal structure as regular binaries, they are not independent programs, but are modules that provide a common set of services shared by different programs. The Microsoft Windows OS makes extensive use of DLLs for various services. Common Windows-based applications also rely on a major portion of the libraries to perform their functions. System DLLs are loaded into physical memory only once and is mapped to the same location in the virtual address space of each process.

Shared DLLs are loaded into physical memory only once and are mapped to the address space of each process that uses them. Typically, operating systems align DLLs on page boundaries. Table 1 shows a few common system DLLs that are widely used by most applications on a Windows platform. Functions in KERNEL32 and NTDLL provide core operating system functions and are indispensable to all applications. The typical graphical desktop application will also employ USER32 and GDI32 for a common 'look and feel'. Other system DLLs are more specific to certain types of applications. For example, WININET that provides networking functionality is used by web browsers and RPCRT4 is used by applications for inter-process communication. The system DLLs shown in Table 1 are only a small subset of the hundreds of system DLLs provided by the Windows platform.

2.3 *Revisiting Simultaneous Multithreading*

The objective of SMT [26] architecture is to maximize overall throughput of instructions, by allowing independent threads to take advantage of a wide issue pipeline in a traditional superscalar machine as shown in Figure 2. Although this allows better utilization of resources, invariably single thread performance degrades. This is due to the fact that critical resources like caches, Load/Store queues and ROB are shared between multiple threads. For this reason, an effective SMT machine needs to not only improve throughput, but also needs to provide fairness to multiple threads that have different execution characteristics.

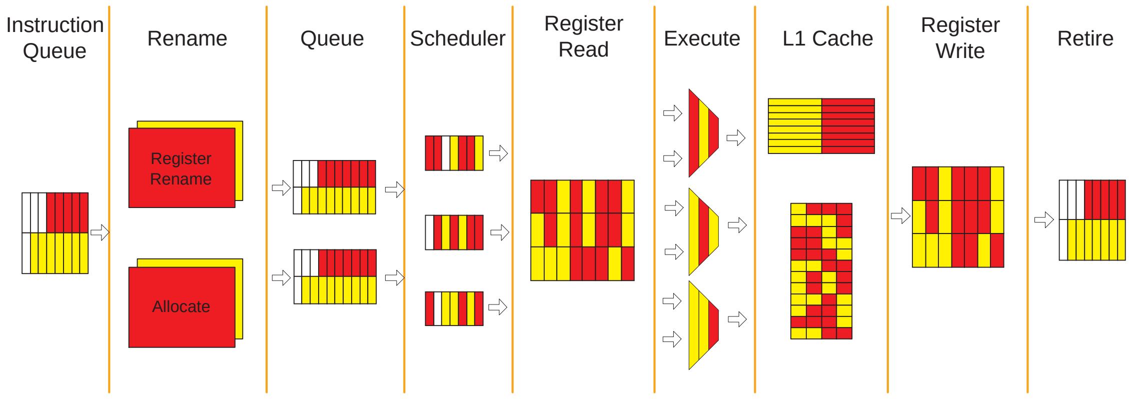


Figure 2: A 2-Way SMT Pipeline

In a normal superscalar machine, all thread and process level scheduling are performed by the Operating System. Modern operating systems use advanced scheduling policies and perform prioritization in order to ensure that threads are treated fairly and in the order of their importance. However, an SMT architecture exposes thread/process level semantics to the microarchitecture, therefore certain traditional assumptions in a superscalar design do not perform well or even result in performance degradation of single thread applications. To best exploit thread level semantics and provide increased throughput, SMT architectures need to be aware of system implications. One example of this aspect can be seen in the IBM Power5 that tightly integrates OS support to improve program execution. The IBM AIX OS allows the specification of thread level priorities to the microarchitecture so that scheduling can be prioritized for threads in the order of importance.

Shared libraries or DLLs is another overlooked arena that can potentially impact performance of SMT architectures. Traditionally, temporal locality exists in programs today due to the fact that the same instructions are accessed repeatedly. However, in the SMT scenario, temporal locality exists between multiple threads or processes due to the fact that code is reused at the software and platform level. In fact, on the Windows NT platform, the average application uses 30 DLLs of which two-thirds are typically shared by other applications. The fact that such locality is not exploited by traditional SMT architectures is the main motivating factor in this work. Most research in this area also uses the traditional SPEC benchmarks that do not use shared libraries as real world applications do. Future processor designs will continue to hoist microarchitecture specifications to the system level in the form of CMP, SMT or virtualization technologies. As long as this trend continues, architects will need to study and leverage system level implications to improve performance.

CHAPTER III

DLL THRASHING AND DUPLICATION

The instruction cache, like most other resources, is shared by independent threads in SMT processors. Since many modern processors implement virtually-indexed instruction caches [13, 5] to reduce latency we base our subsequent discussions on this assumption. Having a virtually indexed cache leads to the *homonym* problem that is dealt with by either flushing the cache on a context switch or the use of thread IDs. Since an SMT processor allows more than one thread to be active at any given point of time, each I-TLB entry or even the I-cache lines must include an additional thread ID field for ascertaining whether the accessed line belongs to the requesting thread during an I-cache lookup.

Shared DLLs are loaded into physical memory only once and are mapped to the address space of each process that uses them. Typically, operating systems align DLLs on page boundaries. Table 1 shows a few common system DLLs that are widely used by most applications on a Windows platform. Functions in KERNEL32 and NTDLL provide core operating system functions and are indispensable to all applications. The typical graphical desktop application will also employ USER32 and GDI32 for a common 'look and feel'. Other system DLLs are more specific to certain types of applications. For example, WININET that provides networking functionality is used by web browsers and RPCRT4 is used by applications for inter-process communication. The system DLLs shown in Table 1 are only a small subset of the hundreds of system DLLs provided by the Windows platform.

Since the microarchitecture does not possess any knowledge about shared libraries, DLL instructions in the I-TLB and I-cache will not be treated any differently than other instructions. In other words, the instructions from the same DLL will appear multiple times in the I-TLB and I-cache labeled by a unique thread ID of their respective owner. Therefore, a process that uses a certain instruction from a shared DLL will miss in the I-cache, when in fact, the required information is already present at the desired location. For the Microsoft

Table 2: DLL Distribution in Windows Applications

Application	**Total Instructions (In Millions)**	**System DLL Instructions**
Acrobat Reader	410	14.6 %
Internet Explorer 5.0	446	15.3 %
Netscape Navigator 4.7	432	17.4 %
PowerPoint 97	366	20.8 %
Visual C++ 6.0	398	11.4 %
Word 97	378	16.4 %

Windows OS, DLLs have preferred addresses in the virtual space. All incarnations of Windows NT have system DLLs loaded at fixed virtual addresses as explained in Chapter 2. This leads to a problem we define as DLL Thrashing. This problem is exacerbated in a direct-mapped I-cache which will generate false conflict misses due to the ping-pong effect among different threads that access the same DLL. For a set-associative cache, this results in a poor utilization of the I-cache in an SMT because of DLL duplication, each indexed by a different thread ID. The extent of the performance impact will highly depend on the type of applications that are concurrently running on the SMT processor and the amount of DLL instructions that are being shared.

As shown in Table 2, we profiled the *shareable system DLL* usage of several Microsoft Windows Applications. Depending on the application, the dynamic DLL instructions account for anywhere between 11% and 21% of the total dynamic instruction stream.[1] More discussion on the DLL distribution will follow in Chapter 6.

Ideally, the DLLs are to be shared among different processes, nonetheless, this is not the case in the hardware structures such as the I-TLB and the I-cache. The rigid separation due to the use of the thread ID in SMT falsely represents that the DLL instructions are attached to a specific process.

It is clear that DLLs provide several benefits in the software development process including standardizing software, promoting software reuse and better software porting and

[1]Please note that we used an older platform (Microsoft Windows NT 4.0) and applications from Microsoft Office 97 due to constraints posed by the platform emulator that was used to capture instruction traces.

maintenance. As applications begin to spend more time on code provided as standard libraries in the form of DLLs, the performance of each individual thread on SMT processors will likely deteriorate, if DLLs are not given any recognition at the microarchitectural level. Understanding and exploiting widely supported system features such as DLLs, though often unattended by processor designers, will provide more opportunities for performance improvement in the future.

In the context of DLL Thrashing, since system DLLs like KERNEL32 and NTDLL are used by all applications that run on Microsoft Windows platforms, there is a higher probability that system DLLs will thrash more often with each other in SMT processors. Since Application DLLs are specific to an application, the extent of thrashing will depend on the number of the homogeneous application threads that run concurrently on an SMT processor.

CHAPTER IV

DLL CONSCIOUS INSTRUCTION FETCH

In this section we propose a solution to address the issue of instruction cache thrashing and duplication in the presence of DLLs in SMT processors. Our technique involves providing the page table, the I-TLB, and the microarchitecture with additional capability to distinguish DLL instructions from regular instructions. By providing this enhancement, we can override the conventional PID (or thread ID) matching that takes place upon each I-cache lookup in an SMT machine. This will enable true sharing of DLL instructions among concurrently active threads for SMT processors.

Minimal support from the OS is also required for DLL instruction identification. As described earlier, shared DLLs are only loaded into a single physical memory location and are aligned on page boundaries. Because DLLs do not straddle page boundaries with non-DLLs, it can be ensured that distinct entries will be allocated for DLL pages in the I-TLB and Page Table. Keeping this in mind, we propose the following light-weight changes in the OS and the microarchitecture:

- A *DLL bit* (or L bit) in the page table that will indicate whether the corresponding page belongs to a DLL. The corresponding L bit will be also required in the I-TLB for each thread.

- A change to the OS page fault handler that will service the page fault as well as set the *DLL* bit for a DLL page in the page table.

- A *DLL* bit for each instruction cache line which will help identify a DLL instruction from an application instruction. This bit is required for a Virtually Indexed Virtually Tagged Cache to be explained in section 4.1, and is optional for a Virtually Indexed Physically Tagged Cache.

Figures 4.2 and 4.1 illustrates two variations of our proposed DLL sharing mechanism for

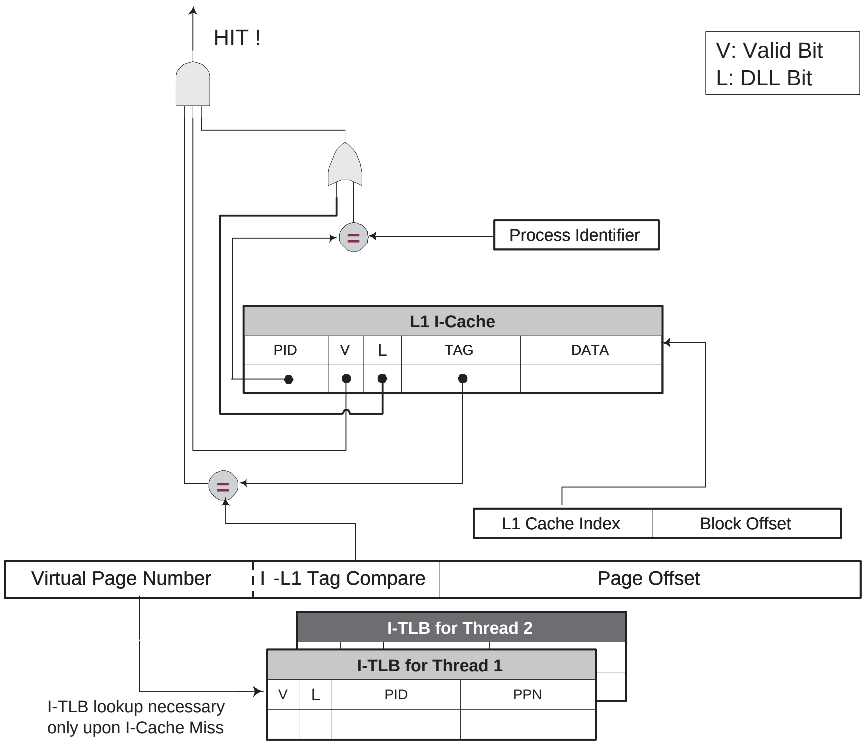


Figure 3: DLL-conscious Instruction Fetch - VIVT Cache

two different styles of a virtual I-cache organization. The modified cache lookup technique for both caches are explained subsequently.

4.1 VIVT Cache Optimization

Figure 4.1 shows a Virtually-Indexed Virtually-Tagged (VIVT) cache along with our microarchitectural modifications to enable DLL instruction sharing. A VIVT cache expedites lookup as the extra latency for address translation is not required. Only when a miss is encountered, is address translation required for accessing the L2, that is generally PIPT (physically indexed and physically tagged). This is why the I-Cache line, as well as the I-TLB include additional fields for the *L* bit and the thread ID (or PID). This L bit is set by looking up the I-TLB whenever a line is loaded into the I-Cache. If the access to the I-TLB misses, the page table for the process is accessed from memory. The page table entry has the *L* bit field and it is copied to the I-TLB. In case of a page fault, the OS page fault handler loads the page to memory and allocates a page table entry for it. At this time the handler sets the *L* bit of the page table entry if the page loaded is a DLL page.

The VIVT I-Cache acquires the virtual tag and index required for I-cache access directly from the virtual address and proceeds to look up the entry. A hit is generated when a tag matches and one of the following two conditions is true: (1) the PID matches the requesting process' PID; or (2) the L bit is set, in which the PID is ignored.

As shown, a PID field is a must need for each I-cache line of a VIVT cache to deal with the homonym aliasing in SMT processors, which carries the overhead of adding a PID in every cache line. Because of these drawbacks, even though VIVT caches could provide faster cache lookup, they are more expensive in terms of implementation. The trace cache in the Pentium 4 is an example of a VIVT cache in an SMT scenario that supports a PID to avoid homonym aliasing between threads in concurrent execution.

