ADAPTING TO DYNAMIC HETEROGENEITY: VIRTUALIZATION FOR THE MULTICORE ERA

by

Philip M. Wells

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2008

Abstract

As the computing industry enters the multicore era, exponential growth in the number of transistors on a chip continues to present challenges and opportunities to computer architects. This dissertation identifies and addresses one emerging issue in particular: that of *dynamic heterogeneity*, which can arise, even among physically homogeneous cores, from changing reliability, power, or thermal conditions, or different cache and TLB contents. This heterogeneity greatly complicates software's traditional task of assigning computation to cores because the conditions can change more rapidly than software can adapt.

This dissertation begins a push toward hardware taking a more active role in the management of its computation resources. This dissertation proposes hardware techniques to *virtualize* the cores of a multicore processor, allowing hardware to transparently remap any number of the *virtual processors* exposed even to a single operating system to any subset of physical cores. Multicore virtualization operates with minimal overhead, and is shown to enable three novel resource management applications.

In the first, *intermittent faults* are exposed as an emerging reliability challenge for computer systems. These faults, arising from a combination of physical variation and fluctuations in operating conditions, can cause certain cores to become unable to reliably execute for a short period of time. Multicore virtualization can quickly adapt to cores' changing capabilities, resulting in numerous performance and other benefits compared to existing techniques.

Multicore virtualization is then used to improve the scheduling of consolidated servers, allowing the cores of a chip to be *dynamically partitioned* among guest virtual machines. Compared to gang scheduling, dynamic partitioning provides higher throughput, lower transaction latency, and more isolation, yet can quickly adapt to bursts in demand and changing capabilities of the underlying hardware.

Finally, a *Mixed-Mode Multicore* (MMM) is proposed, which allows the simultaneous execution of applications that require high reliability, and those that require high performance. Though conceptually simple, several challenges arise, requiring the use of multicore virtualization and other techniques. The proposed MMM design is shown to improve overall system performance, compared to a traditional DMR system, by approximately 2 times when one extra-reliable and one extra-performance application are concurrently executing.

Acknowledgments

First and foremost, I would like to thank my wife, Corinna, who's love, patience, and encouragement has made these past eight years the best of my life. I am grateful also for little Clara, who's playful and loving spirit, unburdened by thoughts of defenses and dissertations, constantly reminds me of the important things in life.

I am forever indebted to Guri, who has taught me to look at every problem as an opportunity, and take every conference paper rejection with a shrug. I thank the other members of my committee for attending numerous practice talks, and providing excellent feedback for this and other work throughout my studies at Wisconsin. I especially thank Mark and David for organizing several Architecture Beers at the Terrace, where I learned from them a lot more than just architecture.

Koushik has been a friend, office-mate, and collaborator for six years, and I have learned more from him that anyone else at Wisconsin. I am also grateful for the numerous discussions over the years, about work and everything else, with Matthew and Jichaun. Without them, my thoughts would have remained even more disorganized. I wish to acknowledge the other wonderful students at Wisconsin, who taught me, both through feedback and by example, how to read research papers critically, present ideas clearly, and save time for lunch.

Finally, I wish to say, "Thank you," to everyone else who has helped me get to where I am, including my parents, family, friends, and teachers. You know who you are, though you probably won't read this.

Table of Contents

Page

List of Tables

List of Figures

Chapter 1

Introduction

Advances in technology are continuing to drive Moore's Law and double the number of transistors available on a chip every two years. This exponential growth, however, presents several challenges in determining how to use those transistors effectively. In the past, computer architects were able to take these additional transistors and use them for creative purposes, such as out-of-order execution and deep speculation, within a single processing core. Such hardware innovations transparently improved the performance of unmodified applications every few years. However, several issues, including wire delay, power consumption, and design complexity, have come to prevent large, monolithic uniprocessors from continuing to be a viable design option.

Multicore processors are thus rapidly become ubiquitous in the desktop and server markets. Intel, AMD, and Sun are each shipping processors with 4–8 cores [Dorsey et al., 2007; Laudon, 2005; Ramanathan, 2006], and have plans for systems with many more cores in the future [Held et al., 2006]. Instead of trying to create one large, complex core to use all of the available transistors, multicore processors integrate several cores across the chip. In this way, multicore processors allow the performance of certain applications to continue to scale with Moore's Law without requiring frequent, slow, power-hungry cross-chip communication.

Though inevitable, the rapid emergence of multicore processors has presented numerous new challenges to architects, system designers, and application programmers alike. One of the biggest challenges facing the use of multicore processors is that ***improving performance commensurate with Moore's Law is now the responsibility of software***. That is, software now has two new complex tasks: 1) software (potentially including the OS, compiler, and application) must extract enough concurrency to keep all the cores busy, and 2) software must explicitly manage the use of those cores in order to express that concurrency to the hardware.

Although these are both important problems, this dissertation focuses on the second one. In particular, *dynamic heterogeneity* among on-chip cores is identified as a source of both opportunities and challenges for future multicore processors. Attempting to address these opportunities and challenges creates uncertainty in terms of which cores are available or most appropriate to run a given computation at a particular time.

It is presently the responsibility of system and application software to unravel this uncertainty, which greatly complicates software's task of managing the use of those cores. In the spirit of dynamically scheduled out-of-order processors, which remove software's burden of directly managing the functional units of a single core, this dissertation proposes to *virtualize* the multiple on-chip cores. Virtualization allows the hardware designer to abstract the low-level details of the cores, such as their dynamic heterogeneity, in order to alleviate software from the burden of this uncertainty.

This work then examines three applications that use multicore virtualization to improve several aspects of server-class systems. The first applies multicore virtualization to adapt to the effects of an emerging reliability challenge: *intermittent faults*, which can prevent reliable operation of one or more cores for a short period of time (less than 1 second). Virtualization allows the chip to abstract the details of its fault management from the system software, while presenting a view of continuous, fully-functional, reliable operation. The second example uses multicore virtualization to improve the scheduling of consolidated servers, providing better performance, better isolation between guest virtual machines, and lower transaction request latency. The third example, *Mixed-Mode Multicore* reliability, enables a chip to be flexible to the different reliability needs of different applications, transparently running a software thread redundantly on two cores when necessary, and significantly improving performance by running it on only a single core when extra reliability is not necessary.

1.1 Motivation: The Emergence of Dynamic Heterogeneity

Several proposals have exalted the benefits of designing processors with statically heterogeneous cores — cores that are designed to have different physical characteristics in order to capitalize on different engineering trade-offs (e.g., [Gschwind et al., 2005; Kumar et al., 2003, 2004]).

Future multicore chips will similarly contain *dynamically heterogeneous* cores as well — cores which observe different, and rapidly changing, execution characteristics, even though they may be physically homogeneous [Wells et al., 2007b]. These differing characteristics may arise from power and thermal management, intermittent faults, different predictive state, among others sources, and include not only changes in the characteristics of an individual core, but also changes in the *number* of available or appropriate cores. Three categories of dynamic heterogeneity are described below, which create numerous challenges and opportunities for managing the use of the on-chip cores.

Reliability, Power and Thermal Challenges Hardware reliability trends, including reduced noise margins and increased wear-out, process, voltage and thermal (PVT) variations, are conspiring to create circuits that will suffer from a variety of hardware faults, including *transient*, *permanent*, and an emerging class of faults called *intermittent faults* [Borkar, 2004; Borkar et al., 2003; Bowman et al., 2002; Constantinescu, 2003; Semiconductor Industry Association, 2005;

Shivakumar et al., 2002]. This latter class consists of hardware faults which occur in bursts from several cycles to several seconds or more due to physical and operating condition variation [Borkar, 2004; Borkar et al., 2003; Constantinescu, 2007, 2003]. These faults can cause the capabilities of the affected cores to vary during this time, or become temporarily unavailable, creating rapidly changing dynamic heterogeneity (Chapter 5) [Wells et al., 2007a, 2008].

In addition to new reliability challenges, individual cores can also suffer from power density problems, where intense computation raises the temperature of a core more quickly that it can dissipate. Many proposals call for dynamic adjustment of voltage and/or frequency to tolerate this effect [Brooks and Martonosi, 2001; Skadron et al., 2003; Wonyoung et al., 2008], or for temporarily stopping the use of an affected core altogether [Chakraborty et al., 2007; Gunther et al., 2001; Powell et al., 2004]. Global (chip-wide) power dissipation can also be a problem, as multicore processors are already approaching the limits of cost-effective cooling. Such limitations restrict the number of cores that can be simultaneously active to fewer than the number of cores that can be integrated onto the chip, though which cores are currently active can vary dynamically and rapidly [Chakraborty et al., 2007]. These mechanisms to tolerate different power and thermal issues also create dynamic heterogeneity.

Opportunities for Dynamic Core Coupling In addition to cores' varying capabilities due to physical constraints, the software executing on a core can dynamically change its requirements. For example, financial applications may require Dual-Modular Redundancy (DMR) (e.g., [Bernick et al., 2005; Smolens et al., 2006]) to maintain sufficient levels of reliability in future systems, while media or web applications can tolerate higher rates of hardware faults in exchange for better performance and lower power consumption. Executing both types of applications on one machine, by allowing cores to dynamically switch between operating together as one logical DMR pair, or operating independently to run two separate threads, offers a way to provide this differentiated level of service (Chapter 7).

Similar to the changing reliability requirements, software also has changing levels of parallelism. Sometimes, task- or instruction-level parallelism of a single thread can be exploited via multiple conjoined cores through speculative multithreading (e.g., [Sohi et al., 1995]) or dynamic *Core Fusion* (e.g. [Ípek et al., 2007]). At other times, those cores can be reconfigured to take advantage of data-level parallelism [Kyo et al., 2007; Sankaralingam et al., 2003], or used to exploit thread-level parallelism by independently executing multiple threads.

The dynamic coupling of cores, for reliability or the extraction of parallelism, not only creates different capabilities of each *logical* core, but also a varying *number* of logical cores, capable of concurrently executing a varying number of software threads.

Opportunities for Enhancing Cache Locality The third category of dynamic heterogeneity examples involves manging the use of per-core predictive structures, such as caches, branch predictors, and TLBs. As noted by Torrellas et al. [1995] in a different context, heterogeneous capabilities of each core arise through differences in the contents of these structures create. Such dynamic heterogeneity can be actively created and then exploited through careful assignment of computation to cores.

In a consolidated server, for example, multiple guest virtual machines (VMs) sometime share the physical cores in a system due to the inflexibility of the system and application software running in each VM [Uhlig et al., 2004]. The cache, branch predictor, and TLB footprints of these distinct VMs are largely or completely independent, leading to frequent thrashing in these structures when context switches are frequent. A mechanism to flexibly map each VM's *virtual processors* to the physical cores, however, provides the opportunity to share a core among virtual processors from the *same* VM. Such dynamic re-mapping specializes each core to execute a certain kind of computation specific to a particular VM, improving performance and isolation between VMs (Chapter 6).

Similarly, sharing a core among computation from multiple threads (from a single VM) can create further opportunities to enhance cache locality [Chakraborty et al., 2006]. The key with this so called *Computation Spreading* is to carefully map similar computation fragments from different threads onto one core, while spreading dissimilar fragments from the same thread onto other cores.

Computation Spreading and flexible consolidated server scheduling not only take advantage of the dynamically heterogeneous capabilities of different cores, but actively *create* this heterogeneity through the fine-grained assignment of computation to cores to further improve execution efficiency.

1.1.1 Rethinking the Roles of Hardware and System Software

These emerging opportunities and challenges of dynamic heterogeneity share two distinct traits. First, the details of each example creates a new layer of complexity between the computation and the physical hardware performing that computation. This complexity in turn creates uncertainty in how computation should be efficiently mapped onto the hardware at any given moment. Unraveling this uncertainty requires detailed knowledge of the current configuration and capabilities of each core — information that modern system and application software does not posses, and cannot easily acquire due to the nature of the relatively static interfaces present in layered systems design.