4.2 VIPT Cache Optimization

Virtually-Indexed Physically-Tagged (VIPT) caches provide an alternative by eliminating aliasing through the use of physical tags. Figure 4.2 illustrates a VIPT cache with our modified lookup technique. Since the physical tag comparison cannot be accomplished before the virtual-to-physical address translation is returned from the I-TLB, as shown in the figure, the *L* bit and the PID need only exist in the I-TLB, reducing some hardware overhead compared with a DLL-conscious VIVT cache implementation.

In the VIPT cache, I-TLB and I-Cache look-up happen in parallel. Upon acquiring the physical tag comparison result, the cache can determine if the necessary instruction exists at a particular location. Since the *L* bit will be set for DLL pages, a DLL instruction will hit in the I-Cache regardless of the PID. The major difference here is that the L bit and PID are obtained from the I-TLB, instead of the cache line. The MIPS R10000 [12] is an example of a processor that employs a VIPT cache and uses an ASID in the I-TLB for address space protection and to reduce the burden of context switch flushes.

The VIPT cache presents an interesting scenario in the presence of DLL instructions, as illustrated in Figure 5. For instance, consider an instruction that misses in the I-TLB for a DLL instruction. The OS will need to service this miss by providing the appropriate translation information to the I-TLB (represents *Phase 1* in Figure 5). In the usual scenario,

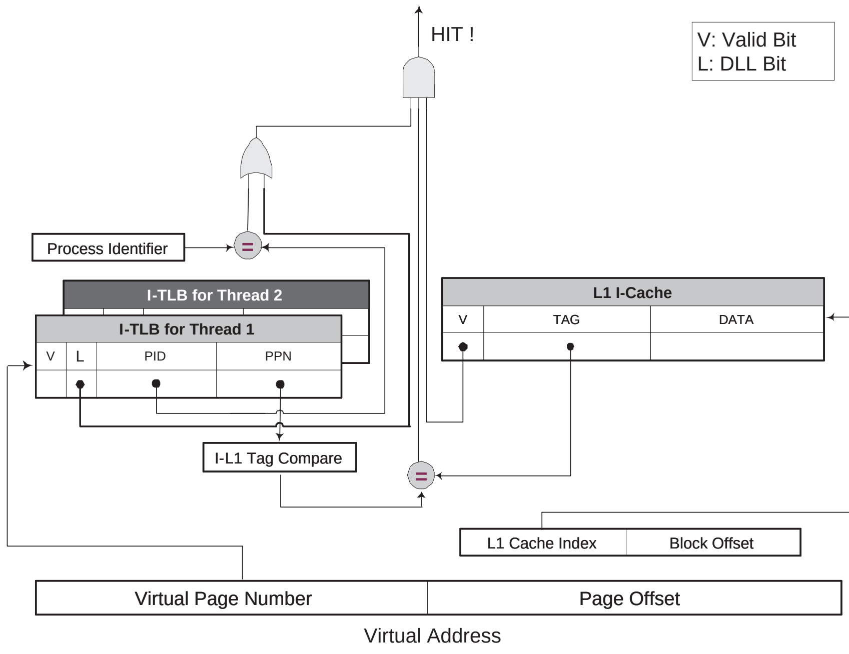


Figure 4: DLL-conscious Instruction Fetch - VIPT Cache

the subsequent cache lookup will also result in a miss, since the TLB has much higher spatial locality. However, in the case of a DLL instruction, it is possible that an I-Cache lookup (*Phase 2*) can result in a hit even upon an I-TLB miss, due to the fact that another process might be using the same DLL instruction. In other words, the system has paid the penalty of page table walk[1], obtaining the translation information, only to eventually hit in the I-Cache. Clearly, in the VIPT cache, the DLL-Conscious technique eliminates this inefficiency by skipping *Phase 1*, since a process independent translation from the I-TLB is used regardless of which process looks up a DLL instruction.

4.3 Design Implications

Sharing binary code has been a fundamental issue faced by software developers for decades. Different implementations of shared libraries in operating systems allow for efficient use of both physical and virtual memory. However, the advent of hoisting processor architectures to the system level (as in SMT or multi-core) will require processor designers to address

[1]The penalty of the page table walk depends on whether a software-managed or a hardware-managed TLB is used.

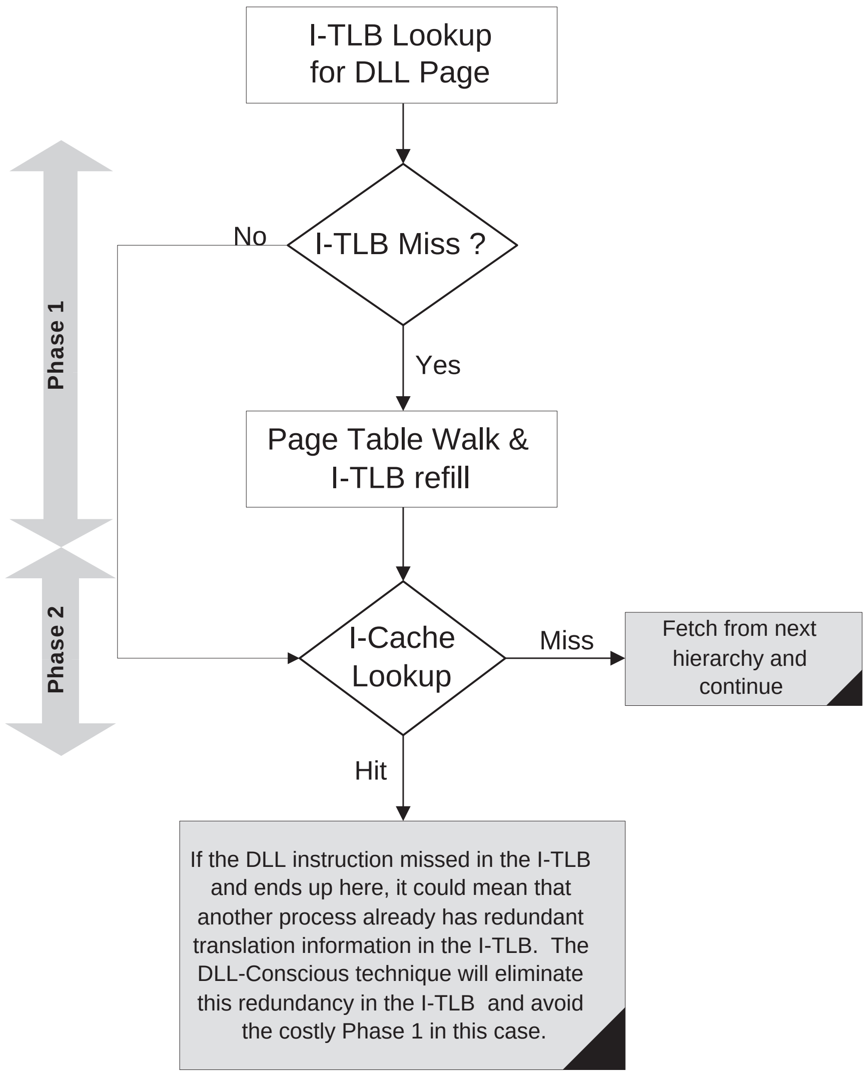

I-TLB Lookup for DLL Page
I-TLB Miss ?
No
Yes
Page Table Walk & I-TLB refill
I-Cache Lookup
Miss
Fetch from next hierarchy and continue
Hit
If the DLL instruction missed in the I-TLB and ends up here, it could mean that another process already has redundant translation information in the I-TLB. The DLL-Conscious technique will eliminate this redundancy in the I-TLB and avoid the costly Phase 1 in this case.
Phase 1
Phase 2

Figure 5: Lookup Process of a DLL instruction in a VIPT Cache

system issues for efficiency. Enabling true DLL sharing is an example of such an issue.

Certain processors employ a (G)lobal bit to prevent operating system code from being flushed out of the I-TLB and/or the I-Cache [10, 2, 1]. In such processors, the DLL-Conscious instruction fetch technique can be applied without any modifications to hardware. The (G)lobal bit can serve a dual purpose in this case. Since accessing system libraries do not cause address space violations, all accesses can be guaranteed valid. One question that arises is whether this can provide a security loophole whereby a malicious process can 'fake' a system DLL in order to cause an alternate process to execute unwanted instructions. However this is not usually possible, since pages for shared libraries are locked and under operating system control, thereby preventing deliberate intervention from external processes. For example, in the Windows OS, if a process tries to load data into a page reserved for a system DLL, the Windows OS will signal an *Invalid Page Fault*.

On another note, another possible method to overcome PID matching through software would be possible, if the OS could spoof a global PID reserved for DLLs. This way, all DLLs could appear to come from a single PID, hence avoiding DLL thrashing and/or duplication. However, this means that the OS would have to spoof PIDs for DLLs on an instruction by instruction basis, which is certainly not plausible. The light-weight microarchitecture technique we propose is truly necessary in order to reinstate true DLL sharing, in an SMT processor.

CHAPTER V

SIMULATION FRAMEWORK

In order to quantify the benefits of the DLL conscious instruction fetch, we required an evaluation methodology that could profile and simulate applications that made extensive use of shared libraries or DLLs. For this reason, traditional SPEC benchmark application simulation using SimpleScalar would not suffice. Furthermore, DLLs are prevalent on the Windows platform and applications making it necessary for us to use an evaluation methodology that could quantify the benefits of our technique on standard day to day applications that are used on the Windows platform.

To profile and simulate Windows applications, we used the TAXI framework [28]. An illustration of the TAXI framework along with our modifications for application tracing, SMT support and DLL address mapping is shown in Figure 5. Our framework constitutes the following components:

- System Emulator: A modified version of the open source Bochs emulator is used to model the target platform including the Windows NT 4.0 operating system and various applications that run on it. Also, we use the same settings to collect instruction and memory traces under full Windows system emulation.

- Performance Simulator: Since the system emulator only models functional behavior, detailed analysis at the microarchitectural level needs to be accomplished by TAXI, a trace-driven performance simulator that integrates an x86 front-end into SimpleScalar [3]. The microarchitecture parameters are summarized in Table 3.

5.1 Bochs System Emulator

Bochs is a highly portable open source x86 IA-32 PC emulator that runs on a variety of platforms. It can emulate an Intel x86 CPU, most commonly used I/O devices and a

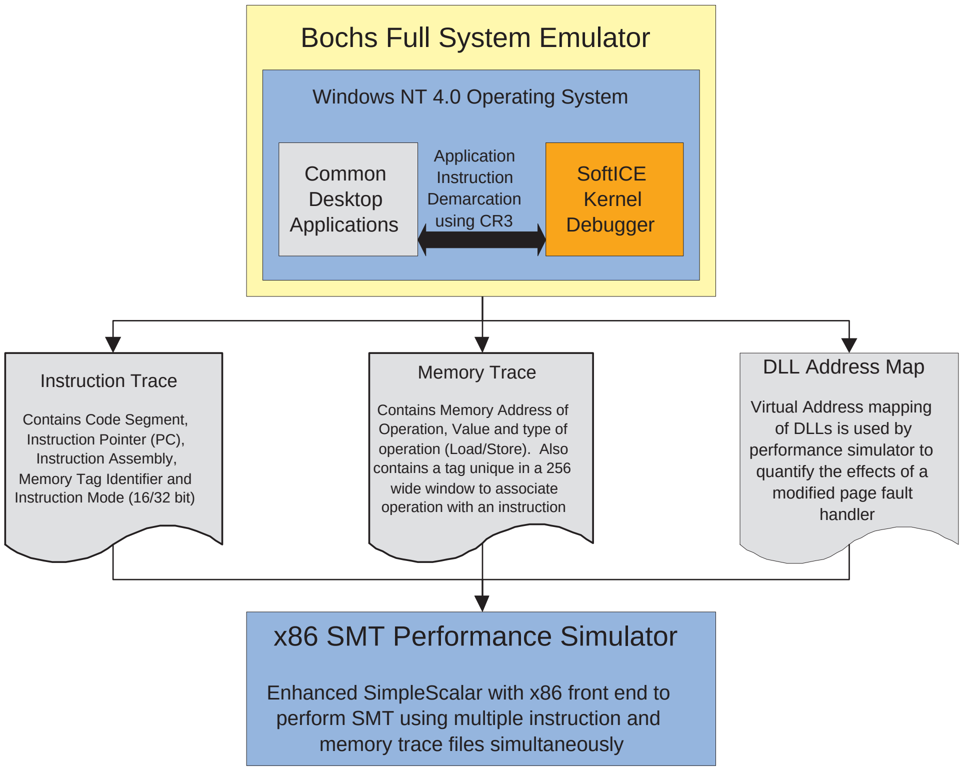

Bochs Full System Emulator
Windows NT 4.0 Operating System
Common Desktop Applications
Application Instruction Demarcation using CR3
SoftICE Kernel Debugger
Instruction Trace
Contains Code Segment, Instruction Pointer (PC), Instruction Assembly, Memory Tag Identifier and Instruction Mode (16/32 bit)
Memory Trace
Contains Memory Address of Operation, Value and type of operation (Load/Store). Also contains a tag unique in a 256 wide window to associate operation with an instruction
DLL Address Map
Virtual Address mapping of DLLs is used by performance simulator to quantify the effects of a modified page fault handler
x86 SMT Performance Simulator
Enhanced SimpleScalar with x86 front end to perform SMT using multiple instruction and memory trace files simultaneously

Figure 6: Enhanced TAXI Framework with SMT support

custom BIOS. It can run most operating systems unmodified including different versions of Windows and Linux. It is essentially a virtual machine that can run a given operating system on a host machine. For example, Bochs can run a completely unmodified Windows installation on a Linux workstation.

Bochs emulates every x86 instruction and all the devices in a PC system unlike other virtualization software like VMware. Due to this reason, it does not achieve high emulation speeds. However, because of the fact that all x86 hardware is emulated, it allows for detailed profiling of the complete operating system and applications that run on it. Bochs is generally used as a testbed for developing device drivers and running some legacy applications that are not available on later platforms.