Second, the capabilities and configurations of the hardware can change very rapidly. Yet the cost of trapping into the operating system to perform and implement a scheduling decision has actually increased in the past several decades relative to the cost of computation [Nellans et al., 2005]. As a result, system and application software often cannot implement policies to address dynamic heterogeneity with sufficient timeliness, even if the hardware/software interface *was* modified often enough to provide appropriate information.

Given these two issues, this dissertation argues that hardware must take a more active role in the management of its resources, in particular, the on-chip cores. This management is made possible by *abstracting* the details of the on-chip cores using a thin virtualization layer implemented in hardware and firmware. Such virtualization and abstraction allows simple homogeneous cores to be exposed to the operating system via the hardware/software interface (i.e., ISA), while innovations in multicore hardware adapt to the opportunities and challenges of dynamic heterogeneity.

1.2 Thesis Statement

The emergence of dynamic heterogeneity and observations of software's limited ability to adapt to it have led to the following thesis statement:

> *Future multicore processors should support software-transparent virtualization of on-chip cores due to the emerging challenges and opportunities of dynamic heterogeneity.*

To support this statement, this dissertation investigates two examples of dynamic heterogeneity, intermittent faults (Chapter 5) and mixed-mode reliability (Chapter 7), which the system software is unable to adequately manage. In addition, a third example is examined where transparent virtualization allows unmodified system software to take advantage of the dynamic capabilities of hardware (Chapter 6).

1.3 Contributions

Within the context of dynamic heterogeneity, this dissertation makes several contributions. First, it proposes a mechanism for *virtualizing* and *abstracting* on-chip cores, and then uses these techniques to improve several aspects of server-class systems. The end result is that nearly-unmodified software can seamlessly adapt to several of the opportunities and challenges of dynamic heterogeneity.

Multicore Virtualization A layer of virtualization, and techniques to support it, are proposed in Chapter 4. This layer allows the chip manufacturer, through a combination of hardware and firmware, to dynamically remap computation from one core to another more appropriate core, and to seamlessly suspend computation on one or more cores when necessary. The unique contribution of this chapter is to allow more *virtual processors* to be exposed to the system software (either a software hypervisor or a single operating system) than there are physical cores available to execute those virtual processors. Such a system is known as an *overcommitted* system.

Adapting to Intermittent Faults The work in Chapter 5 brings the inability of current system software to tolerate the dynamic heterogeneity caused by intermittent faults to the attention of architects and system designers. This chapter further demonstrates that the proposed multicore virtualization techniques, the ability to operate as an overcommitted system in particular, allows easy adaptation to the effects of these faults, while presenting a view of continuous, fully-functional operation to the system software.

Improving Consolidated Servers Although the benefits of abstracting the details of hardware reliability problems from the system software may be apparent, multicore virtualization techniques can also be used to improve performance. As one demonstration of this fact, Chapter 6 uses the increased scheduling flexibility provided by the ability to overcommit a single guest virtual (VM) machine to improve the performance and reduce the transaction latency of consolidated servers. Isolation between guest VMs is also improved, while still allowing VMs to consume a varying fraction of the physical resources to handle bursts in demand.

Mixed-mode Reliability Chapter 7 proposes *Mixed-Mode Multicore Reliability* to allow applications that require extra reliability, as well as applications that do not, to simultaneously execute on one machine. The challenges of such a system are addressed by 1) replicating a small fraction of the less-reliable application's activities (TLB translations of L1 cache misses) to isolate the state of critical applications, 2) operating as an overcommitted system to handle the varying number of available physical cores, and 3) using virtualization to allow hardware to handle the details while software specifies its requirements via a simple interface.

1.4 Dissertation Outline

Chapter 2 provides background information on virtualization and emerging trends in hardware reliability. Details of the experimental methodology are provided in Chapter 3. Chapter 4 proposes techniques for virtualizing the cores of a multicore processor, focusing on the unique challenges facing an overcommitted system. Chapter 5 discusses intermittent hardware faults, and compares several techniques for adapting to the effects of these faults. Chapter 6 demonstrates how the multicore virtualization techniques can improve consolidated server scheduling. Chapter 7 examines the challenges, and proposes solutions, to the simultaneous execution of both DMR and non-DMR applications on the same machine. Chapter 8 summarizes the contributions and insights of this dissertation, discusses how multicore virtualization has and can impact other research, and revisits the thesis statement.

Chapter 2

Background: Virtualization and Reliability

Trends in hardware reliability provide a powerful qualitative and quantitative motivation to abstract the details of hardware fault tolerance from the software. For this reason, two major components of this dissertation (Chapters 5 and 7) identify and address emerging reliability issues.

Virtualization provides the means to achieve this abstraction. This dissertation proposes novel mechanisms for virtualizing the cores of a multicore processor, which are described in detail in Chapter 4. This chapter provides background information on virtualization and reliability.

2.1 Processor Virtualization

Virtualization is a generic term used to describe the translation of one interface into another interface of equivalent complexity [Popek and Goldberg, 1974; Smith and Nair, 2005]. Virtualization has been used for many decades by companies such as IBM to maintain backward binary compatibility with legacy application and system software and new hardware running new ISAs [Popek and Goldberg, 1974]. The original software is said to be running in a guest *virtual machine* (VM), which appears to the software to be the equivalent of the original hardware. One or more VMs may be running on a single physical machine (the host) under the control of a *virtual machine monitor* (VMM). Virtualization has been used more recently by many companies to enable the *consolidation* of multiple VMs onto a single machine (e.g., VMware [Waldspurger, 2002]).

Abstraction is a term used to denote the translation of an interface into one of lesser complexity, in effect, hiding less relevant details of an underlying implementation [Smith and Nair, 2005]. Layered computer systems liberally use abstraction between layers in order to better understand the issues within a given layer. Virtualization and abstraction are often related. For example, this dissertation proposes to virtualize the cores of a multicore processor in order to abstract the complexity of their emerging dynamic heterogeneity.

Virtualization of an entire system, especially to run multiple VMs on a single machine, involves many complex tasks, including virtualization of memory, I/O, and processors. This primary focus of this dissertation is *processor virtualization* (PV) for multicores.

PV creates a level of indirection between the virtual processors (or VCPUs) that are exposed to the operating system via the hardware/software interface (i.e., ISA), and the physical processors (i.e., physical cores of a multicore system) actually implemented in hardware. PV refers to the management and sharing of these physical core resources among the VCPUs [Mullender et al., 1994]. In particular, PV answers the question: How and when is each VCPU allowed to use which physical core? PV involves both the low-level mechanisms for correctly multiplexing the architected processor state (e.g., registers and software-visible TLBs) among multiple VCPUs, as well as the high-level policies governing their use.

PV can be performed in a manner entirely transparent to the guest OS (i.e., *pure virtualization*, as in VMware [Waldspurger, 2002]), or in cooperation with the guest OS (i.e., *para virtualization*, as in Xen [Barham et al., 2003], Denali [Whitaker et al., 2002], and the IBM Power5 Hypervisor [Armstrong et al., 2005]). Both techniques have several advantages and disadvantages. Primarily, pure virtualization can support unmodified, legacy software, but can suffer performance loss when the software makes assumptions about the hardware that are invalid for virtualized resources [Barham et al., 2003; Wells et al., 2006]. Para virtualization eliminates these assumptions, but

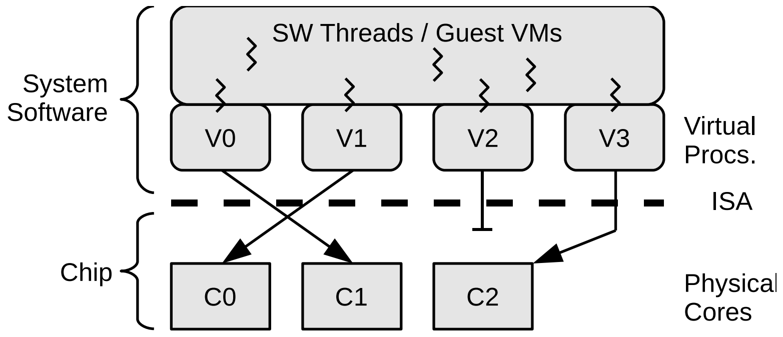


Figure 2.1 Multicore Virtualization. Four VCPUs are exposed the the system software via the ISA, while only three physical cores are actually present. VCPUs V0, V1, and V3 have been transparently migrated to cores C1, C0, and C2 respectively, while VCPU V2 has been transparently suspended.

sometimes requires significant modification to the guest OSs and the hardware/software interface in order to do so.

In the context of this dissertation, pure PV is used to provide software-transparent migration of a VCPU from one core to another, and to provide support for *suspending* the execution of one or more VCPUs, so that the number of VCPUs exposed to the system software can be more, less, or the same as the number of physical cores. This *multicore virtualization* is performed below the ISA, so that the lowest level of system software (e.g., the operating system (OS) or traditional software hypervisor such as VMware) can remain completely unmodified. Multicore virtualization is depicted in Figure 2.1, which shows transparent migration of VCPUs V0, V1, and V3 to cores C1, C0, and C2 respectively, and the suspension of VCPU V2. Such multicore virtualization allows hardware innovation to address dynamic heterogeneity while abstracting its details and complexity from the system and application software.

2.1.1 Server Consolidation

Server consolidation is a term used to denote the process of moving two or more services, such as web or email hosting, from multiple, separate machines onto one physical machine [Armstrong et al., 2005; Figueiredo et al., 2005; Sun Microsystems, 1999; VMware, 2006b; Waldspurger, 2002]. Consolidation aims to simplify the operation and reduce the cost of an organization's multiple servers (or of servers from multiple organizations). The installation of the original services, including the system and application software, as well as the configuration, are kept intact as much as possible, and simply copied to the new machine. A *system VMM* is employed to virtualize the entire system, and preserve the illusion that each service, running in its own guest VM, is the only service running on the machine. This is possible because, in the average case, each VM only utilizes a small fraction of the physical machine simultaneously. Consolidated server workloads are used to help motivate and evaluate the proposals in Chapters 6 and 7.

2.2 Trends in Hardware Reliability

The components of future multicore processors will become less reliable as technology scales, because individual devices are increasingly susceptible to a variety of hardware faults caused by a variety of factors, including high-energy particle strikes, manufacturing process variation, device wear-out, and temperature and voltage fluctuations [Borkar, 2004; Borkar et al., 2003; Bowman et al., 2002; Constantinescu, 2003; Semiconductor Industry Association, 2005; Shivakumar et al., 2002].

Faults may manifest as *transient* faults, which can affect a single transistor or wire for less than one cycle, causing a single bit flip. Transient faults are also called *single-event upsets* [Normand, 1996], or *soft* errors [Mukherjee et al., 2005; Shivakumar et al., 2002]. Numerous academic and industry proposals have proposed software, architectural, microarchitectural, and circuit techniques for tolerating transient faults [Aggarwal et al., 2007; Austin, 1999; Gomaa et al., 2003; LaFrieda et al., 2007; Mitra et al., 2006a,b, 2005; Mukherjee et al., 2002, 2005; Reinhardt and Mukherjee, 2000; Reis et al., 2005a,b; Rotenberg, 1999; Smolens, 2008; Smolens et al., 2004, 2006; Weaver and Austin, 2001; Yeh, 1996; Zhang et al., 2006; Zhou, 2006]. Many of these proposals rely on the property that transient faults are independent and temporary, i.e., they may affect one device in one cycle, but will not affect the same, or other, devices during the window of time necessary to detect and recover from the fault.

Faults may also manifest as *permanent* faults, either due to a manufacturing defect, or after irreversible wear-out damage has occurred. Several proposals have targeted these types of faults as well [Aggarwal et al., 2007; Austin, 1999; Bower et al., 2005; Govil et al., 2000; LaFrieda et al., 2007; Schlichting and Schneider, 1983; Shyam et al., 2006; Smolens et al., 2007; Weaver and Austin, 2001; Yeh, 1996]. Many of these proposals rely on the property that permanent faults will consistently affect the same devices in the same manner.

There is a third emerging class of faults called *intermittent* faults, however, which do not observe the properties of either transient or permanent faults [Borkar, 2004; Borkar et al., 2003; Constantinescu, 2007, 2003]. Intermittent faults can occur frequently and irregularly for several cycles to several seconds or more, and then disappear for a period of time. Because of this, intermittent faults present further challenges for designers of reliable systems. These faults commonly arise due to physical variation (e.g., process variation or in-progress wear-out), combined with variation in the operating conditions (e.g., voltage and temperature fluctuations) [Borkar, 2004; Borkar et al., 2003; Constantinescu, 2007, 2003]. Intermittent faults are discussed in more detail in Chapter 5.

2.2.1 Fault Frequency

Due to the degree with which chip manufacturers guard any information concerning hardware failures, it is difficult to ascertain the fault rates of current, let alone future, chips. Nonetheless, gross, order of magnitude (OOM) estimates can be made from published data.

For example, considering only transients induced by high-energy particles, IBM's Power4 systems, shipping in 2002, target a Mean Time To Failure (MTTF) from transient errors of 10 years [Bossen, 2002, as quoted by Mukherjee et al., 2005]. The MTTF of each processor chip in a multiprocessor system must thus be at least one OOM longer, and possibly higher considering other components in the system. Although low-cost, commodity chip manufacturers are unwilling to disclose their target MTTF, their chips are likely to have a MTTF several (e.g., 2–3) OOM shorter than IBM's server products. In addition, the MTTF of combinational logic is decreasing by approximately one OOM every two years due to the smaller amounts of charge these high-energy particles need to impart to cause an error [Shivakumar et al., 2002]. Incorporating multiple low-overhead circuit-level techniques (e.g. [Blough et al., 1999; Ernst et al., 2003; Hamilton and Orailoglu, 1998; Ismaeel and Bhatnagar, 1997; Shyam et al., 2006; Smolens et al., 2007]) may increase the MTTF by 2–3 OOM, but these techniques aren't fully composable, as they often target the same components and same causes or errors. It is thus plausible that the MTTF of commodity processors from transient errors in 2012, as visible to the microarchitecture, will be 4–6 OOM shorter than IBM's 10 years, or on the order of one failure every 5 minutes to 10 hours.

Though the details remain secret, permanent faults are already significant in commodity chips. For example, results from industry indicate, based on confidential data, that approximately 10% of processors experience a permanent fault of some kind (not necessarily debilitating) within 3 months and 25% within 1 year [Aggarwal et al., 2007]. These results are said to already include numerous architectural enhancements to reduce the rate of errors. These faults are expected to further increase in frequency in future technologies; a one OOM increase in rates results in 95% of chips experiencing a permanent fault within one year [Aggarwal et al., 2007].

Such frequent errors can manifest to, or be masked by, software in a variety of ways [Li et al., 2008]. Nonetheless, many applications, and users of those applications, cannot tolerate these high fault rates. Or alternatively, many customers cannot tolerate the financial or other risks associated with such rates. Some level of microarchitectural detection, recovery, and adaptation to the effects of these faults thus appears to be necessary for future generations of multicores.

2.2.2 Detecting Faults

Microarchitectural reliability techniques have shown great promise at tolerating faults, i.e., at hiding the effects of faults that escape circuit reliability techniques from the system- and user-level software running on the hardware. These techniques all use some form of spatial or temporal redundancy to detect, and/or recover from faults in the underlying circuits.

Several microarchitectural techniques purport to be low-overhead [Bower et al., 2005; Shyam et al., 2006; Smolens et al., 2007]. While this claim may be true, these techniques also only increase the MTTF from one specific type of fault (e.g., permanent faults) by approximately one OOM. Similar to the circuit techniques, these low-overhead microarchitectural techniques are not always composable either. As devices become more unreliable, the ways in which faults manifest increase, with a consequential increase in the complexity and overhead of the techniques to tolerate them. Thus, such low-overhead techniques can likely help maintain current rates of faults for one or two generations of processors, but not longer.

Several high-overhead reliability schemes such as those employing Dual Modular Redundancy (DMR), have become popular for situations where high-coverage detection and/or recovery is required. These DMR proposals provide spatial redundancy by joining two physical cores, each executing a copy of the program, into one logical processor visible to the system software. Such DMR schemes have been proposed and investigated by a multitude of academic and industry researchers [Gomaa et al., 2003; LaFrieda et al., 2007; Mukherjee et al., 2002; Reinhardt and Mukherjee, 2000; Rotenberg, 1999; Smolens et al., 2006; Weaver and Austin, 2001; Zhou, 2006], and have been implemented to varying degrees in real products [Bernick et al., 2005; McEvoy, 1981; Slegel et al., 1999]. Most academic proposals focus on single-event transient faults, but many can be adapted to provide very high fault coverage for intermittent and permanent faults as well (e.g., [Aggarwal et al., 2007; Austin, 1999; LaFrieda et al., 2007; Smolens et al., 2006;

Weaver and Austin, 2001; Yeh, 1996]), making them suitable for the variety of factors producing faults in future multicores.

Assuming that faults manifest independently on physically separate cores, and that only one fault occurs at a time, DMR can provide perfect detection of faults. Even after relaxing these assumptions, DMR can still provide many OOM reduction in fault rates. The benefits of dual redundancy, however, come at a large overhead (approximately 100%) in terms of power and throughput, and can also significantly impact the IPC of a single thread due to frequent synchronization. An open question is whether the cost/benefit tradeoff of using DMR will allow circuit and process researchers to use more aggressive designs with higher fault rates, but lower power, faster clocks, and higher yields.

This dissertation takes the position that DMR will be useful for a growing number of applications with every technology generation, yet many applications will remain sufficiently reliable for the next decade while using only lower-overhead microarchitectural and circuit techniques.

2.2.3 Fault Recovery

Detecting the occurrence of a fault is one important aspect to system reliability. A second, related concern is recovering from the effects of the fault before it can corrupt the state of the software running on the faulty hardware.

Proposed fault recovery schemes operate at many different levels in the hardware/software stack, but nearly all of them delay the writing of a value to “safe” storage until fault-free execution can be verified. If a fault is detected, the value is not committed, and a mechanism (such as rolling back to the previously verified safe state) is invoked to ensure consistency with other values in the system.

Circuit techniques typically prevent a value from being written to a pipeline latch before it is verified fault-free, or use a second, delayed latch for recovery before writing a value to the register file (e.g., [Ernst et al., 2003]).

Microarchitectural reliability techniques generally employ one of three techniques for fault recovery: 1) store unverified results in the pipeline, and using the physical register file for safe storage [Gomaa et al., 2003; Rotenberg, 1999; Smolens et al., 2006; Weaver and Austin, 2001; Zhou, 2006], 2) store unverified results in the cache and/or register file without making them visible to other cores, and use the cache as safe storage [LaFrieda et al., 2007], or 3) store unverified results in the cache hierarchy after making them visible to other cores, and use a double buffering checkpoint mechanism in the memory hierarchy to create a consistent, logically global safe checkpoint, from which all cores can simultaneously recover when a fault is detected [Nakano et al., 2006; Sorin et al., 2002].

Software reliability techniques rely on software's ability to buffer and roll back to a consistent checkpoint.

A large body of prior work has postulated about the pending increase in fault frequency as technology continues to scale. This dissertation does not make significant contributions in terms of the actual mechanisms for detecting or recovering from hardware faults. Instead, it addresses the need to *adapt* to the dynamic heterogeneity caused by intermittent faults (Chapter 5), and to remain flexible to the dynamic heterogeneity caused by the changing reliability needs of software (Chapter 7).

Chapter 3

Experimental Methodology

Experimental results in this dissertation are derived from a full-system, execution driven simulator, which models a single-chip multicore processor with 8 cores. Workloads consist of a diverse set of commercial server, and other applications running Solaris 9 and Linux 2.6 operating systems on the SPARC V9 platform.

This chapter presents the common details of the simulator, target multicore system, and workloads, as well as the methodology used in most experiments. Individual chapters include a detailed explanation of the specific experiments in that chapter and any methodological differences.

3.1 Simulator Implementation

This simulation infrastructure consists of a *functional* component, based on Simics 2.0.28 [Magnusson et al., 2002], as well as an 80k line C++, cycle-level architectural *timing* component developed within Dr. Sohi's research group over the past five years, called *ms2sim*.

The functional component provides the ability to execute unmodified operating systems and commercial workloads running on the SPARC V9 platform. It models the logical effects of instruction execution, devices, interrupts, etc., of a Sun Enterprise 8000 server with UltraSPARC IIICu processors for Solaris workloads, or UltraSPARC II processors for Linux workloads.[1] These two processors are functionally very similar. The primary difference that can affect the results in

[1]Linux has not been properly ported to the UltraSPARC IIICu-based Enterprise servers.

this dissertation is the configuration of the TLBs. UltraSPARC II uses a 64-entry fully associative TLB for both data and instruction. UltraSPARC IIICu uses a 16-entry fully associative plus a 128-entry 2-way associative instruction TLB and a 16-entry fully associative plus a 1024-entry 2-way associative data TLB, and results in dramatically lower miss rates for some workloads. In order to keep the behavior of these two processors as similar as possible, the UltraSPARC II TLB was modified to use the same configuration as the UltraSPARC IIICu, but maintain the functionality of the UltraSPARC II TLB. Although source code for the SPARC TLBs was provided by Simics, and modified extensively during the course of this research, source code for most other portions of the functional simulator is not available.

The timing component models the execution characteristics of a single multicore processor, including out-of-order cores and cache hierarchy. It uses the Simics Micro-Architectural Interface (MAI) to tie into the functional simulator. MAI allows the timing simulator to dictate the exact cycle in which the functional simulator should proceed each instruction through which pipe stage. Within each core, *ms2sim* models the register file, functional units, instruction window, load/store queue, store buffer if used, and control flow. It also handles all aspects of the memory hierarchy timing, including cache arrays, banking and bandwidth, network, coherence protocols, and maintaining the appropriate consistency model. The memory hierarchy is not responsible for manipulating data. For functional correctness, *ms2sim* must properly handle control flow, load-store dependencies within a core, the store buffer, and maintain multiprocessor consistency. Simics MAI determines register dependencies among instructions and provides this information to the timing simulator.

Simics MAI allows more fidelity in multiprocessor simulations compared to simulators which use dynamically (or statically) generated traces, since the functional simulator observes loads receiving the proper architectural value at the appropriate time dictated by the timing simulator. Compared to timing-first simulators (e.g., [Mauer et al., 2002]), MAI does not require the timing simulator to duplicate the efforts of the functional simulator. In addition, this interface allows slightly more fidelity compared to timing-first simulation in multiprocessor simulations. In certain cases, the timing-first simulator and the functional, shadow simulator can provide different, though both valid, sequentially consistent memory orderings. In these cases, the timing simulator assumes an error occurred in its execution, and restarts. Simics MAI also allows correct modeling of non-sequentially consistent (SC) memory models, though SC is used for all results in this dissertation.

The simulation infrastructure also allows for the use of a simplified processor model to speed up simulations, though it uses the same memory hierarchy. Simics models the simplified timing aspects of the cores, which represent a single-instruction, blocking core. This execution model provides approximately a 10X improvement in simulator speed, allowing the examination of longer periods of simulated execution.

The timing of other components of the system, such as disks and network interface cards, are functionally modeled, but not modeled by the timing simulator.

Virtualization Implementation In general, the Simics functional simulator correctly handles the execution of the VCPUs, while the timing component dictates when and with what timing characteristics those VCPUs execute. However, the various components of multicore virtualization interact with the functional and timing components of the simulator in different ways, and are described in Section 4.4.1 after discussing the proposed virtualization techniques.