The TAXI framework constitutes a slightly modified Bochs emulator that includes support for instruction and memory tracing. The modifications enable Bochs to trace instructions and memory operations. The modified Bochs emulator prints out an instruction trace that contains the effective address of the instruction (EIP), the instruction assembly and the code segment. In addition, each instruction also has a flag that indicates whether it is a 16-bit or 32-bit instruction, as well as a tag that is used to identify any memory operation that is associated with that instruction. All memory operations are recorded in a separate file. This file contains the tag to associated the operation with a given instruction, the memory address and the value that is to be stored or loaded.

We installed the Microsoft Windows NT 4.0 operating system on a Bochs disk image. Additionally we installed Microsoft Office 97 Suite, Adobe Acrobat Reader 6.0, Microsoft Visual Studio 6.0 and Netscape Communicator 4.7 in this targeted OS. Although emulating the latest Windows XP OS would have been desirable, time constraints posed by Bochs prevented us from doing so. Newer operating systems and applications have large working sets and since Bochs emulates every single instruction, emulation times proved to be too long to be usable.

For purposes of this project, we made some modifications to the Bochs emulator. Support was included to trace application specific code that would later enable us to study performance characteristics of application specific code only, without any operating system activity. This was done in a two step procedure. To demarcate application code, the CR3 register in the x86 architecture can be used. The CR3 register holds the physical address to the page directory base for any given running process. Since the address is physical, it can be guaranteed to be unique for a given process that is running on an operating system. By tracing instructions and memory operations only when a given context was running in the emulator, all third party code can be avoided from the trace.

To obtain the page directory base address of an application that we were interested in, a kernel debugger can be used. We used the SoftICE kernel debugger within Bochs and traced the application code in the following manner. As soon as an application is launched, SoftICE can be switched into debug mode. At this point, any running process can be queried on to obtain the CR3 value. After obtaining the CR3 value, emulation was halted within Bochs by bringing up the configuration menu. Instrumentation was then initiated for the given application from that point onwards. In this manner, application code alone was traced from the full system simulator.

We profiled six applications for this study. Applications were traced based on their common functionality as described below.

- Adobe Acrobat Reader: We opened two PDF documents that were between 5 — 10 pages each.
- Microsoft PowerPoint: We started a presentation consisting of 17 slides with text and graphics.
- Microsoft Word: Two MS Word documents with text and some figures were opened.
- Microsoft Visual C++ 6.0: We compiled a small console application.
- Microsoft Internet Explorer 5.0: We launched two web documents that consists of text and graphics. Pages opened were http://news.google.com and http://www.cnn.com.
- Netscape Navigator 4.7: We opened the same set of web pages used in Internet Explorer.

5.2 x86 Performance Simulator

In order to observe and study the benefits of the DLL-Conscious instruction fetch on an SMT architecture, a cycle-accurate SMT simulator was needed. Although, there are SMT simulators like SMTSIM [26], they cannot simulate bechmarks that use the x86 instruction set. The TAXI performance simulator integrates an x86 front-end on the out-of-order SimpleScalar simulator, by mapping x86 instructions into equivalent instructions in the PISA instruction set that SimpleScalar employs. To enable concurrent execution of multiple threads, the trace-driven TAXI performance simulator was enhanced to model an N-Way SMT machine.

In order to enable SMT, the existing front-end in TAXI was provided with additional state information to enable thread identification. Compared to an normal superscalar processor, an SMT processor maintains distinct architectural state for each thread. To this end, each thread maintains its own program counter and architectural register files were duplicated as well. The enhanced fetch unit can be fetch instructions and memory references

'

Table 3: Microarchitecture Parameters

Parameters	**Values**
Fetch/Decode width	4-wide & 8-wide
Issue/Commit width	4-wide & 8-wide
Branch predictor	2-Level GAg, 512 entries
BTB	4-way, 128 sets
L1 I-Cache	DM, 2way and 4way 16KB and 8KB
I-TLB	32 Entries
L2 Cache	4-way, Unified 256KB
L1/L2 Latency	1 cycle / 6 cycles
Main Memory Latency	120 & 350 cycles
ROB Size	48 & 256 entries

from multiple trace files depending upon the number of concurrent threads that needed to be simulated. The bandwidth at the front-end is shared by multiple cycle-interleaved threads. Similar to commercial SMT processors like the Intel Pentium4 HT and the IBM Power5, fetching from each thread was performed using Round Robin(RR). Although there are various other policies like the ICOUNT [27] that provide higher throughput in certain scenarios, the DLL-Conscious instruction fetch mechanism is transparent to this fact. The proposed technique is completely independent of any fetch policy that is used in an SMT processor.

The remaining hardware structures that were not critical to maintaining correct architectural state were shared between the threads. These includes all the Instruction Caches and Data Caches, Translation Lookaside Buffers, Register Update Unit and Load/Store Queue. The branch predictor was also shared, however a global history was maintained for each thread, in order to prevent threads from polluting the branch history. The traditional cache model in SimpleScalar and TAXI does not specify whether the caches are virtually or physically addressed. Since most commercial high-performance processors employ first level caches that are virtually addressed, the Cache and TLB in were provided with the additional PID field, in order to identify the thread that owned the line in the I-Cache/D-Cache or the I-TLB/D-TLB. Since the cache module is used for other structures like the BTB, they were also tagged with the PID. Along with this the baseline architecture performed static partitioning of the RUU similar to the ROB in the Intel Pentium4. This was done to ensure fairness and prevent certain threads from starving the others from contending for RUU entries.

In addition to the baseline machine model as described, the DLL-Conscious Optimization was also implemented by incorporating the L-bit in the I-Cache and the I-TLB as explained in Section 4. It was also necessary to simulate the effects of an enhanced page fault handler that can demarcate DLL instructions in the microarchitecture. To achieve this effect, DLL address mapping from the SoftIce kernel debugger in Bochs was provided to the performance simulator. This allowed the performance simulator to identify a DLL instruction as soon as it is fetched and update the L-bit in the I-Cache and the I-TLB. All experiments in the following chapter are performed by comparing the baseline SMT machine versus the SMT machine that employs the DLL-Conscious instruction fetch technique.

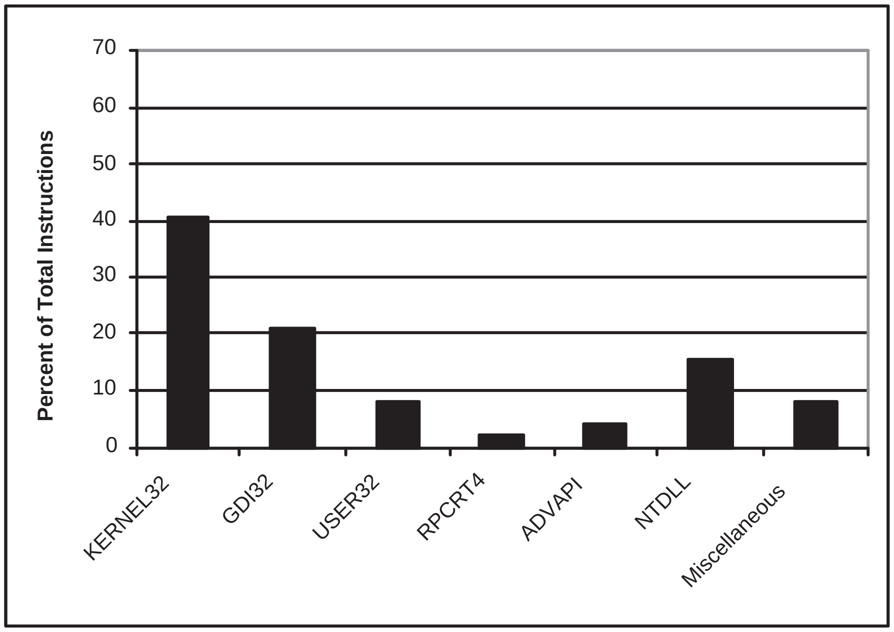
CHAPTER VI

EXPERIMENTAL RESULTS

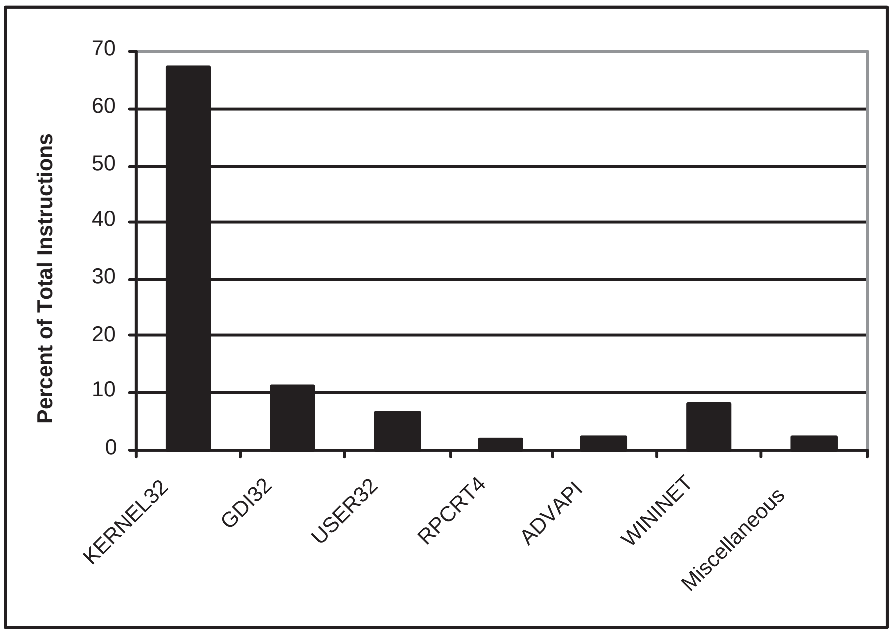
The DLL-Conscious Instruction fetching technique is useful for any architecture that supports multi-threading and uses an ID to distinguish between threads. This work deals only with SMT architectures, since it is an extreme form of multi-threading and effectively highlights the problem discussed. Since DLLs are a widely used form of shared libraries, this section enumerates and analyzes the results obtained on SMT machines running the Windows NT OS.

First, Figure 7 shows the breakdown of all DLL instructions for the applications we studied. The normalized distribution of different DLLs gives a good idea what DLLs contribute most of the execution time for each application. Common DLL characteristics indicate greater potential for sharing and will consequently benefit from our technique. For every application, there are a much larger number of KERNEL32 instructions than the other dependent DLLs. On an average, 52% of the total dynamic DLL instructions came from KERNEL32. This is expected since this DLL provides memory management and I/O functions that are critical to most applications. For Netscape Navigator, Visual C++ 6.0 and Adobe Acrobat Reader, NTDLL also represents a major portion of the dynamic instruction stream. Applications that use OS support frequently for functions like multi-threading will spend more time in NTDLL. With these two DLLs making up more than 60% of the total DLL instructions, the rest of the instructions were non-uniformly distributed between modules like GDI32 and USER32 which provide functionality for 2-D graphics and windowing/messaging respectively.

We now analyze the effect and extent of DLL instructions thrashing by analyzing the number of DLL misses encountered by each running thread, overall I-Cache hit rates and IPC improvement. For all simulations, we used a Round Robin(RR) fetch policy for fetching from different threads. We used the RR fetch policy because of the fact that the standard

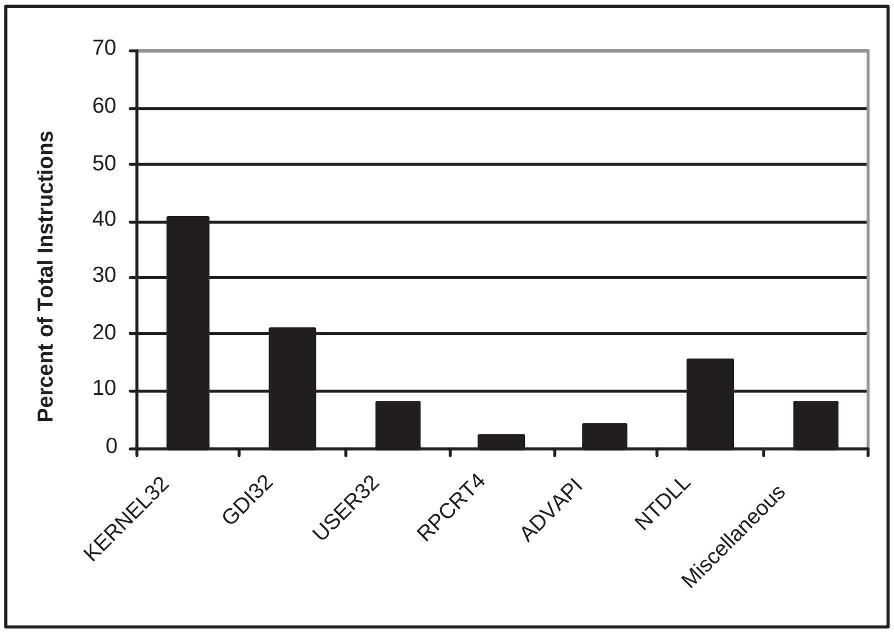


(a) Adobe Acrobat Reader 6.0

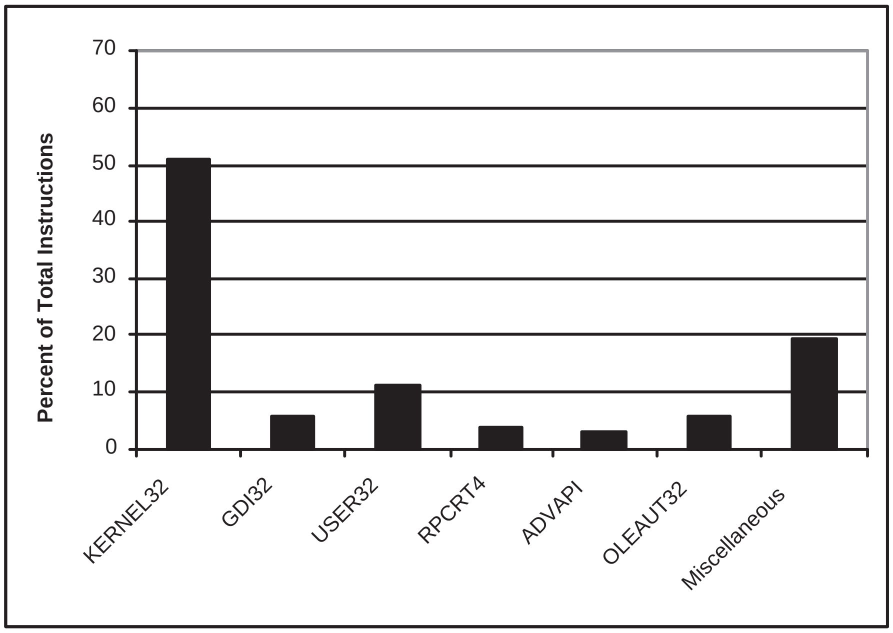


(b) Microsoft Internet Explorer 5.0

Figure 7: Distribution of DLL usage

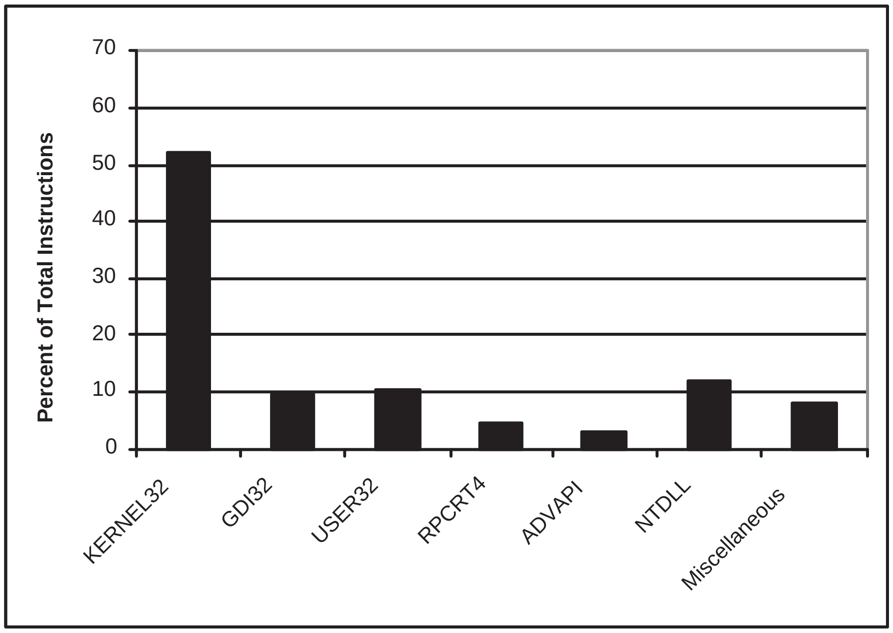


(c) Netscape Navigator 4.7

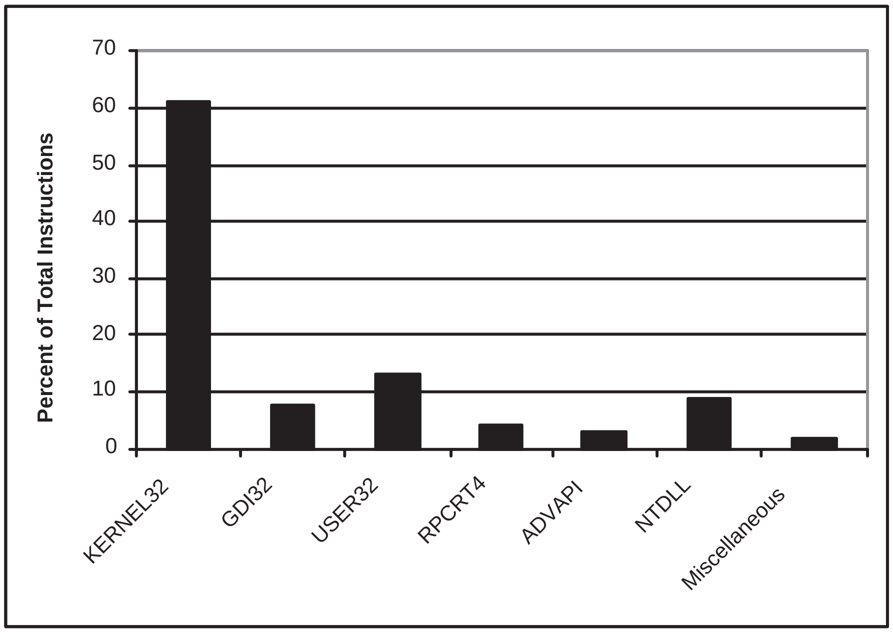


(d) Microsoft PowerPoint 97

Figure 7: (continued)



(e) Visual C++ 6.0



(f) Microsoft Word 97

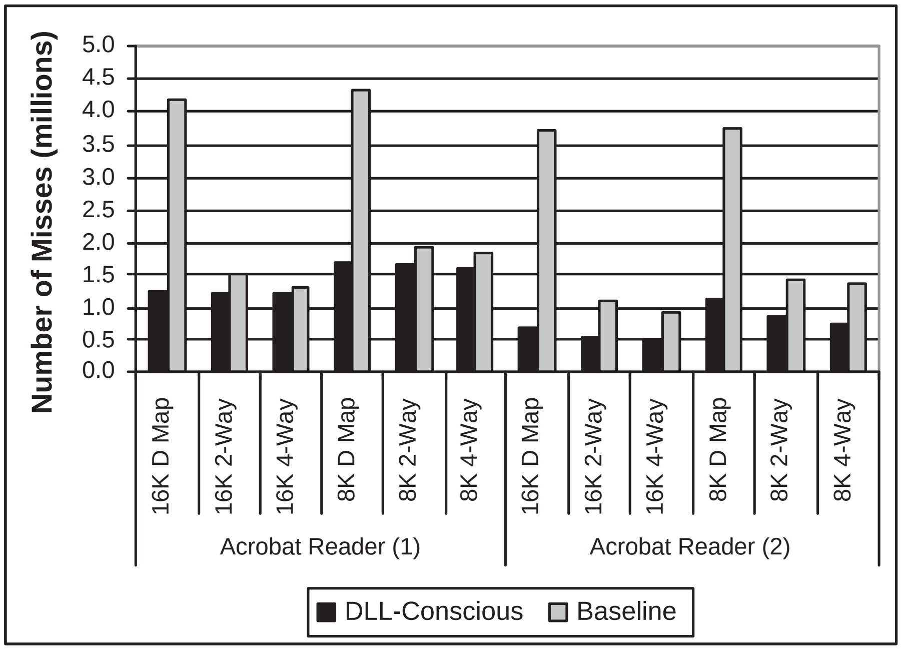
Figure 7: (continued)

Windows applications that we profiled do not exhibit IPC characteristics that are significantly different from each other. The most prevalent SMT processors today, the Pentium4 and the Power5, also uses the RR fetch policy. Although fetch policies like ICOUNT [27] can sometimes attain higher throughput in the scenario where one thread is running significantly faster/slower than the others, our technique is transparent to this fact. The DLL-Conscious instruction fetch technique is completely independent of any standard fetch policy used in an SMT processor. We simulated between 400 and 580 million instructions in our trace-driven simulator for different cases of a 2-Way and a 4-Way SMT.

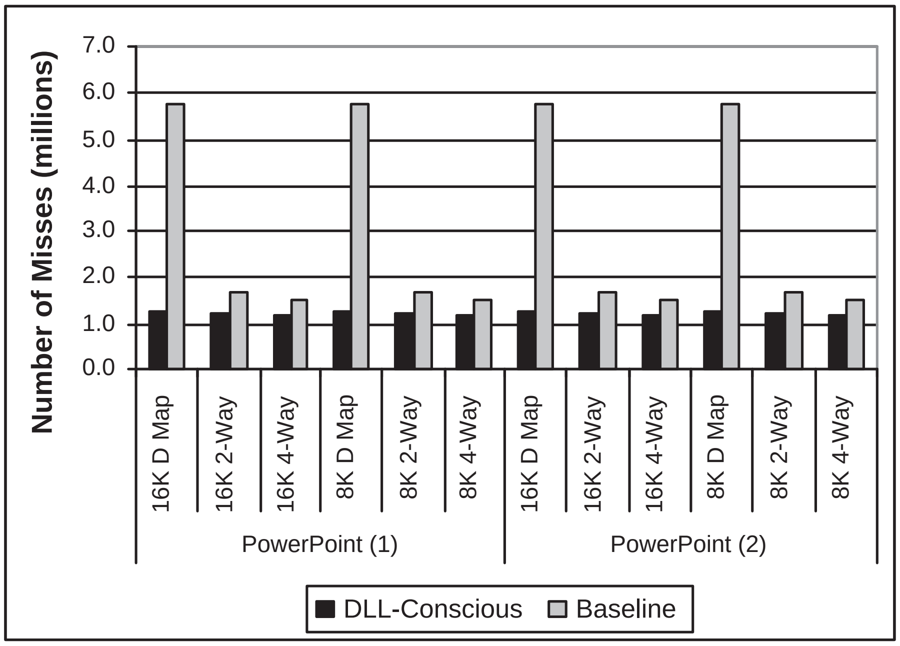
6.1 Analysis for a 2-Way SMT machine

Figure 6 shows the number of DLL misses for different IL1 configurations in a 2-Way SMT processor. This type of analysis is used for two reasons. The first being that it focuses specifically on the benefits gained from our proposal and highlights absolute improvements using our technique. Secondly, it presents an individual analysis of DLL misses per process and shows the degree of symbiosis, i.e. the extent of DLL thrashing between certain types of applications. There are a couple of different scenarios which determine the extent of DLL sharing between the running threads. The first scenario involves running homogeneous threads, that is, both threads are from the same application. However, the two threads represent two runs on the same application with two different working sets instead of simply simulating the same trace twice. The remaining scenarios use two threads involving different applications.

Note that we present a fair amount of analysis on homogeneous running threads under Windows. It has been observed that on a standard Windows machine, there are usually at least 30 or more background services and application processes running at any given time. Given a multitasking environment, there are usually multiple copies of the same background service that appear as separate processes, or even distinct processes of the same application that are running on a given system. User applications like the current version of Microsoft Excel and Internet Explorer spawn new processes with different PIDs for different instances of the same application. Hence, concurrent execution of homogeneous threads is fairly likely