Consolidated Server Implementation Studies with consolidated workloads are performed in Chapter 6. These studies assumes the use of a software VMM, similar to VMware ESX Server, which virtualizes I/O, memory, and the execution of privileged instructions. Without access to such a VMM that supports the SPARC Enterprise 8000 platform used in this dissertation, the

Fetch, issue, commit	4 instructions / cycle
Fetch buffer	2-entry
Integer pipeline	8 stages
I-Window & ROB	128 entries, OOO issue
Load and store queues	32 entries each, w/ bypassing
YAGS branch predictor	8k-entry choice, 2k-entry except tables [Eden and Mudge, 1998]
Cascaded indirect pred	256-entry filter, 1k-entry except tables [Driesen and Hölzle, 1998]
RAS	32-entry table, modified for commercial workloads (Section 3.2.1)
Private L1 instr cache	16KB, 2-way, 1-cycle, coherent
Private L1 data cache	16KB, 2-way, 1-cycle, write-back
Private L2 unified cache	512KB, 4-way assoc, 15 cycle ld-to-use, 4 banks, 4-stage, inclusive
On-chip shared L3 cache	8MB, 16-way, 55-cycle load to use, 8 banks, pipelined, exclusive
On-chip interconnect	Logical crossbar, 10-cycle latency
Main Memory	365 cycle load-to-use, 24GB/sec

Table 3.1 Target Multicore Parameters

execution of consolidated workloads is emulated without modeling the overhead of this software VMM. The timing simulator models the VMM, which is responsible for sharing the physical resource of the target machine among guest VMs. By articulating which VCPUs from which VM run on which cores at what time, the timing simulator causes VMs to dynamically share the cores, caches, and TLBs. The methodology for consolidated servers is discussed in more detail in Section 6.3.1.

3.2 Target Multicore System Configuration

Most experiments in this dissertation model a single-chip multicore processor with eight cores. Experiments in Chapter 4 cannot use a larger number of cores because experiments are limited by the maximum of 24 OS-visible VCPUs that can be installed in the simulated SunFire system. To facilitate using the same workloads in all chapters, Chapter 5 also uses an 8-core processor, and experiments in Chapter 7 use a 16-core processor with dual redundancy.

In most experiments, each core is modeled as a 4-wide, dynamically scheduled, out-of-order core, with a 128-entry instruction window, executing at 3GHz. Within each core, *ms2sim* models the register file, functional units, instruction window, load/store queue, control flow, and other aspects of a dynamically scheduled pipeline. The configuration of the processing core is shown in top of Table 3.1.

The processor implements a fetch buffer, containing two 64-byte cache lines worth of instructions, which can be fetched without accessing the cache. A hit in the fetch buffer prefetches the next cache line.

The load-store queue performs conservative disambiguation. Loads wait until older stores addresses are known before executing. Loads that cannot execute due to an older store with an unknown address are prefetched. Stores prefetch write permission once both their address and data are known, but do not write to the cache until they commit.

In order to run simulations for a longer simulated time, a second, in-order configuration, was used for certain experiments. This second configuration uses a simple model which executes one instruction at a time, with each instruction taking one cycle in the absence of a memory stall. Memory stalls block the processor. Experiments using this in-order model are noted in the corresponding chapter.

The on-chip cache hierarchy includes private L1 instruction and data caches, which are each 16KBytes and 2-way set-associative. The L1 data cache is write back. The L1 instruction cache is kept coherent with the data cache and other cores. Located with each core is a 512KByte private, unified L2 cache. L2s maintain coherence via a MOESI directory protocol. An on-chip directory is co-located with an 8-way banked, shared, on-chip level 3 cache, which maintains exclusion with the private L2s (i.e., acts like a large victim cache). L2s and L3 banks are connected via a logical crossbar with a 10-cycle point-to-point latency, giving the L3 a 55-cycle load-to use. Off-chip memory observes a 365-cycle load-to use, and is limited to a bandwidth of 24GBytes/sec. The line size for all caches is 64 bytes. Caches do not perform prefetching.

The use of a 3-level on-chip hierarchy has not been common in academic research outside of papers published in cooperation with this work [Chakraborty et al., 2006; Wells et al., 2006, 2008]. This may be due in part to the implementation complexity, or to a "follow the crowd" mentality. Several years ago, private L2 caches and a shared, exclusive L3 cache seemed a good compromise in the ongoing debate between 2-level multicore designs centered around better locality from private L2 accesses and better capacity from a shared L2 cache. IBM's Power5 used a similar hierarchy to the 3-level used in this work, though the Power5's exclusive L3 sits off-chip [Kalla et al., 2004]. More recently, this choice of design has been vindicated by the quad-core AMD Opteron processors, which feature 512KByte private L2s, and a 2MByte L3 victim cache [Conway and Hughes, 2007].

3.2.1 Return Address Stack

A conventional return address stack (RAS) consists of a FIFO of program counter (PC) addresses. When a program makes a function call, the PC following the call is pushed onto the RAS, and later popped from the RAS (and used as a branch target prediction) when a return is made from the function. Modern RASs include support for multiple head pointers and other mechanisms to repair a stack against addresses that were pushed or popped for speculative, wrong-path instructions (e.g., [Jourdan et al., 2005; Skadron et al., 1998]).

With OS-intensive commercial workloads, however, hardware speculation is only once source of problems with the RAS. Another arises, especially in Solaris code, when a function makes a `tail call` (calls one function at the end of another). In this case, the compiler optimizes the return path, so that the innermost function returns directly to its caller's caller. Because SPARC only includes jump and call instruction which *link*, e.g., save the return address to a register, the return address of the middle function is placed onto the RAS, though that PC is never executed.

A typical method for repairing the stack after such a call is to search the rest of the RAS for the correct return address on a RAS misprediction, and pop off any entries between the current top and the correct return address. After one misprediction, the hope is that this method allows the RAS to continue correctly. However, one misprediction on the compiler "optimized" call will still occur.

To remedy this situation, a small filter table was added to the RAS to predict which PCs should not be pushed onto the RAS during a call. On a misprediction, if the correct return PCs is found in the top four entries of the RAS, all intervening PCs are added to the filter table. The next time one of those PCs appears as the return address of a function call, it is not added to the RAS.

To the best of my knowledge, predicting *not* to push entries onto the RAS is a small, but novel contribution. However, this optimization has not been studied in sufficient detail to determine whether other architectures and operating systems are affected by this problem as well.

3.3 Workloads

Several different workloads were used for experiments in this dissertation. Most experiments use several commercial workloads running on Solaris 9. Experiments in Chapter 4 were also performed with several workloads running on Linux 2.6.10. The individual workloads are briefly described below. Not every experiment was run with all workloads.

Apache The Surge client [Barford and Crovella, 1998] is used to drive the open-source Apache web server, version 2.0.48. To reduce OS idle time, the Surge client was configured with no user think time.

pmake Parallel compile using GNU make and the Sun Forte Developer 7 C compiler (on Solaris) or gcc-3.3.4 (on Linux). Serial linking phases are avoided.

Zeus The Surge client was used to drive the commercial Zeus web server, configured similarly to the Apache web server.

OLTP OLTP uses the IBM DB2 database to run queries from TPC-C. The database is scaled down from TPC-C specification to about 800MB and runs 192 concurrent user threads with no think time.

pgoltp pgoltp uses the PostgreSQL database version 8.1.3 [PostgreSQL Global Development Group] to run TPC-C-like queries from the OSDL dbt2 test suite [Open Source Development Labs]. The database is scaled down from TPC-C specification to about 800MB and runs 200 concurrent user threads with no think time.

pgbench Pgbench runs TPC-B like queries on the PostgreSQL database [PostgreSQL Global Development Group]. Pgbench is packaged with the PostgreSQL source distribution.

Spec2000Mix This benchmark is used in Chapter 4 for experiments with 24 VCPUs. All SPEC2000 benchmarks are concurrently run, except the FORTRAN90 FP benchmarks for which no open source Linux compiler exists. Instead, two copies of gcc, mcf, and art are run to keep all 24 VCPUs busy. Benchmarks have been warmed-up for one billion cycles.

vortexMIX This benchmark is a simple multiprogramming workload consisting of 8 copies or *255.vortex* from SpecINT2000 running reference inputs, and has been warmed-up for one billion cycles. This benchmark is used in Chapter 5.

artOMP This benchmark is 330.art_m from the SpecOMP2001 suite, using reference inputs, and warmed up and running in steady-state. This benchmark is used in Chapter 5.

barnes This benchmark is from the Splash2 suite is also used in Chapter 5. This benchmark is setup with only 512 bodies in order to increase the impact of synchronization. It uses `pthreads_cond_wait()` for barrier synchronization

barnes/U This is a variant of barnes, which implements barrier synchronization with user spin locks.

All workloads are run for several simulated seconds, and sometimes minutes, using just the Simics functional simulator to warm up the workload and OS disk cache before a checkpoint is taken that is used by our timing simulations.

Although the timing simulator models a 3GHz processor, the OSs are configured for a 1GHz processor. This difference can impact the frequency of OS timer interrupts, slightly increasing OS scheduling overhead. Disk latencies are set arbitrarily low (1-10μs) in order to reduce the amount of time that the CPUs spend idle.

Some workloads in Chapter 4 use 24 VCPUs, and were warmed up on a simulated system with 24 processors. Other workloads use 8 VCPUs and are warmed up an an 8 processor system. This chapter uses both Linux and Solaris workloads. `Apache` and `pmake` are run on both OSs. Spec200Mix is only run on Linux, and the rest only on Solaris.

Some experiments in Chapter 5 use Solaris's Dynamic Reconfiguration [Sun Microsystems, Inc.] capabilities. For these workloads, the command `cfgadm`, which prints out the current system slot configuration, was run prior to warming up the workload in order to load the necessary kernel modules so that the appropriate kernel functions could be called directly by the simulator without the overhead of the command. The technique for doing so is described in more detail in Section 5.4.1.1.

Consolidated server workloads used in Chapter 6 were combined from the 8-processor workloads used elsewhere.

Microbenchmarks are used in Chapters 5 and 6, and are described in the appropriate chapter.

3.4 Methodology

Due to inherent variability in these workloads as described by Alameldeen and Wood [2003] (primarily from interrupt processing and OS scheduling decisions), a small (up to 10-cycle) random variation is added to the latency of each main memory access. Several trials of each benchmark are then run per experiment. Results are averaged over these trials. The 95% confidence interval is included on most performance graphs.

Experiments in Section 4.4.2 and Chapter 5 were started with warmed cache checkpoints. Experiments in Sections 4.4.3–4.5.3.1 and Chapter 6 were not, but were run at least long enough to have L3 misses ten times in excess of L3 cache lines (in practice, they were run much longer).

Experiments in Chapter 4 are each run for a specified number of workload transactions. Performance is then determined using the total number of cycles taken. Unfortunately, this method breaks down for shorter runs, especially with workloads such as `pmake`, `OLTP`, and `pgoltp`, which contain long running, variable-length transactions.

To remedy this situation, experiments in other chapters are run for a specified number of cycles or a specified number of *user* instructions. Performance is then determined by examining *User IPC*: the number of committed user instructions across all VCPUs divided by the *total* number of cycles from both user and OS. Data in this thesis (e.g., Figure 4.4) corroborates the results of others (e.g., [Wenisch et al., 2005]) which conclude that user commits correlates very closely to workload transactions for workloads without a significantly user spin component.

Chapter 4

Multicore Virtualization

Past generations of uniprocessors enabled software-transparent hardware innovation, such as dynamic branch prediction and out-of-order execution, within a single processing core. Future multicore processors will also require the ability to make hardware innovations among multiple cores in order to address the challenges and opportunities of dynamic heterogeneity. This chapter discusses techniques for virtualizing the cores of a multicore processor to enable such hardware innovation, while allowing this innovation to remain transparent to the system and application software running on those cores.

In many respects, this multicore virtualization is analogous to the virtualization and abstraction provided by a dynamically scheduled out-of-order processor. Such a processor abstracts the number and timing characteristics of the functional units from the compiler, automatically extracting parallelism from, while still providing the semantics of, a sequential instruction stream. Similarly, a virtualized multicore can abstract the details of its multiple cores, including their number and the complexity of their dynamic heterogeneity.

The goal of this chapter is to develop an underlying framework to support applications which address the examples of dynamic heterogeneity from Chapter 1. These examples suggest two main requirements for the multicore virtualization framework. First, basic multicore virtualization must support both the ability to move the execution of a virtual processor (VCPU) from one core to another, and the ability to temporarily suspend execution of a VCPU when there are no appropriate cores on which it can run. Section 4.1 proposes techniques to implement this functionality, and discusses how this problem is similar to that of traditional software Virtual Machine Monitors (VMMs), but how its design objectives evolve for multicore processors.

The second major goal of the virtualization framework is to provide the basic support while preserving not only the documented semantics of the hardware/software interface, but also the illusion of other assumptions made by the software, such as simultaneous execution of all VCPUs and performance symmetry. This second goal leads to new challenges, and novel contributions. In particular, Section 4.2 defines and examines the problems of an *overcommitted* system, where the number of VCPUs exposed to the OS exceeds the number of available cores. Hardware spin detection is proposed as a solution to this problem, which allows multicore virtualization to exist within the context of completely unmodified software.

An experimental evaluation in Section 4.4 presents the overheads of various components of virtualization, helps understand the problems of an overcommitted system, and analyzes the effectiveness of the proposed spin detection hardware. An overview of the experimental methodology is described in Section 4.4.1. The complete methodology is presented in the Appendix.

4.1 Basic Multicore Virtualization

The most basic requirements for multicore processor virtualization are mechanisms to allow a VCPU to be migrated from one core to another, or paused when no appropriate cores are available. Such support is needed, for example, if a core is overheating or otherwise unable to perform reliable computation. In this case, it may be desirable to pause the VCPU running on that core for a period of time, or to move that VCPU to another core to continue running.

This basic support is very similar in concept to the processor virtualization performed by traditional software VMMs. There are three primary differences, however, between traditional processor virtualization, and the type of multicore virtualization needed to address emerging dynamic heterogeneity. First, the timescales at which multicore virtualization must act are often shorter than for a typical software VMM. Second, the layer in the system performing the virtualization functionality is different, providing new opportunities. Third, the basic virtualization support is intended primarily to support a single guest VM, eliminating the burden of virtualizing all other aspects of the system.

The first difference arises because several applications of dynamic heterogeneity require context switching and migrating VCPUs on the order of 10 thousand to 100 thousand cycles (e.g, Computation Spreading [Chakraborty et al., 2006] and Mixed-Mode Reliability in Chapter 7). The latencies of such actions thus become much more critical compared to a software VMM, which typically does not perform a VCPU context switch more often than every few milliseconds (VMware ESX Server 3 defaults to 50ms [VMware, 2006a]).

The second difference is that since multicore virtualization is performed beneath the hardware/software interface, a design can include hardware support that is appropriate for that particular generation or product, depending on a projects design goals. This chapter makes the assumption that hardware support can be used to speed up certain operations. However, this hardware support should be restricted to the cases where it most improves performance, and should be kept as simple as possible to maintain designability.

Because the basic multicore virtualization is intended to support only a single guest VM, the third difference arises since many aspects of the system can remain unvirtualized. Necessary tasks involve virtualizing the visible processor state, TLBs, and interrupts, and providing control logic to manage the virtualization. The manner in which these tasks are implemented, within the context of multicore virtualization for dynamic heterogeneity, is described in Sections 4.1.1–4.1.4. Basic multicore virtualization is similar to process migration one level lower: an OS might migrate one of an application's threads to another core, unbeknownst to the application, while a virtualized multicore might migrate one of an OSs VCPUs, unbeknownst to the OS.

Other components of the system that are commonly virtualized by software VMMs include memory, I/O, and privileged instruction execution. These are also the components that tend to add significant overhead to software VMMs. But when supporting only a single guest VM, virtualization of these is not necessary. Keep in mind however, that the use of multicore virtualization does not preclude the use of a traditional software VMM. Such a system simply sits above the hardware/software interface and operates as normal, possibly virtualizing multiple guest OSs. Combining multicore virtualization with a software VMM is used in flexible consolidated server scheduling (Chapter 6), and in Mixed-Mode Reliability (Chapter 7).

4.1.1 Virtual Processor State

A VCPU's *architected state*, which consists of its memory, register values, and for SPARC, its TLB state, must be preserved by the VMM. The memory state can simply be communicated as needed via the on-chip coherence network that is already required to support shared memory multiprocessing. Registers and certain TLB state must be saved and restored, similar in concept to an OS saving the state of a process when it is context-switched.

The UltraSPARC IIICu architecture functionally modeled has a large number of architected registers. Including windowed, alternate global, floating-point, privileged, ASI-mapped, and TLB control registers, this comprises 277 64-bit registers, or 2.2KB.

This dissertation proposes using a simple mechanism with limited hardware support, and no special-purpose storage, by simply storing the VCPU state in the memory hierarchy. A portion of the physical address space is set aside for this storage, and microcode is used to load and store the state values. When the VMM wishes to deschedule a VCPU, it interrupts the corresponding core, which flushes its pipeline and executes the microcode to save the current VCPU state. The microcode can then restore the state for the next VCPU (as specified by the VMM). The latency of this operation is then determined by the available memory read (and write) ports and the cache bandwidth. While this solution incurs some cache overhead, it is a small fraction of the multi-megabyte caches found on current chips. If the VMM chooses to run the descheduled VCPU on another core, the VCPU state can be transparently migrated using the on-chip coherence protocol when the microcode executing on the new core begins loading the state.

4.1.2 TLB Virtualization

SPARC V9 uses a software managed TLB, which means that most storage and operational aspects of the TLB are architected and OS-visible. TLB storage consists of control registers and cached page translations (TLB *entries*). Control registers are treated just as all other processor control registers, and migrated as part of the VCPU state.

TLB entries can also be treated just like the other registers, however, the Cheetah+ MMU used in the UltraSPARC IIICu processor has nearly 20KB worth of TLB entries per VCPU (1,184 quad word entries). Migrating these entries as part of VCPU state can result in significant cache and latency overheads (see Section 4.4.2).

TLB entries can be either *locked* or *unlocked*. Locked entries are managed completely by software. They are used for certain OS code pages (e.g., TLB miss handler), and other pages for which the OS cannot tolerate a miss. Locked entries are sometimes different for different VCPUs, even when all VCPUs are running the same OS. Thus, locked entries must be treated as part of the VCPU state. Fortunately, the architecture is limited to 32 locked entries (16 each for instruction and data TLB), comprising a small fraction of VCPU state.

With unlocked entries, the OS has more limited control. In particular, the TLB is allowed to evict these entries at nearly any time. A second option for virtualization is thus to invalidate all unlocked entries when a VCPU is descheduled. However, all future accesses after transferring to a new core (or after being rescheduled on the same core) will result in a TLB miss, leading to high overheads as well (see Section 4.4.2).

A third option is to tag each entry with a VCPU ID. During translation, a TLB entry is used only if its VCPU ID matches the currently executing VCPU. Although this option keeps the overhead of pausing and migrating VCPUs low, if migrations are frequent or TLB working sets are small, it requires more hardware support than the prior options. First, it requires additional storage for each entry and an additional tag match on each lookup. Second, TLB demap operations must be handled with care, since a TLB entry can remain valid even after a VCPU has moved to another core. If the translation is demapped on the new core, a hardware shootdown mechanism is necessary to demap the translation on other cores as well.

In this work, we make the observation that a VCPU ID is not needed when sharing the TLB array among multiple VCPUs from the same VM. This observation holds since SPARC V9 uses OS-visible context IDs to tag TLB entries. Although an OS is free to use these context IDs for whatever purpose it sees fit, both Solaris and Linux use a unique context for each process address space. Thus, when a software process is migrated from one OS-visible VCPU to another, or when multiple threads of the same process simultaneously execute on multiple VCPUs, the same TLB context ID is used. VCPUs from the same VM are thus allowed to share unlocked TLB entries, and do so without requiring additional tag hardware. Such sharing increases the effective size of the TLB, since any translation shared among two or more VCPUs is not replicated.

This final option does not require any additional hardware on the critical path of a translation, though unlocked entries are still invalidated when switching between VCPUs of different VMs (e.g., in Chapter 6). A hardware shootdown mechanism is also still necessary. A comparison of three of these possible mechanisms is presented in Section 4.4.2.

For other architectures, which do not implement an architected TLB, all of these proposed techniques remain applicable. Sharing entries among VCPUs, however, requires context IDs, which some architectures, e.g., X86, do not use.

4.1.3 Interrupt Virtualization

The final step in the virtualization of on-chip cores is a mechanism for properly handling interrupts, including both hardware- and software-initiated interrupts.

This work assumes the use of a centralized interrupt control logic, implemented in hardware, for this purpose. A simple table is used to map an incoming interrupt from the proper VCPU to the core currently executing that VCPU. If the VCPU is currently executing, the interrupt is then delivered to the physical core. If the table reports that the VCPU is currently paused, then the interrupt is buffered until the VCPU is run again.

For most experiments, this dissertation uses the policy of quickly scheduling a paused VCPU on the core where it was most previously executing, and then delivering the interrupt with minimal delay. As discussed in Section 4.2, such a policy is particularly important for software-initiated interrupts (i.e. CPU cross-calls), as the initiating software often waits for the recipient to acknowledge receipt.

4.1.4 Virtualization Controller

Virtualization control logic is performed through a centralized physical structure called the *Virtualization Controller* (VC). This structure is assumed to be implemented in hardware, and receives inputs from the cores and its own timer. It consists of a table mapping each VCPU to its current core (which can be none), and simple control logic to direct the scheduling of VCPUs. The actual function of the control logic depends upon the virtualization application. A practical implementation of the VC could also use programmable logic.

4.2 Overcommitted Virtual Machines

Many examples of dynamic heterogeneity from Chapter 1 create a varying number of available or appropriate physical cores. In order to provide efficient utilization of the machine when the maximum number of cores *are* available, a chip manufacturer (or machine administrator) may wish to expose each of those cores as an OS-visible VCPU. During times when fewer cores are available than the number of exposed VCPUs, the system is said to be *overcommitted*, in the sense that the OS expects each VCPU to continue executing, but there are not enough cores to do so simultaneously. This section addresses how a virtualized multicore can make an overcommitted system possible.

Current software VMMs, such as VMware, allow the *total* number of VCPUs, across multiple VMs (each VM running its own OS), to be overcommitted as long as the number of VCPUs exposed to any individual VM is less than or equal to the minimum number of available cores. The unique contribution of this section is that the proposed techniques enable a *single* VM to be overcommitted as well. These techniques, for example, allow one or more VMs running in a traditional software VMM to each be overcommitted, allow a single OS running on the machine without a software VMM to be overcommitted, or even allow a software VMM itself to be overcommitted.

This section discusses the problems associated with overcommitting for a single OS, and proposes a solution to do so even with completely unmodified software. Section 4.4 presents an experimental analysis of the problems and proposed techniques. Chapters 5, 6, and 7 qualitatively and quantitatively demonstrate why overcommitting a single OS is desirable, and perhaps even necessary, for future multicores.

4.2.1 The Problem: Synchronization Overhead

When overcommitting the cores of a processor, multiple VCPUs must share each core. But a multiprocessor OS which is unaware of any virtualization underneath assumes all its VCPUs are executing simultaneously. Obviously, this assumption does not hold for an overcommitted system.

In order for each VCPU to make forward progress, the Virtualization Controller (VC) allows each VCPU sharing a particular core to run for a maximum amount of time before context switching the running VCPU for another one. This maximum time is called the *timeslice*. For correctness reasons, especially with overcommitting with a single OS, the timeslice must be short enough such that the OS doesn't panic after realizing several of its VCPUs are not executing. For Solaris 9, the maximum timeslice is approximately one second. The maximum timeslice used by the VC is specified in a register, and can be updated by the system administrator.