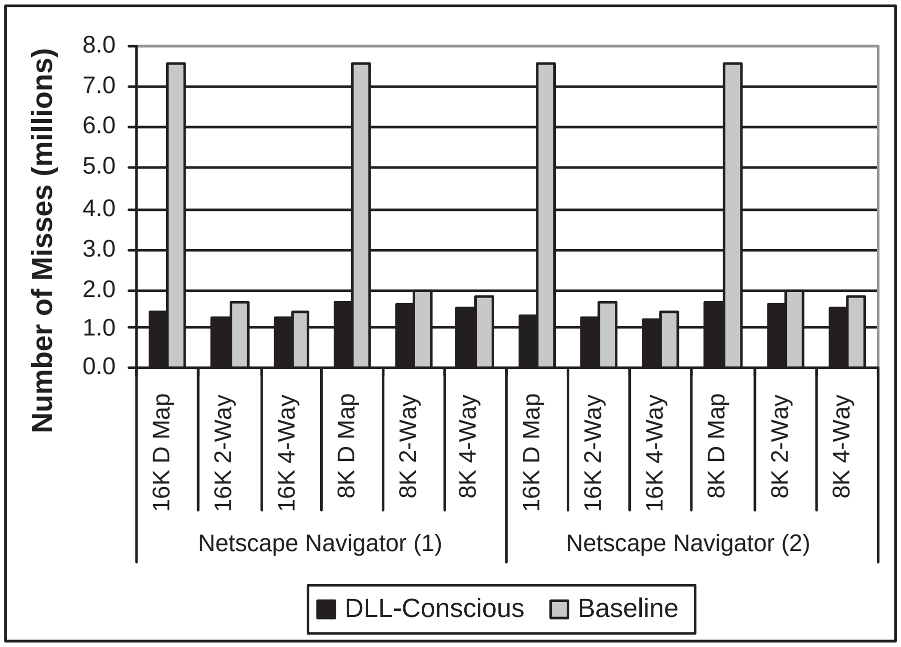


(a) Acroread and Acroread

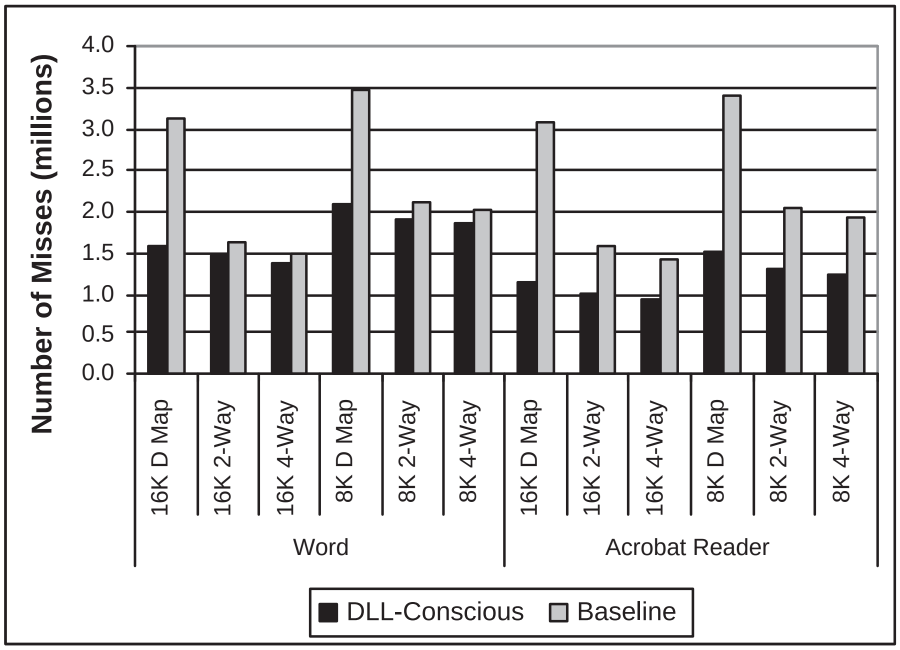


(b) PowerPoint and PowerPoint

Figure 8: DLL Misses per Thread for a 2-Way SMT

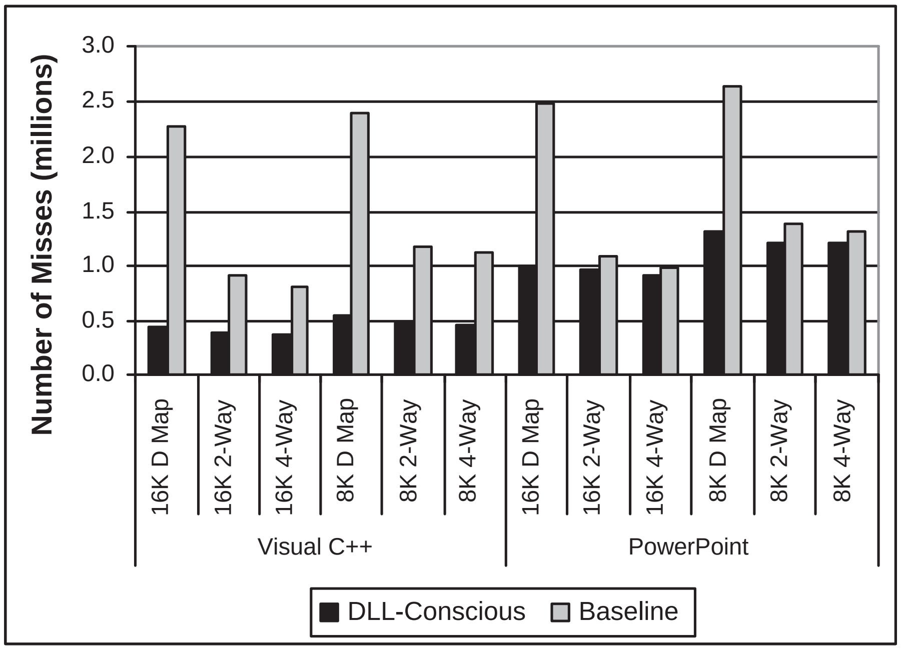


(c) Netscape and Netscape

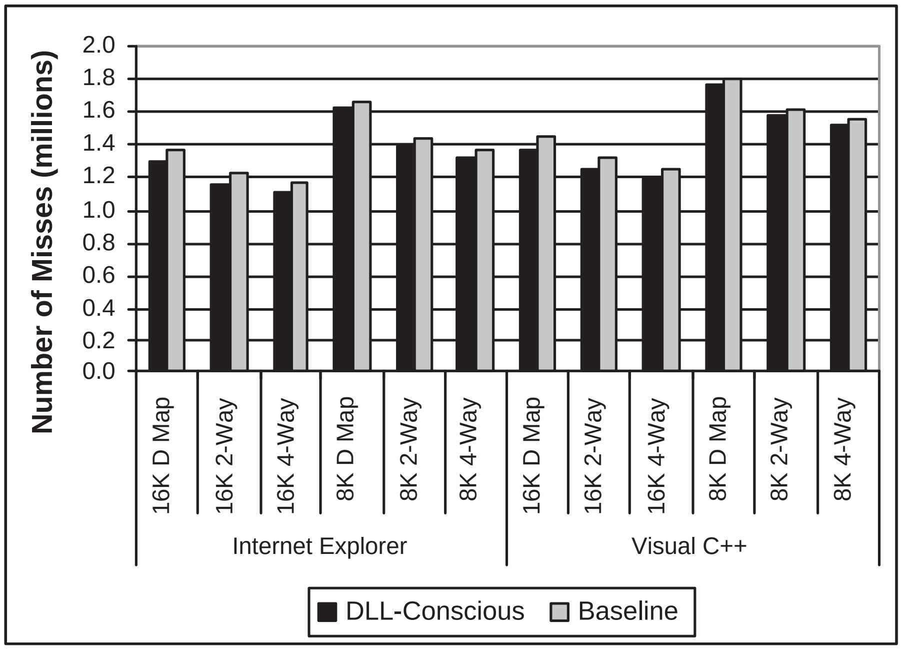


(d) Word and Acroread

Figure 8: (continued)



(e) Visual C++ and PowerPoint



(f) IE5.0 and Visual C++

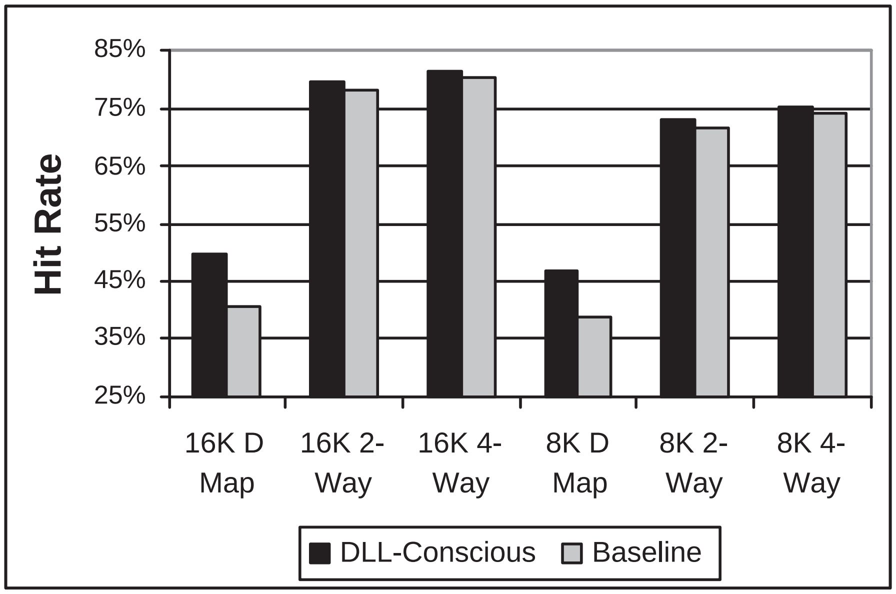
Figure 8: (continued)

to occur on a standard Windows platform.

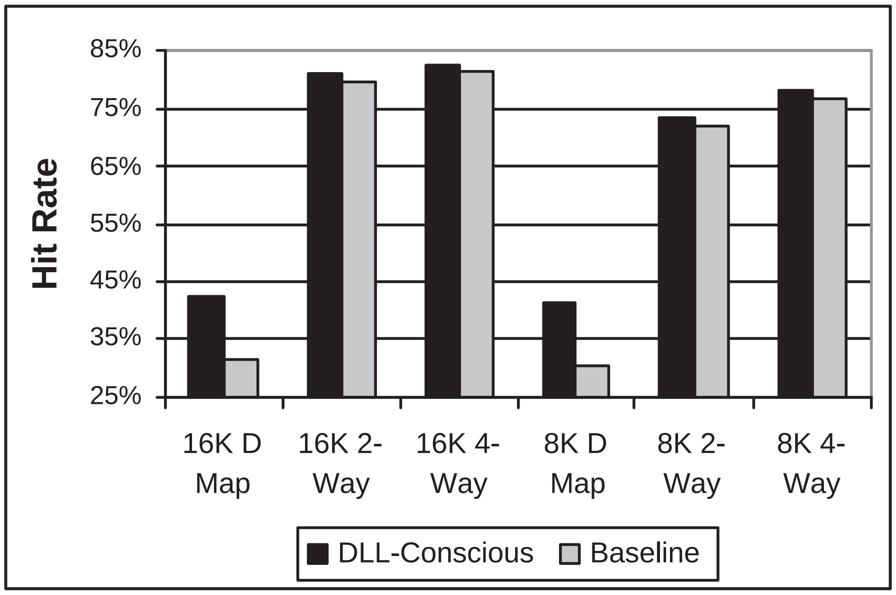
Figures 8(a), 8(b), and 8(c) represent the number of DLL misses per process for running homogeneous threads, e.g. two identical copies of PowerPoint. This is the case which we expect the maximal possible DLL sharing to be received as both applications have the same dependent and shareable DLLs. It is obvious that without any knowledge of shareable DLLs, both homogeneous threads will thrash each other constantly. The worst thrashing case is observed in Netscape Communicator. Note that Netscape also receives the best improvement using our technique. From these 3 figures, we also notice that although we reduce the number of DLL misses in all cases, the most pronounced gains come from the direct mapped I-cache, which reduces the total number of DLL misses by 3.3 to 5.4 times for Netscape. It is also observed that the DLL-conscious technique significantly reduces the DLL instruction miss rate from 71.6% down to 12.8%. Another interesting observation is that the total number of DLL misses using a direct-mapped cache with our technique is even lower than using a 2-way or 4-way set associative cache that does not employ our technique. Also note that a direct-mapped cache can offer a lower latency for running at ultra high frequency at the potential expense of more conflict misses. With our technique, the number of conflict misses can be substantially reduced, making a direct-mapped cache implementation more attractive.

Figure 8(d), 8(e), and 8(f) show DLL miss results by running heterogeneous applications under an SMT processor. Even though these applications are different, they could share system DLLs provided by the OS or application DLLs if both applications were released by the same software vendor. Therefore, the performance gains will highly depend on how many instructions are effectively shared between application threads. As shown in figure 8(d), for the direct mapped case, we reduce DLL misses for both threads by approximately 3 times.

Figure 8(e) shows Visual C++ running together with PowerPoint. Visual C++ is observed to have DLL misses reduced by 5 times, while PowerPoint DLL misses are reduced by approximately 2.5 times. The miss rate for this technique goes down to 6.4% (DLL-Conscious), from 22.1% (baseline). This result is particularly interesting because we see