Kernel panics with long timeslices are not the only problem, however. In particular, a paused VCPU can be holding an OS kernel lock, or be the recipient of a software interrupt (CPU cross call), leading to synchronization problems and live-lock and/or severe performance loss. The two major components of this synchronization problem are described in more detail below.

Mutex Locks The first problem manifests itself through the use of mutex spin locks to protect critical sections in the OS kernel. For example, a Solaris kernel thread attempting to acquire an adaptive mutex lock will sometimes block when the lock is already held, but for performance reasons, will spin when the owner of the lock is running on another VCPU (assuming the lock will not be held long). If this lock-holding VCPU is not currently executing on a core, however, the thread will spin unnecessarily. Other, non-adaptive locks are not preemptible and always spin. Linux 2.6.10, the version used for this study, does not use adaptive locks and always spins, though it sometimes uses semaphores for preemptible synchronization.[1]

[1]The Linux 2.6.16 kernel added support for adaptive mutex locks using a similar policy as Solaris.

Cross-Calls A multiprocessor OS often employs synchronous software interrupts, or cross-calls, to communicate between different (virtual) CPUs. An example of this use is for TLB shootdowns, where one CPU wishes to change a page mapping (e.g., to invalidate an entry or update the protection bits) that may be cached in other CPUs' TLBs [Rosenburg, 1989]. Another example, prevalent in the Solaris kernel, is the use of cross-calls to preempt one or more remote CPU(s) to run a higher priority thread. Linux does not use cross-calls for this type of scheduling action. When a cross-call is invoked, the software spin-waits until the hardware notifies it that the interrupt has actually been delivered to the remote CPU.

For SPARC, the remote CPU sends a NACK when it already has an outstanding software interrupt, in which case, the sender must continue to retry until the request is ACKed. Other architectures, such as x86, also require software to ensure that these interrupts are not lost.

Fast execution of cross-calls becomes challenging in an overcommitted environment where all the VCPUs of a VM are not running concurrently. Since OSs typically do not use nested locking, the forward progress of the lock-holder is always guaranteed in a simple mutex lock. But in a cross-call, the forward progress of the initiating VCPU, which is often holding a lock (such as page table lock or run-queue lock), is dependent upon the recipient VCPUs. For example, when multiple VCPUs send interrupts to a paused VCPU before they can be handled, one or more of the senders will continue to spin until its interrupt can be delivered. In other words, when any responding VCPU is paused with a pending software interrupt, it will thwart the forward progress of the interrupt initiator, which in turn will cause a cascading affect on other VCPUs attempting to acquire the lock held by the initiator.

Modifying the interface between the OS and hardware to allow (and guarantee delivery of) an unlimited number of outstanding interrupts might alleviate this problem. This chapter, however, focuses on unmodified software.

4.2.1.1 User Code

The problems associated with preempting a lock holder are well understood in the context of user programs [Anderson et al., 1992; Wisniewski et al., 1993; Zahorjan et al., 1991]. The results of this work has led developers of many parallel applications to realize that a user application is already running in a virtual environment provided by the OS. The OS can, and will, preempt a user thread at any time for any number of reasons. User applications thus tend not to use synchronization constructs that assume all threads are always simultaneously scheduled. Instead, user applications often also use an form of an *adaptive* mutex lock, i.e., one that may spin for a short period of time, and then block. None of the server-class applications in this study have a significant user spin component. `barnes` does have user spin, but implements adaptive locks that yield to the OS scheduler after a short time.

4.2.2 The Solution: Hardware Spin Detection

In any parallel application (a multiprocessor OS being a complex and interesting example), there are two generic approaches to mitigate the synchronization problem among threads which are not concurrently executing: 1) avoid preempting a thread holding a lock or 2) pro-actively preempt a thread that is excessively spinning in favor of executing a more productive thread.

In an optimized parallel application, the length of critical sections is typically short. Avoiding preemption of the lock holder can often yield good performance for such applications. However, this approach requires precise information about the lock holder, which is unavailable from an unmodified multiprocessor OS running in a virtual machine.

The second approach to reduce synchronization overhead is to detect when a VCPU is excessively spinning on a lock and preempt the VCPU at that point. In order to avoid making any modifications to the OS, this section proposes a simple yet effective heuristic to identify spin loops in hardware by observing the dynamic instruction stream.

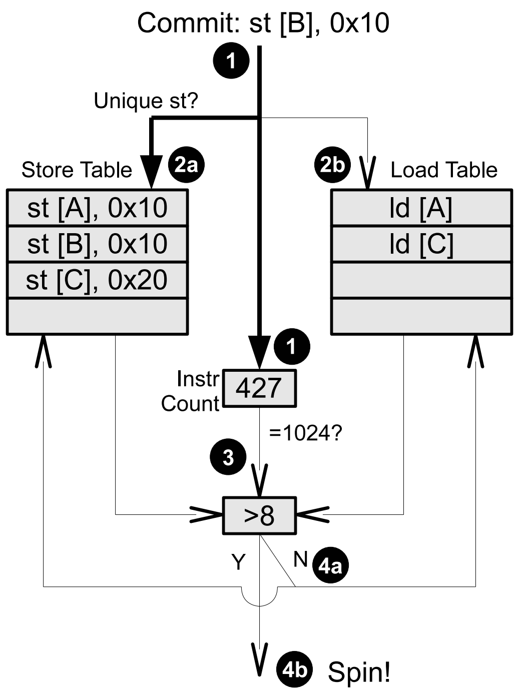
The proposed heuristic relies on the observation that a program executing in a spin loop has a distinctive execution pattern. While waiting for certain events and not making any forward progress, a thread typically makes very few, if any, modifications to the program state. We can infer this lack of program state modifications from the absence of store instructions that change values in memory. Consequently, this execution pattern can be easily recognized by observing few *unique* stores committed by the program in a given interval, where the uniqueness of a store is determined by having an address or value different from other stores.

An important exception to the above observation arises when a memory location is register allocated. For example, while searching an array structure, a program may not execute any store instructions since the array index variable is likely to be register allocated. Therefore, during long search operations, predominantly found in user code, a program may not execute any store instructions. To avoid such false positive spin detections, we also check for unique load instructions (uniqueness determined by the load address only) when executing user code.

Thus, a kernel spin is detected when the number of unique stores executed within N committed instructions is less than some pre-defined threshold. On the other hand, a user spin will be detected when both unique stores and loads are less than that threshold. Sensitivity experiments demonstrate that for a period of N=1024 committed instructions, a threshold value of eight is effective to detect all known spin loops with near-zero false positives.

4.2.2.1 Spin Detection Buffer

This section proposes a simple hardware structure, the *Spin Detection Buffer* (SDB), to implement spin detection functionality. It employs two fully associative, eight entry content-addressable memory (CAM) structures to hold the unique stores and loads, respectively. During a given period N instruction (N=1024 for the experiments here), each committed store (and load when in user mode) searches the appropriate CAM to determine if its address/value is unique. A unique load



1 The core commits an instruction and increments the *Instruction Count* register.

2a For a store, the SDB inspects the store table to determine if that store is unique, and if so, inserts it. The current store is not unique, and the table is not updated.

2b For a load, if the processor is in user mode, the SDB inspects the load table, and inserts an entry if the load is unique.

3 When the committed instruction count reaches 1024, the number of entries in each of the tables is examined. If either table reaches 8 entries before the counter reaches 1024, steps 2a and 2b are not performed.

4a If one table has 8 entries, the SDB is reset by clearing the tables and zeroing the counter. The SDB is also reset when a user/OS mode switch occurs.

4b If neither table has 8 entries, insufficient unique stores and/or loads were observed during the 1024 window of instructions, and a spin is reported to the Virtualization Controller.

Figure 4.1 Spin Detection Buffer (SDB) Organization and Operation

or store then inserts its address/value into the appropriate CAM array. Once either array becomes full, subsequent instructions need not search the CAM.

At the end of the period of committed instructions, the SDB simply checks the number of entries in each array. If there are less than eight valid entries in the store array and the VCPU is executing in the OS, the SDB indicates a spin. If there are less than eight entries in both arrays and the VCPU is executing user code, the SDB again indicates a spin. Otherwise, the arrays are flushed and it is assumed that the VCPU is making forward progress. If a user/OS mode change occurs within the period, forward progress is assumed regardless of the number of entries in the arrays. Figure 4.1 shows the organization and describes the operation of the SDB.

Updates to these structures are not on the critical path, since they are performed after instructions commit. Because these arrays do not have a strict correctness requirement, other optimizations could be performed to avoid using a CAM, if desired (e.g., a Bloom filter). Even with a

CAM, however, the power requirements of the SDB are likely to be minimal because each structure is only 8 entries, and during normal (non-spinning) execution, is accessed at most 8 times in a window of 1024 instructions.

Using the SDB, the maximum timeslice offered to each VCPU by the VC can be cut short if that VCPU is observed to be spinning. The result is that the *effective timeslice*, i.e., the amount of time each VCPU actually executes on the core before being preempted, can be much smaller than the maximum timeslice specified by the system administrator. The minimum effective timeslice is the amount of time taken to commit 1024 instructions.

4.3 Hardware/Firmware Complexity

The multicore virtualization techniques presented in this chapter involve modest hardware and firmware complexity. Hardware support is needed to load and store VCPU state, handle TLB shootdowns, implement the Virtualization Controller (VC), and spin detection.

The functionality of transferring a VCPU from one core to another is assumed to be implemented mostly using firmware/microcode with loads and stores to a reserved portion of the physical address space. Yet some hardware support is necessary to allow microcode to run with sufficient permission to access all state registers. The microcode itself is expected to be relatively simple.

TLB demaps from a VCPU running on a core need to be sent to other cores on which that VCPU may have executed. This operation is simpler than a normal hardware TLB shootdown, however, since the shootdown does not have nearly as strict synchronization requirements. The shootdown needs only to occur before the VCPU performing the demap is migrated again to the other cores, but not before the remapping VCPU is allowed to continue executing. The reason is that, in the absence of migration, the system software still maintains TLB consistency among VCPUs.

The VC requires a small, infrequently accessed table mapping VCPUs to their currently assigned cores. This table is necessary both for scheduling decisions and for interrupt delivery. The VC also requires scheduling logic with inputs from each core, spin detection hardware, and mapping table. This logic performs simple scheduling decisions, directs the migration of virtual processors, and maintains the mapping table. For Chapters 5 and 7 the VC is assumed to be hardened or replicated to protect against faults.

The hardware complexity of the SDB is described in Section 4.2.2.1.

4.4 Experimental Results

After describing the virtualization methodology, this section presents experimental results for the overheads of basic virtualization, the need for spin detection in an overcommitted system, and results of an overcommitted system using the SDB.

4.4.1 Methodology

As described in Chapter 3, experimental results in this and other chapters are derived from a full-system, execution driven simulator. Experiments in this chapter use both the out-of-order (OoO) and in-order core models. Experiments in Section 4.4.2 use the OoO core, are run for 90-150 million cycles, and start with caches already warmed up.

In order to run simulations for a longer simulated time, the in-order model is used for Sections 4.4.3, 4.4.4, and 4.5. Most simulations using this configuration are run for more than 1 billion cycles, and start with cold caches.

Virtualization Implementation In general, the Simics functional simulator correctly handles the execution of the VCPUs, while the timing component dictates when and with what timing characteristics those VCPUs execute. However, the various components of virtualization interact with the functional and timing components of the simulator in different ways.

The scheduling controller is implemented in the timing simulator. It receives input from various parts of the simulator, makes a scheduling decision, and implements that decision by informing each core which VCPU it should be running presently. Timing aspects of the the scheduling controller are not modeled.

VCPU state manipulation (Section 4.1.1) is implemented entirely within the timing simulator. Real firmware instructions are not implemented. Instead, when a core receives a request to stop executing a particular VCPU, it drains the pipeline, and then begins storing the register state into a dedicated physical address space set aside for that VCPU. Stores are sent to the memory hierarchy two per cycle to model execution bandwidth constraints. This bandwidth also matches the bandwidth of the L1 caches. Once the state has been stored, the core notifies the scheduling controller, which will then either inform the core to load the state of a different VCPU, or to go idle. Two loads are also issued to the memory hierarchy per cycle, invoking coherence and main memory misses when necessary, and may complete out of order. This methodology essentially models hardware support to store and load registers without any overheads from software. A true firmware-based mechanism, though allowing simpler hardware, may involve additional overhead.

Interrupt virtualization is handled mostly by the functional simulator. Interrupts are simply delivered to the appropriate VCPU by Simics. However, if that VCPU is not currently running on a core, the interrupt is buffered by the functional simulator and actually delivered once the VCPU begins executing again. Only one interrupt is buffered per VCPU. Any additional interrupts are NACKed.

A significant portion of the TLB virtualization is performed by modifying the TLBs used by the functional simulator. The mapping of VCPUs to TLBs is not altered, meaning that control registers need not be manipulated. However, the TLB mappings are changed (or invalidated) during simulation to reflect the mapping that would be present in the appropriate physical core's TLB.

During normal execution, when the OS wishes to modify a virtual-to-physical mapping (i.e., virtual-to-real mapping in the context of processor virtualization), it will send a TLB shootdown request to every processor (i.e., VCPU) which might be caching that translation. Rather than send shootdowns to all processors, Solaris tracks which processors execute a particular applications, and only sends shootdown requests to that subset. When VCPUs are moved from one core to another, and when the TLB arrays are shared among different VCPUs, the OS cannot know which cores' TLBs may contain a mapping that it wishes to change or invalidate. Thus, for execution correctness, the timing simulator must cooperate with the functional TLBs to perform shootdowns on any core which may have executed the VCPU performing a TLB demap operation. The simulator pessimistically shoots down all cores' TLBs, and does not model the latency of that operation.

The SDB is modeled within the timing simulator by examining the committing instruction stream. It notifies the scheduling controller when a VCPU is suspected of spinning.

The control logic controls the maximum (non-spinning) timeslice of each VCPU. This value can be configured to meet the demands of the application.

4.4.2 Basic Virtualization Overheads

Several experiments were performed to measure various components of the overhead involved in the proposed multicore virtualization. In these experiments, the number of VCPUs matches the number of cores, and the VCPUs are not migrated among cores. Instead, VCPUs are periodically interrupted, descheduled (which involves draining the pipeline, storing VCPU state into the memory hierarchy, and taking any appropriate action with TLB entries), and then immediately rescheduled on the same core (which involves reloading the VCPU state from the memory hierarchy). This enables the measurement of the latency and cache overheads of saving and restoring

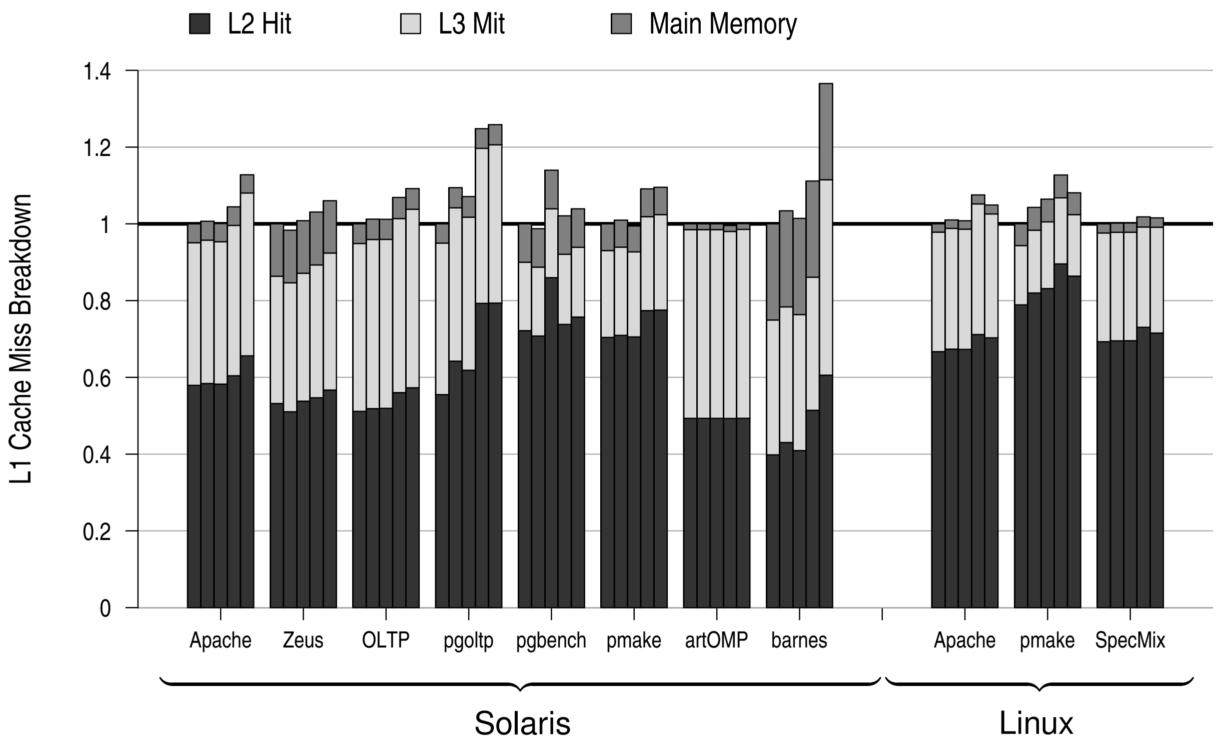


Figure 4.2 Cache Overheads of Virtualization. *Breakdown of the cache overhead (additional misses) of the various TLB schemes relative to an unvirtualized machine. Bars from left represent the unvirtualized baseline,* Ideal TLB*,* Shared TLB*,* Transfer*, and* Invalidate *schemes.*

VCPU state, as well as the overheads of the various options for handling TLB entries, while avoiding the cache effects of migration. The periodic interrupts occur every 100k cycles (33μs) for these experiments.

Figures 4.2 and 4.3 show the cache and runtime overheads of the basic multicore virtualization, relative to an unvirtualized machine, using several schemes for handling the architected SPARC TLB. L1 cache misses are broken down into L2 hits, L3 hits, and misses to main memory. Misses incurred for lines containing VCPU state are *not* included, only the additional misses to other lines resulting from the additional cache pressure of the VCPU state. *Ideal TLB* represents the performance of an unvirtualized TLB, where each VCPU maintains its architected TLB without cost, though the remainder of the VCPU state is properly virtualized. *Shared TLB* represents the scheme, used throughout the rest of this dissertation, where unlocked TLB entries are shared

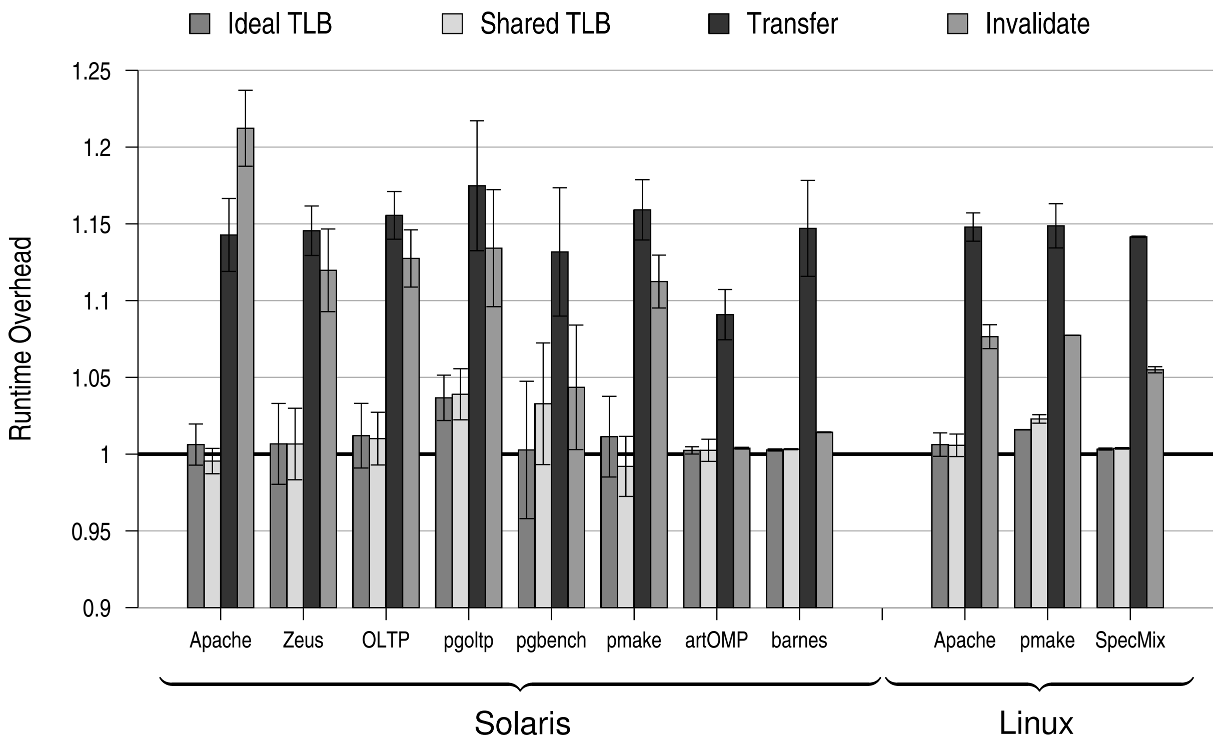


Figure 4.3 Runtime Overheads of Virtualization. *Runtime overhead of the various TLB schemes relative to an unvirtualized machine.*

among VCPUs from the same VM. For this experiment, without multiple VCPUs on each core, *Shared TLB* differs from *Ideal TLB* in that the overheads from maintaining TLB control registers, locked entries, and hardware shootdowns are included. *Transfer* represents the scheme where all TLB entries are included in the VCPU state and hence saved and restored from the caches. *Invalidate* represents the overhead from invalidating all unlocked entries on a VCPU switch.

The cache overheads when the TLB is ideally virtualized are minimal, as shown by Figure 4.2. Many workloads incur a few additional L1 and L2 misses, but not significant additional off-chip misses. Results are very similar for the *Shared* scheme, the primary difference being only 280 bytes of state per VCPU. Treating the entire TLB state as architecturally visible VCPU state, as is done with the *Transfer* scheme, however, does create significant additional cache pressure, resulting in 5–10% additional L1 and L2 misses. Notably, invalidating the TLB also creates significant cache pressure for several TLB-intensive workloads, including `apache` and `pmake`. Although the VCPU state is the same size as with the *Shared* scheme, the act of refilling the TLB entries using the TLB miss handler causes additional misses to the caches that are not present for the other schemes.

As shown in Figure 4.3, the runtime overhead when the TLB is ideally considered is negligible for all benchmarks. In addition to minimal cache pressure, the latency of storing and loading the register state is approximately 350 cycles, less than 1% of the 100k cycle period. Similarly, the runtime overhead for the *Shared* scheme is also minimal — less than 1% for all benchmarks. The overhead when transfering all TLB entries, however, is 10–20%. This significant overhead is due in part to additional cache pressure, and in part to the time required to load and restore 22KBytes worth of VCPU state (which is also likely to incur many cache misses).

For applications which make heavy use of the TLB, such as `Apache`, invalidating all entries every 100k cycles has a performance nearly as large, or greater than, the *Transfer* scheme. For other applications, which have such a large TLB footprint that most of the entries are dead (e.g., `artOMP`), the runtime overheads are similar to the *Shared* scheme.

In summary, the overheads of the *Shared* scheme, used throughout the rest of this dissertation, are less than 1% of runtime for all workloads when using a timeslice of 100k cycles (33μs). Even with significantly smaller timeslices, the overheads are still small. Thus, a more hardware-intensive solution for managing VCPU state is unnecessary for the applications of multicore virtualization that are considered in this dissertation.