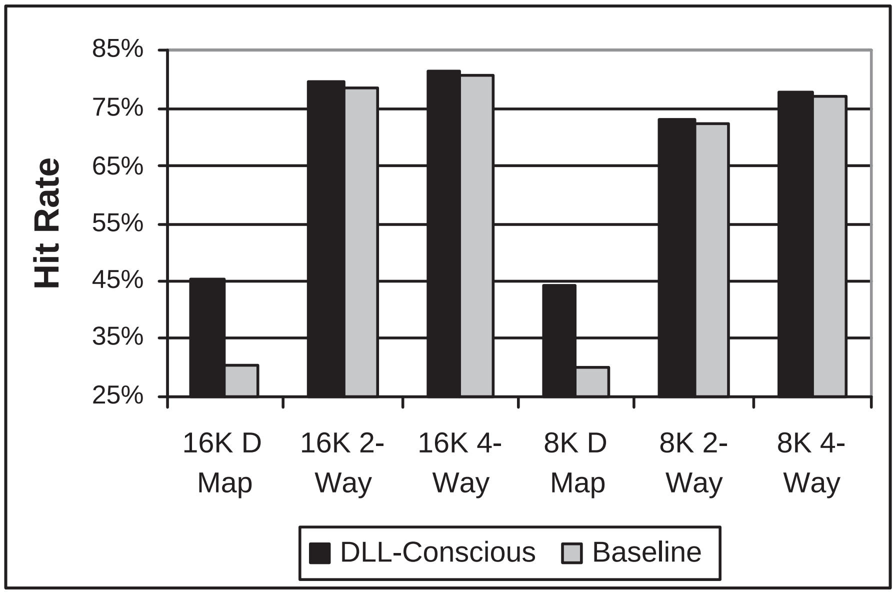


(a) Acroread and Acroread

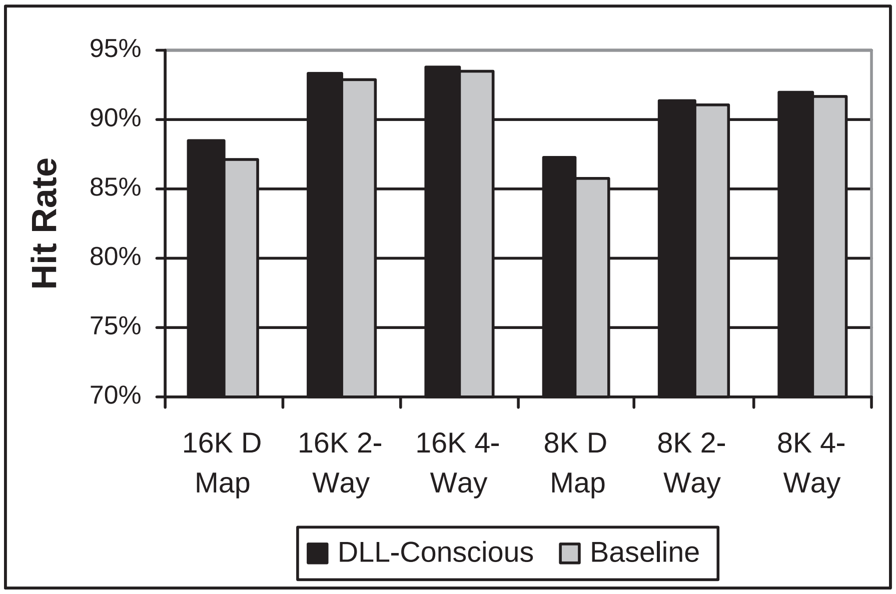


(b) PowerPoint and PowerPoint

Figure 9: I-Cache Hit Rate for a 2-Way SMT

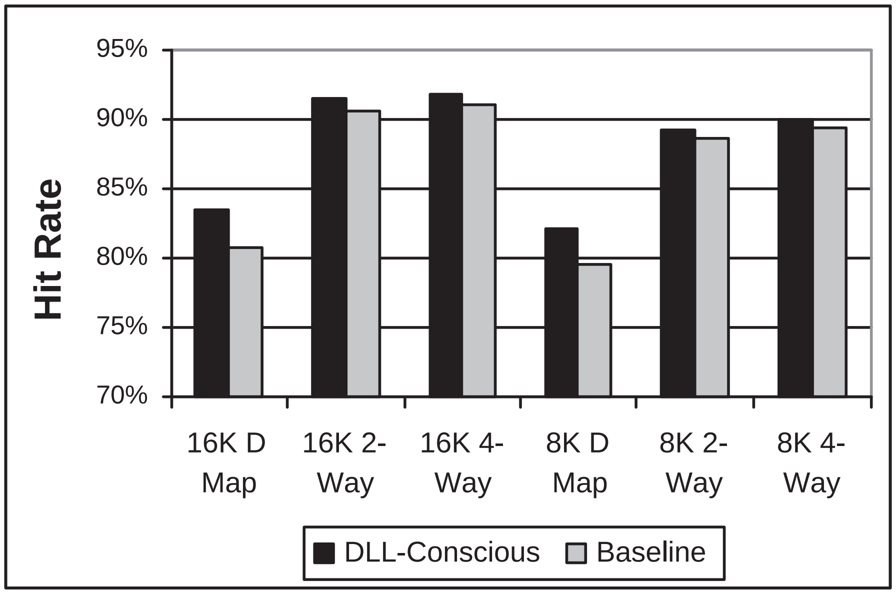


(c) Netscape and Netscape

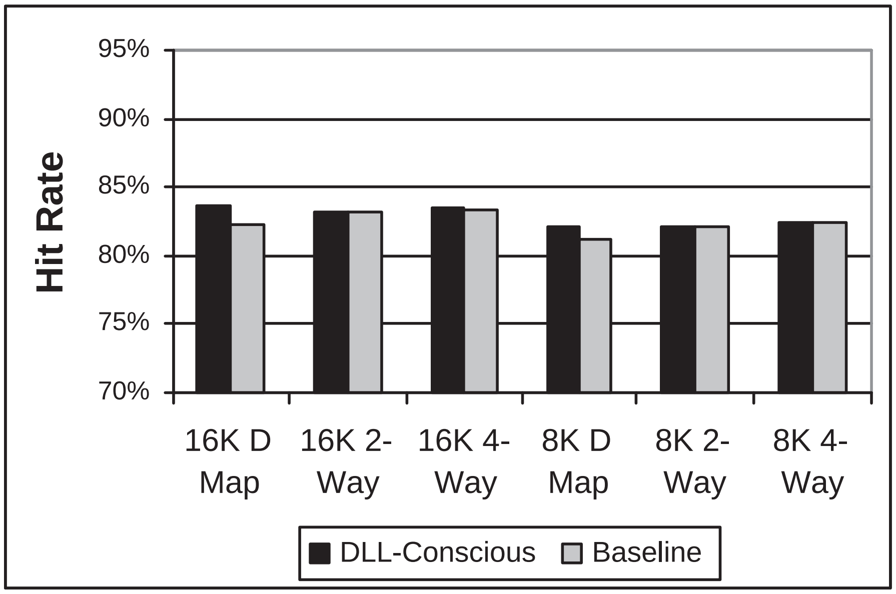


(d) Word and Acroread

Figure 9: (continued)



(e) Visual C++ and PowerPoint



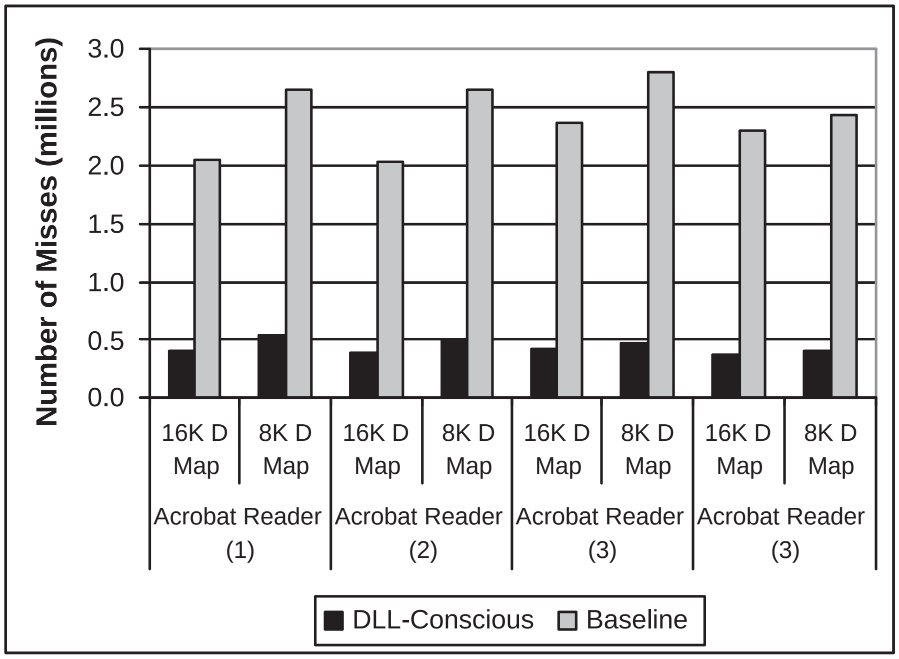
(f) IE5.0 and Visual C++

Figure 9: (continued)

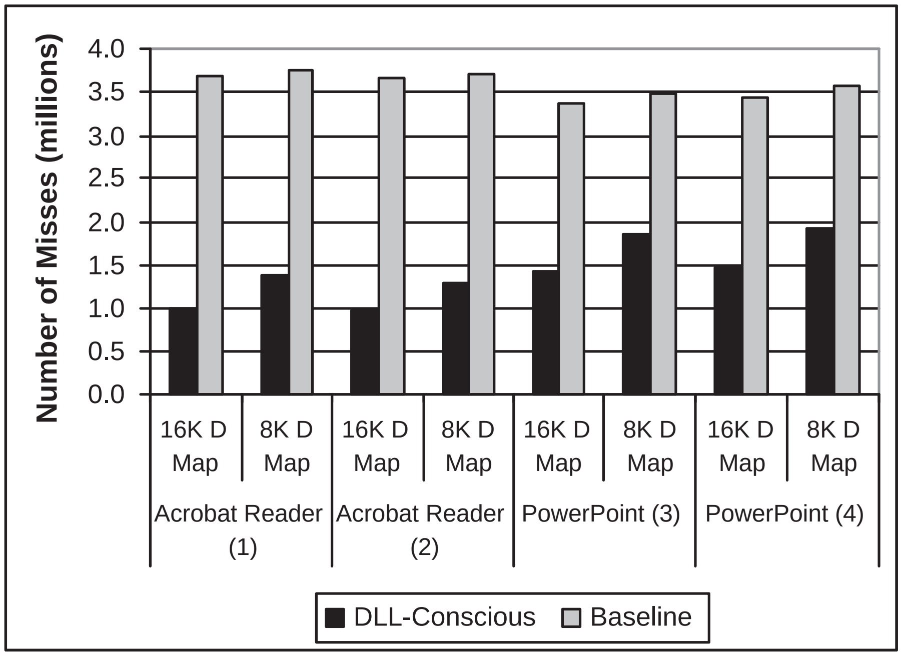
that Visual C++ reduces the number of DLL misses by a larger factor than in the case where we ran two instances Acrobat Reader or PowerPoint. This is due to the fact that even though there will be a significant amount of DLL instructions that thrash when running similar threads, some applications will suffer from a higher extent of DLL instructions that conflict with non-DLL application instructions. Visual C++ has a smaller DLL footprint as shown in Table 2 and also does not appear to conflict with non-DLL code, hence more benefits are obtained.

On the other hand, the PowerPoint thread does not just execute more DLL instructions but also calls a much larger variety of DLLs. As indicated in Figure 7, 17% of DLL instructions in PowerPoint are from miscellaneous system DLLs. A larger number of instructions from many different DLLs can potentially result in more conflicts with non-DLL instructions, which is why the number of DLL misses reduced vary greatly from the other active thread(s). Running Visual C++ along with Internet Explorer 5.0 shows the least miss reduction in our 2-Way SMT experiment due to very limited sharing of DLLs. Although both applications depend a great deal on KERNEL32, they do not seem to contain conflicting instructions in cache sets. It is important to point out that, however, our mechanism will always alleviate DLL instruction conflicts when they exist and will never result in increased number of misses for DLL or non-DLL instructions, even in the worst case.

Next, we present the overall I-Cache hit rate to help us to understand the overall effect based on our DLL-conscious optimization in SMT processors. Results are presented for the same 2-Way simulations discussed earlier. As expected, direct-mapped caches demonstrate better overall improvements than set-associative caches. Also, homogeneous threads have a higher symbiosis in sharing DLLs, thus achieving better cache hit rates than the execution of heterogeneous threads. Overall, the best improvement lies in the case with two Netscape Communicators running shown in Figure 9(c), In which, the cache hit rate was boosted from 30% to 47%, a 57% increase. Similar improvements are observed in Figure 9(a) and Figure 9(b). We should note that since our scheme always reduces contention, it never increases the miss rate under any circumstance. Thus the only cost for having our scheme is the minimal extra hardware. Our scheme will never degrade performance, even when

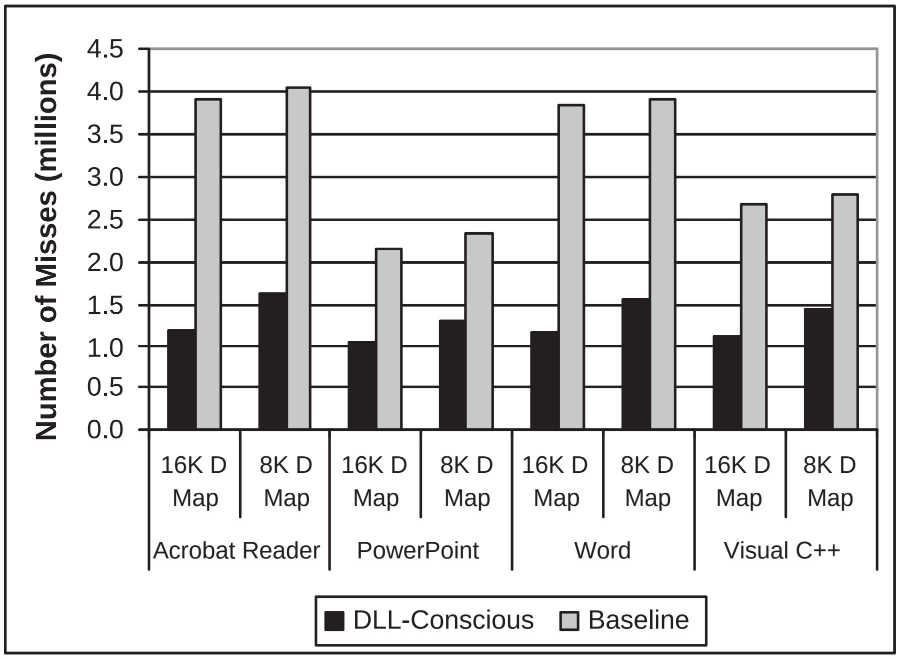


(a) Acroread - 4 Instances



(b) Acroread and PowerPoint - 2 Instances Each

Figure 10: DLL Misses per Thread for a 4-Way SMT



(c) Acroread, PowerPoint, Word and Visual C++

Figure 10: (continued)

DLL sharing is absent.

6.2 *Analysis for a 4-Way SMT machine*

We will now analyze the previously discussed metrics with 4 applications threads running concurrently. A larger number of application threads imply a greater potential for DLL instruction sharing. We quantified three scenarios for the 4-Way SMT. The first case involves four identical threads of Adobe Acrobat Reader, which represents the extreme case for DLL sharing. The second case runs two identical threads of Acrobat Reader and two of PowerPoint while the last case executes four heterogeneous application threads at the same time. Figure 10 shows the misses encountered by each thread in these scenarios [1].

Figure 10(a) shows that all Acrobat threads reduce the number of DLL misses by an

[1]Though it is expected that total number of ICache misses would increase for a 4-way SMT from a 2-way SMT, we see that the number of misses per thread have actually decreased. The reason behind this is that we simulated an average total of approximately 500 million instructions thus decreasing the number of instruction per thread for a 4-way SMT. Indeed, if we add up the misses of individual threads we would see that total number of misses for a 4-way machine is larger than that of a 2-way one.

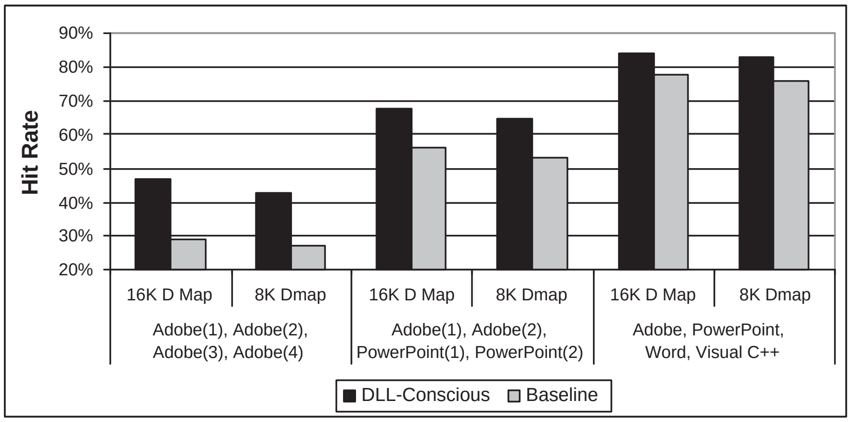


Figure 11: I-Cache Hit Rate: 4-Way

average of 5.5 times. Extensive sharing of the same system DLLs by these threads is the reason for this improvement. On the other hand, Figure 10(b) shows that Adobe Acrobat threads reduce the DLL misses by a factor of 3.2x while PowerPoint reduce the number of DLL misses by about 2.1x. Finally, Figure 10(c) shows an average DLL miss reduction of 2.8x, 1.9x, 2.9x and 2.1x for Acrobat Reader, PowerPoint, Word and Visual C++, respectively. As shown, even with four heterogeneous threads, we are able to reduce a significant amount of DLL misses, clearly indicating the large extent of DLL sharing among these applications.

The overall I-Cache hit rates for different cache sizes are presented in Figure 11. We observe that running four threads of Adobe Acrobat Reader thrash extensively on both DLL and non-DLL instructions resulting in low I-Cache hit rate. Consequently, the benefits gained from sharing DLL instructions between these threads increases the hit rate from 29% to 47%, a 62% improvement, for a 16KB direct mapped I-cache. Running two identical threads each of Acrobat Reader and PowerPoint with the DLL-Conscious technique results in a 22% improvement, while running four non-identical threads result in an average improvement of 9%. The benefits of our technique are clearly more pronounced when more threads are in concurrent execution.

6.3 IPC Improvements

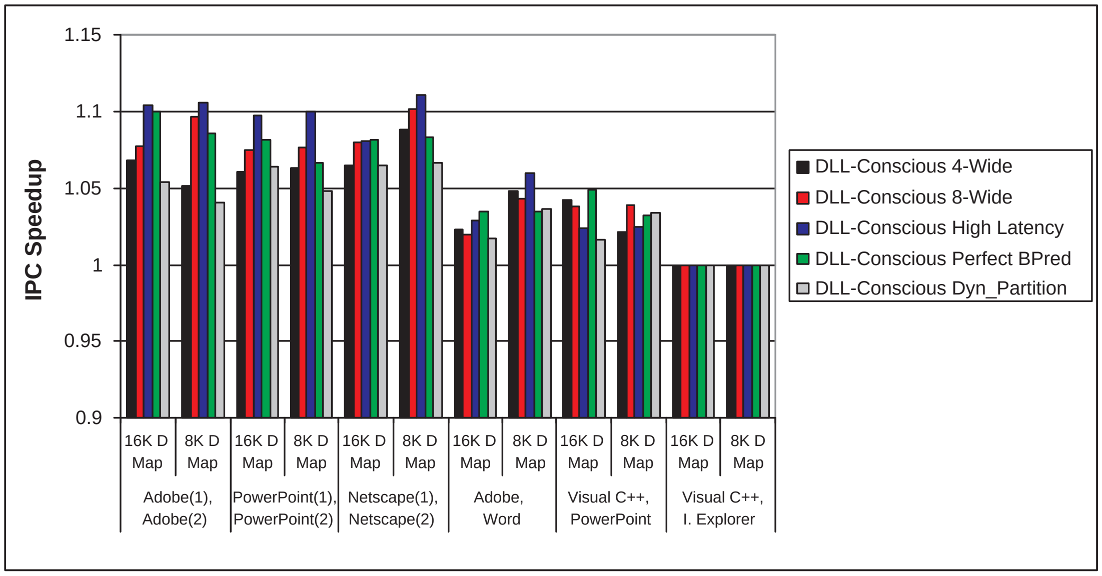
We will now analyze the IPC improvement gained by the use of our DLL instruction sharing technique in the direct mapped I-cache. We have quantified the overall IPC that constitutes all instructions as well as isolated the IPC of DLL instructions alone in all our simulations. The IPC improvements for 2-Way and 4-Way SMT machines are presented in Figure 12 and Figure 13, respectively. IPC observations were made for a few different configurations on top of the baseline micro-architecture parameters that were shown in table 3. Discussion on IPC observations are made below.

6.3.1 Observations on a 4-Wide Machine

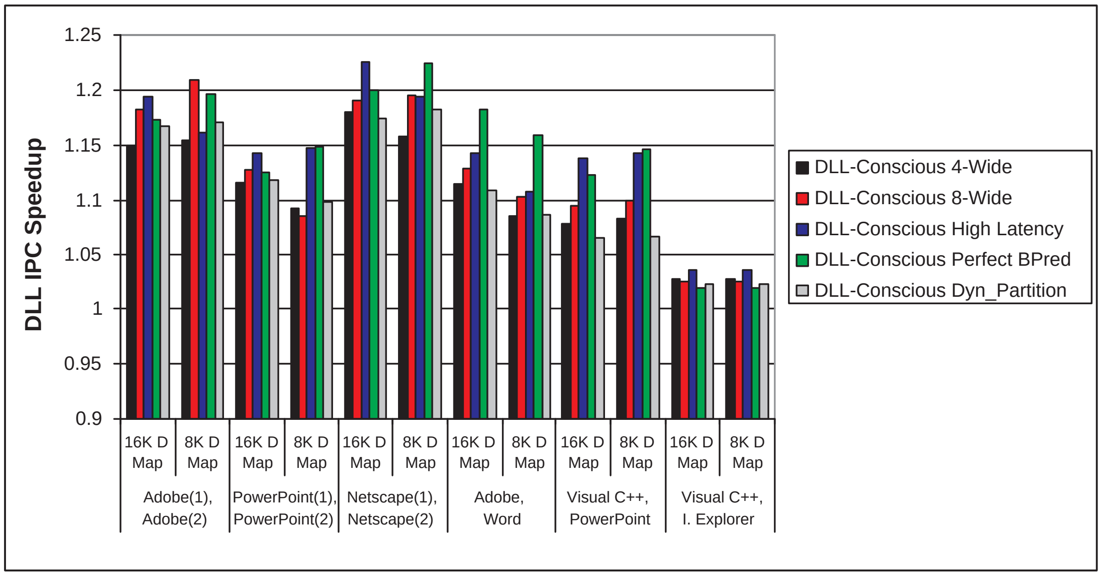
The first observations were made on the baseline parameters shown in table 3. We observe that IPC is generally less than 1 for all Windows applications due in part to the CISCy x86 instructions. For 2-Way SMT machines, running identical threads of Netscape Communicator was shown to increase the DLL IPC by an average of 17% and the overall IPC by approximately 7%. This was followed by Adobe Acrobat Reader which showed a DLL IPC improvements of 15% and an overall IPC improvement of about 6%. The Visual C++ and Internet Explorer pair seem to provide the lowest benefits of all with an average DLL IPC improvement of 2.8% and no noticeable improvement in the overall IPC. This is consistent with Figure 8(f) where it is clearly seen that there is very little reduction in the number of DLL instruction misses while running Visual C++ and Internet Explorer concurrently. For 4-Way SMT machines, greater benefits are observed due to the larger potential for sharing. DLL IPC increases by as much as 21% while running identical threads of Adobe Acrobat Reader, while the overall IPC increases by about 10%. With four non-identical threads, we notice an average improvement of 11% in the DLL IPC and 4.7% in the overall IPC is observed as well.

6.3.2 Observations on an 8-Wide Machine

In order to understand the implications of ignoring DLL instructions on future superscalar architectures, we studied the benefits that are to be gained on processors that have plenty



(a) Overall IPC - 2-Way SMT



(b) DLL IPC - 2-Way SMT

Figure 12: IPC Improvements for 2-Way SMT

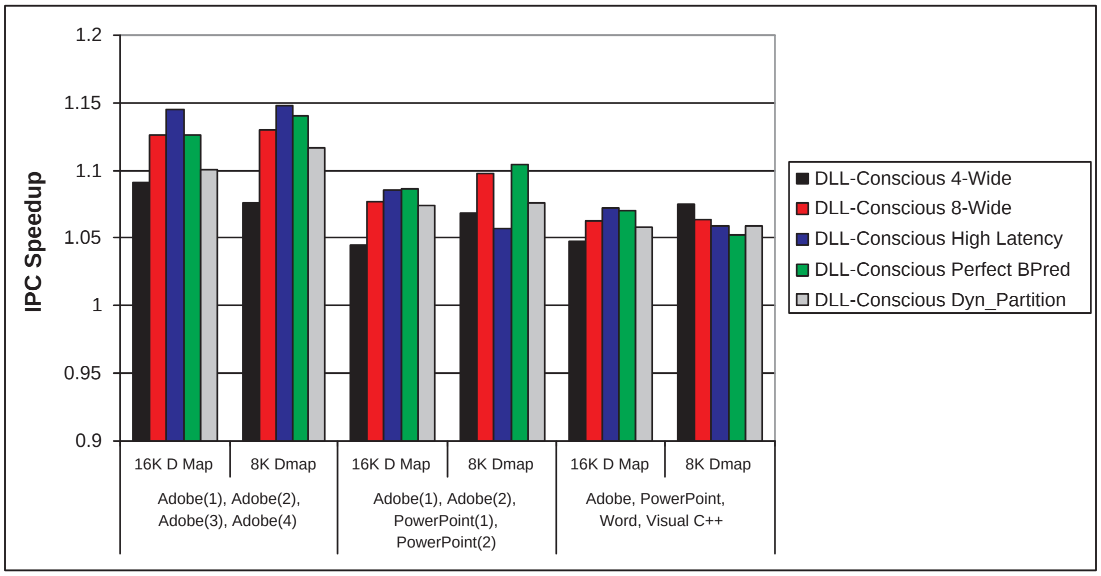
of resources and are capable of higher instruction issue. Our modified processor can fetch, decode, issue and commit 8 instructions(as opposed to 4 in the earlier configuration). In order to support the wider machine, we also increased the ROB size to 256 entries and doubled the load-store queue to 64 entries(up from 32). We study this case because future SMT processors will be wider and will suffer from less resource contention due to concurrent threads, making the cache performance even more important.

Because the 8-Wide machine suffers lesser resource contention, higher performance gains are obtained by enabling DLL sharing. For 2-Way SMT, identical threads of Netscape Communicator improved the DLL IPC by 21% and overall IPC by approximately 9.5%. Homogeneous threads of Acrobat Reader showed DLL IPC and overall IPC improvements of 18% and 8%. As in the earlier case, heterogeneous threads show lesser benefits even on the 8-Wide machine, because of lesser DLL sharing, with the Visual C++ and Internet Explorer pair that show the least benefits with an improved DLL IPC of approximately 3%. On a 4-Way SMT configuration, running 4 homogeneous Acrobat threads show an improvement of 23% in the DLL IPC and more pronounced overall IPC improvements of approximately 14%. Improvements with heterogeneous threads also increase, where we see about 14% improvements in the DLL IPC and 6.3% in the overall IPC. Overall, we see that an 8-wide machine is more sensitive to I-Cache misses and that significant performance gains can be accomplished by enabling DLL sharing using our technique.

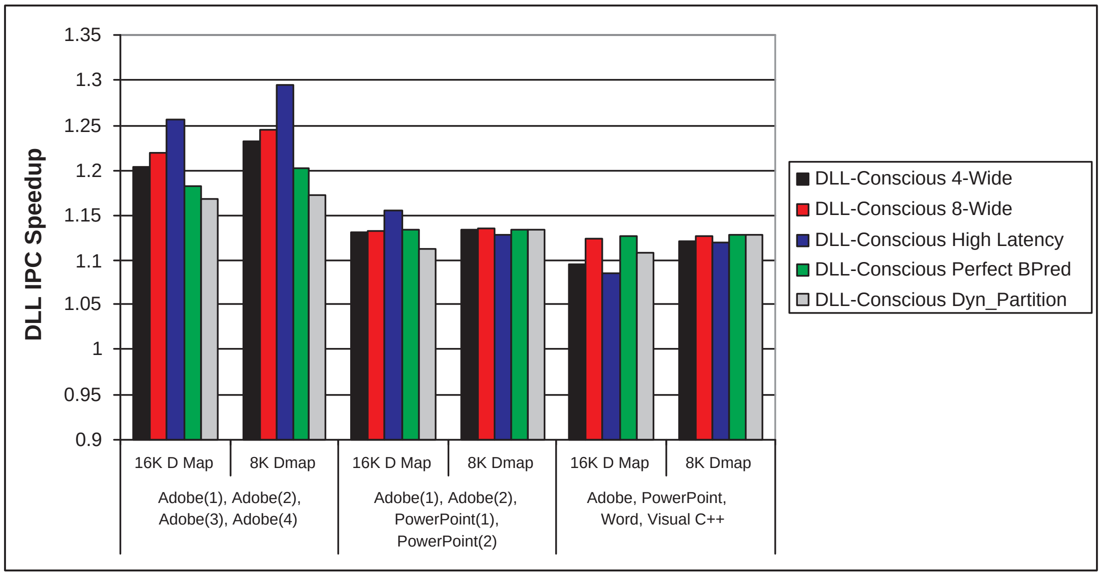
6.3.3 Sensitivity of Memory Latencies

In addition to being wider, future processors that operate at ultra-high frequencies will also have a wider memory gap. To this end, we have quantified the IPC benefits on a processor with higher cache and memory latencies. From the baseline parameters shown in Table 3, we increased the L1 latency to 2 cycles, the L2 latency to 10 cycles and the main memory latency to 350 cycles. Note that the changes were performed for the 4-Wide machine (instead of the 8-Wide machine).

Cache misses become more costly by increasing memory latency. Since our technique reduces cache misses, we get more performance improvement over the baseline if memory



(a) Overall IPC — 4-Way SMT



(b) DLL IPC — 4-Way SMT

Figure 13: IPC Improvements for 4-Way SMT

latency is increased. This results are shown in Figure 12 and Figure 13. Best improvements on 2-Way SMT for two threads of Netscape are approximately 23% for DLL IPC and 11% for overall IPC. Acrobat Reader threads on 2-Way SMT gain about 19% in the DLL IPC and 11% in the overall IPC. Homogeneous threads of PowerPoint also gain sizable benefits with an average DLL IPC improvement of 17% and overall IPC improvement of 9.4%. We observe even better improvements on a 4-Way SMT processor with high latency. With four threads of Acrobat Reader, DLL IPC improvements are improved by up to 30% and overall IPC increases by 15%, in the best case using the 8K cache. Even running four different applications shows a DLL IPC improvement of about 15% and overall IPC by 6%. From the system perspective, the implication of DLL Thrashing is significant, especially on future processors with higher memory latencies.

6.3.4 Sensitivity under Perfect Branch Prediction

To quantify the upper limit of applying the DLL-Concsious technique to SMT processors, we have also observed IPC benefits under perfect branch prediction. Applying perfect branch prediction will alleviate the front-end even further, and helps understand the implication of the proposed technique on a system that is sensitive to I-Cache misses.

We observe better absolute improvements in IPC under perfect branch prediction. Even though the baseline IPC is higher, we observe a DLL IPC improvement of approximately 22% and overall IPC improvement of 9% for two homogeneous Netscape threads. For homogeneous threads of Acrobat Reader, we also observe DLL IPC improvements of approximately 17% and overall IPC improvements of about 8%. As expected, 4-Way SMT shows further IPC benefits where we observe 21% improvement in DLL IPC and 14% in overall IPC for homogeneous threads of Acrobat Reader. Even in the case for least potential sharing with four distinct applications, we observe that we can achieve approximately 13% improvement in DLL IPC and 6% improvment in overall IPC. In effect, we observe that as the front end bottleneck is alleviated in SMT machines, the processor becomes highly sensitive to I-Cache performance and the DLL-Conscious instruction fetch technique.

6.3.5 Sensitivity under Dynamic Resource Partitioning

In order to observe the effect of the DLL-Conscious technique under dynamic resource partitioning, we performed yet another sensitivity study that varies the RUU partitioning in our simulator. Our baseline machine performs static RUU partitioning similar to the Pentium 4 where the ROB entries are statically divided among the threads equally. In contrast, this sensitivity study performs dynamic resource partitioning whereby the first half of the RUU entries are equally divided among the threads. The remaining half is up for dynamic partitioning whereby threads can compete for the entries [21]. Other simulation parameters were similar to 8-wide machine in the earlier section.

We notice that in the best case for 2-Way SMT, homogeneous threads of Netscape Communicator attains a 18% improvement in DLL IPC and 6.8% in overall IPC. This is followed by homogeneous threads of Acrobat Reader that show an improvement of 16.6% improvement in DLL IPC and 5.4% in overall IPC. On the lower end, we notice that the usual Visual C++ and Internet Explorer obtain no improvements in overall IPC due to minimal DLL sharing. In the case of 4-Way SMT, we observe that homogeneous threads of Acrobat Reader obtain a 16.9% improvement in DLL IPC and a 10.1% improvement in overall IPC. Even in the case with four heterogeneous threads, we notice a DLL IPC improvement of approximately 11% and overall IPC improvement of 5.6%. In short, we notice that dynamic partitioning slightly increases the throughput IPC compared to the statically partitioned machine, which is the reason for the reduced percentage improvements.

CHAPTER VII

RELATED WORK

A large percentage of processor architecture research use the SPEC or Mediabench benchmark suites to evaluate their techniques. While these results are representative for some application classes, there were other studies focused on generic desktop applications. For example, the characteristics of Microsoft Windows NT applications were studied by Lee et al. in [15]. Their work showed distinctive characteristics of Win32 desktop applications that include larger working sets and heavy use of shared libraries or DLLs. Their findings included the fact that an average of 20 system DLLs are shared by the typical desktop applications running under Windows NT. Redstone et al. [22] and Flautner et al. [9] analyzed operating systems and their interactive application behavior on SMT and SMP machines. Their research underlined the importance of system level performance which is also the driving factor in our work.

Research to analyze system level performance and implication on architecture have been undertaken by many in the past [16, 22, 4, 6, 9, 8]. Lo et al. [16] proposed the use of two existing software techniques (page coloring and bin hopping) to reduce contention in a VIPT L1 cache caused by different running threads in Online Transaction Processing Applications (OLTP). Their work focuses on reducing conflict misses in the I-cache caused by address mapping conflicts in the I-cache. Page coloring or bin hopping could be used to remove conflicts in DLLs in SMT processors, but this will simply result in duplicates of shared DLL instructions to exist in different cache locations resulting in wasted space. Our technique on the other hand, removes false contention by sharing DLL instructions from a single location and at the same time enables other threads to exploit shared DLLs loaded by other running threads.

Vlaovic, Davidson and Tyson [29] proposed a DLL BTB to improve the performance of real world applications that heavily use DLLs. Their work focused on exploiting branch behavior exhibited by DLL calls, while our work focuses on taking advantage of DLL-based instruction sharing between different application threads.

Notably in MIPS architecture, a (G)lobal bit was introduced in the TLB to ignore the Address-Space ID (ASID) matching during a TLB lookup. The nG (not Global) bit in the ARM 1176 [2] architecture serves a similar purpose. The nG bit, determines if the translation is marked as global (0), or process-specific (1) in the TLB. For process-specific translations the translation is inserted into the TLB using the current ASID. The purpose of the G bit in MIPS and nG bit in ARM however, is to eliminate the side effects to the kernel routines caused by context switches. By setting the PGE bit in the CR4 register, the Intel P6 architecture [1] also provides a G bit in each Page Table Entry to prevent a TLB entry from being flushed during a context switch. In contrast, our work addresses the effect of shared DLLs that are used by distinct threads in concurrent execution on an SMT processor.

Many techniques have been studied for improving the performance of SMT processors in the past [21, 24, 25]. Tullsen et al. investigated the impact of different instruction fetch policies [27]. Kumar et al. proposed compiler techniques to remove inter-thread conflict misses in the I-Cache in multithreaded architectures [14]. Lo et al. proposed compiler optimization [17] to exploit benefits of SMT and also convert thread-level parallelism to instruction level parallelism via SMT [18]. The performance of an SMT processor is also sensitive to the type of application that runs on it. Since the execution resources of the processor are split and shared among threads, the performance of each individual thread is likely to suffer. To address this issue, studies in [21, 24, 25] were performed to investigate the performance impact of different resource partitioning methods, scheduling policies and symbiotic behavior of applications in SMT processors. Note that our work is complimentary and applicable to these prior art.

CHAPTER VIII

CONCLUSIONS

In conclusion, this work investigated a form of often-neglected contention encountered by SMT processors due to the use of shared DLLs. We propose a technique involving light-weight modification in the OS and the instruction fetching mechanism to eliminate this false contention. The contributions of this work are:

- The effect of DLL thrashing due to shared instructions using common Windows-based applications were analyzed and quantified.
- A light-weight technique was proposed to address the issue of shared DLL instruction thrashing for two types of virtually-indexed cache designs in SMT processors. The technique incorporates share bits in the I-TLB and/or the I-cache to reinstate DLL sharing among active threads.
- A simulation methodology for full system level simulation on SMT based systems was developed for this work. Using the Bochs system emulator and the TAXI framework that we extended, we were able to quantify the DLL behavior, the I-cache behavior, and the performance gain using our technique.

As more processors support SMT with larger number of concurrent threads, the effect of DLL thrashing among threads will continue to exacerbate and cannot be ignored. As state of the art OSs like Windows XP provide more standard DLLs, the DLL Thrashing and Duplication issue will worsen and more benefits can be expected from our technique. Modern upcoming platforms are not only going to provide system DLLs for critical functionality like memory management, but also for advanced capabilities like networking and inter-device communication. Our proposed technique that makes an SMT processor DLL conscious, will remove the performance loss from unnecessary false contention by reinstating true sharing of DLL instructions in hardware.

REFERENCES

[1] *IA-32 Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide*, 1997-2004.

[2] *ARM1176JZ-S Technical Reference Manual*, 2004.

[3] Austin, T. M., "Simplescalar tool suite." http:/www.simplescalar.com, Feb. 2005.

[4] Barroso, L. A., Gharachorloo, K., and Bugnion, E., "Memory system characterization of commercial workloads," in *Proceedings of the 25th annual international symposium on Computer architecture*, pp. 3–14, 1998.

[5] Chan, K. K., Hay, C. C., Keller, J. R., Kurpanek, G. R., Schumacher, F. X., and Zheng, J., "Design of the HP PA 7200 CPU," *Hewlett-Packard Journal*, 1996.

[6] Chen, J. B., Endo, Y., Chan, K., Mazieres, D., Dias, A., Seltzer, M., and Smith, M. D., "The measured performance of personal computer operating systems," *ACM Trans. Comput. Syst.*, vol. 14, no. 1, pp. 3–40, 1996.

[7] Diefendorff, K., "Compaq chooses SMT for Alpha," *Microprocessor Report*, vol. 13, no. 16, pp. 1–7, 1999.

[8] Endo, Y., Wang, Z., Chen, J. B., and Seltzer, M., "Using latency to evaluate interactive system performance," *SIGOPS Oper. Syst. Rev.*, vol. 30, no. SI, pp. 185–199, 1996.

[9] Flautner, K., Uhlig, R., Reinhardt, S., and Mudge, T., "Thread-level parallelism and interactive performance of desktop applications," in *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pp. 129–138, 2000.

[10] Heinrich, J., *MIPS R4000 Microprocessor User's Manual*, second ed., 1994.

[11] Jacob, B. and Mudge, T., "Software-managed address translation," in *Proceedings of the 3rd IEEE Symposium on High-Performance Computer Architecture*, p. 156, 1997.

[12] Jacob, B. and Mudge, T., "Virtual memory in contemporary microprocessors," *IEEE Micro*, vol. 18, no. 4, pp. 60–75, 1998.

[13] Jacob, B. and Mudge, T., "Virtual memory: Issues of implementation," *IEEE Computer*, vol. 31, no. 6, pp. 33–43, 1998.

[14] Kumar, R. and Tullsen, D. M., "Compiling for instruction cache performance on a multithreaded architecture," in *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pp. 419–429, 2002.

[15] Lee, D. C., Crowley, P., Baer, J.-L., Anderson, T. E., and Bershad, B. N., "Execution Characteristics of Desktop Applications on Windows NT," in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pp. 27–38, 1998.

[16] Lo, J. L., Barroso, L. A., Eggers, S. J., Gharachorloo, K., Levy, H. M., and Parekh, S. S., "An analysis of database workload performance on simultaneous multithreaded processors," in *Proceedings of the 25th annual international symposium on Computer architecture*, pp. 39–50, 1998.

[17] Lo, J. L., Eggers, S. J., Levy, H. M., Parekh, S. S., and Tullsen, D. M., "Tuning compiler optimizations for simultaneous multithreading," in *International Symposium on Microarchitecture*, pp. 114–124, 1997.

[18] Lo, J. L., Emer, J. S., Levy, H. M., Stamm, R. L., and Tullsen, D. M., "Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading," *ACM Transactions on Computer Systems*, vol. 15, no. 3, pp. 322–354, 1997.

[19] Marr, D. T., Binns, F., Hill, D. L., Hinton, G., Koufaty, D. A., Miller, J. A., and Upton, M., "Hyper-Threading Technology Architecture and Microarchitecture," *Intel Technology Journal*, vol. 6, no. 1, pp. 4–15, 2002.

[20] Nemirovsky, M. D., Nemirovsky, A., and Sankar, N., "Prioritized instruction scheduling for multi-streaming processors. Clearwater Networks, Inc.," 1998.

[21] Raasch, S. E. and Reinhardt, S. K., "The Impact of Resource Partitioning on SMT Processors," in *Proceedings of the 2003 International Conference on Parallel Architectures and Compilation Techniques*, pp. 15–25, 2003.

[22] Redstone, J. A., Eggers, S. J., and Levy, H. M., "An analysis of operating system behavior on a simultaneous multithreaded architecture," in *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pp. 245–256, 2000.

[23] Sinharoy, B., "IBM POWER5 Systems," in *Microprocessor Forum*, 2003.

[24] Snavely, A., Tullsen, D. M., and Voelker, G., "Symbiotic jobscheduling with priorities for a simultaneous multithreading processor," in *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, (Marina Del Rey, California), pp. 66–76, 2002.

[25] Tuck, N. and Tullsen, D., "Initial Observation of the Simultaneous Multithreading Pentium 4 Processor," in *Proceedings of the 2003 International Conference on Parallel Architectures and Compilation Techniques*, pp. 26–35, 2003.

[26] Tullsen, D., Eggers, S., and Levy, H., "Simultaneous Multithreading: Maximizing On-Chip Parallelism," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 157–168, 1995.

[27] Tullsen, D. M., Eggers, S. J., Emer, J. S., Levy, H. M., Lo, J. L., and Stamm, R. L., "Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor," in *Proceedings of the 23rd International Symposium on Computer Architecture*, pp. 191–202, 1996.

[28] Vlaovic, S. and Davidson, E. S., "TAXI: Trace Analysis for X86 Interpretation," in *Proceedings of the 2002 IEEE International Conference on Computer Design*, pp. 508–514, 2002.

[29] Vlaovic, S., Davidson, E. S., and Tyson, G. S., "Improving BTB Performance in the Presence of DLLs," in *Proceedings of IEEE/ACM 33nd International Symposium on Microarchitecture*, pp. 77–86, 2000.

[30] Yamamoto, W., Serrano, M. J., Talcott, A. R., Wood, R. C., and Nemirovsky, M., "Performance Estimation of Multistreamed, Superscalar Processors," in *Proceedings of the 27th Annual Hawaii International Conference on System Sciences*, vol. 1, pp. 195–204, 1994.