4.4.3 Spinning in an Overcommitted System

To expose the challenges of virtualizing a multiprocessor OS in an overcommitted environment, we consider the case of a single guest VM, configured to use 24 VCPUs, but run on an eight core system. In other words, a multicore processor with only eight (available) cores still exposes 24

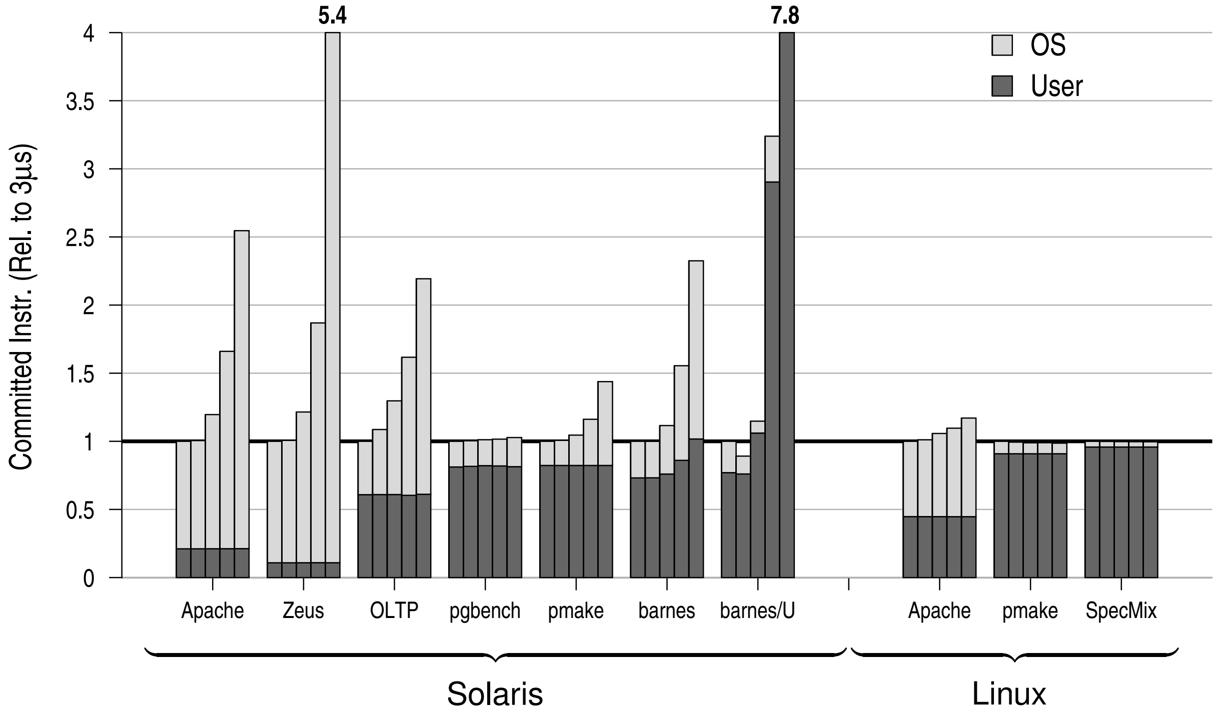


Figure 4.4 Normalized Instruction Count for Various Timeslices. Five bars for each benchmark (from left to right) represent results for 3μs, 6μs, 17μs, 33μs and 66μs timeslices. Results are normalized to the 3μs timeslice.

cores to the OS. There are possibly hundreds of software threads that the OS schedules among its 24 VCPUs. The hardware/firmware layer implemented within the processor then selects 8 VCPUs to execute on the cores at any given time. To maintain cache and TLB affinity, we assume that three VCPUs are mapped to a given core, though only one is executing at a time. We avoid live-lock and kernel panic by ensuring that each VCPU periodically receives a share of the time on its assigned core. Each core preempts the running VCPU after a given *timeslice* so that the other VCPUs can run. Preemption occurs independently for each core.

Figure 4.6 shows the breakdown of instructions executed by the user code and the OS, respectively, for various VCPU timeslices, while performing the same amount of work (i.e., the same number of workload *transactions*). While the aggregate user instructions executed remain

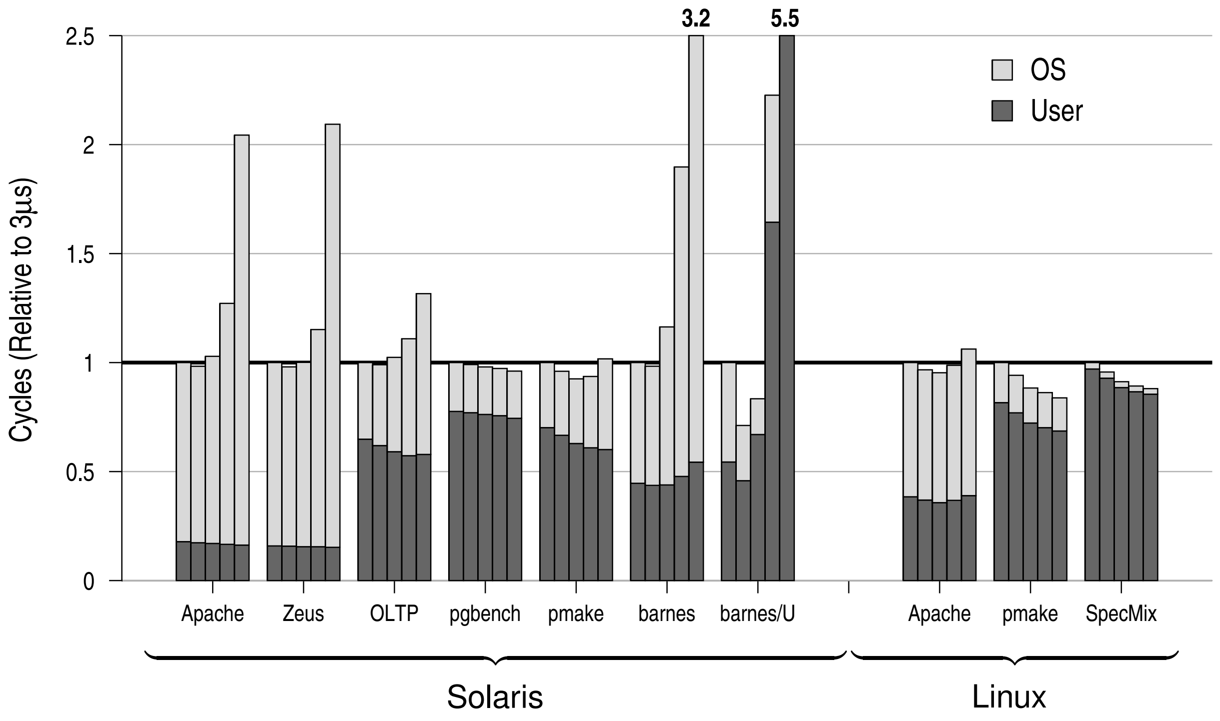


Figure 4.5 Normalized Runtime for Various Timeslices. Five bars for each benchmark (from left to right) represent results for 3μs, 6μs, 17μs, 33μs and 66μs timeslices. Results are normalized to the 3μs timeslice.

very stable across different timeslices, several workloads observe a dramatic increase in OS instructions for larger timeslices. For example, `OLTP` executes 2.2 times more instructions with a timeslice of 66μs compared to 3μs, entirely due to OS instructions. Longer timeslices, such as 1ms, can cause a kernel panic in several Solaris workloads. `barnes` and `barnes/U` are interesting cases, as they are the only benchmarks to have any increase in user instructions. These benchmarks execute two barriers every time step, synchronizing all 24 VCPUs. In `barnes`, the barrier wait, implemented by a call to `pthreads_cond_wait()`, spins for awhile in user mode, and then blocks, yielding the thread to the OS scheduler. But the OS scheduler has nothing else to run on that VCPU, so it becomes idle. Nearly all of the increase in OS instructions from `barnes` are from the OS idle loop. For `barnes/U`, which implements the barrier with user spin locks, the increased instructions is almost entirely from this simple `while(barrier.spin)` loop. Note that the problem size of both `barnes` and `barnes/U` has been reduced to make synchronization important.

The workloads running on Linux show significantly less overhead due to additional OS instructions compared to their Solaris counterparts, largely due to the absence of frequent cross-calls. However, `Apache` on Linux also shows excessive spinning when using longer timeslices.

The relative runtime for the same timeslices is shown in Figure 4.7. Although not as dramatic for some benchmarks (note the different scale on the y-axis), the runtime for `Apache` on Solaris for the 66μs timeslice is still more then 2.5 times that of the 3μs case, and that of `barnes` and `barnes/U` balloons by 3.2 and 5.5 times, respectively. While a timeslice as low as 3μs can mitigate the OS spin problem, it can also hurt cache locality (in addition to incurring the other virtualization overheads from Section 4.4.2). For example, the 66μs timeslice of `pmake` on Solaris, despite a 50% increase in instructions due to spinning, has no change in runtime since it observes a 33% reduction in L1 misses compared to the 3μs timeslice. `pmake` on Linux, which has little to no synchronization overhead, sees a 35% reduction in L1 misses, and a 20% reduction in runtime. These results clearly indicate that, in order to maximize cache locality, we need a virtualization solution which allows VCPUs to run for a longer timeslice when they are performing useful work.

4.4.4 Spin Detection Results

This section examines the ability of the SDB to mitigate the synchronization overhead apparent from Section 4.4.3. Figures 4.6 and 4.7 show the committed instructions and runtime, respectively, when using the SDB to detect spins. Experiments use a base (non-spinning) timeslices of 3–66μs. Results in both graphs are normalized to the 3μs timeslice without spin detection.

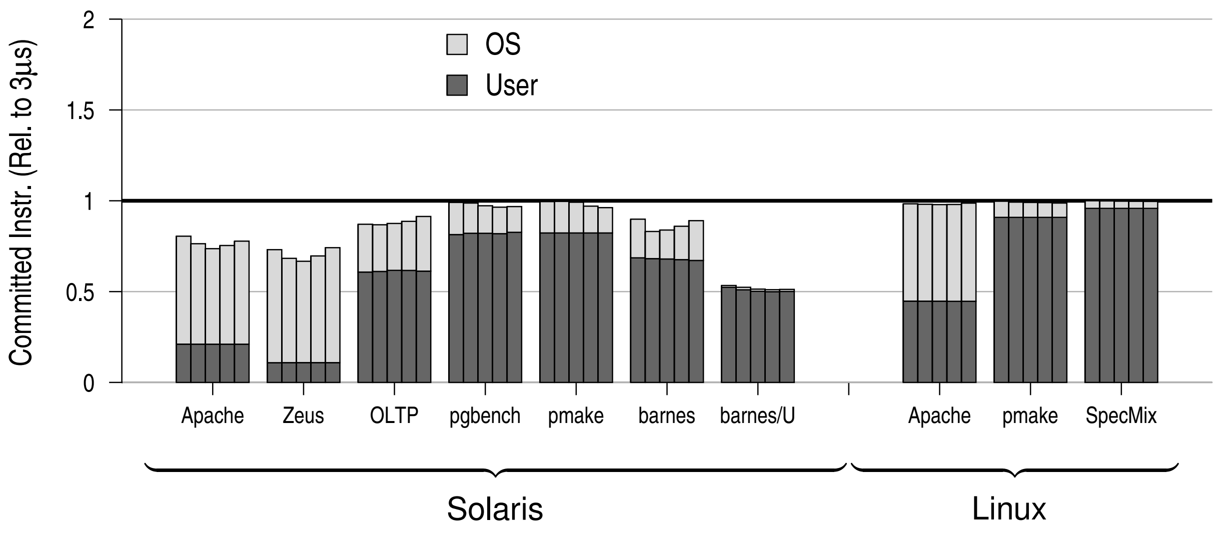


Figure 4.6 Normalized Instruction Count using the SDB. Five bars for each benchmark (from left to right) represent results for 3μs, 6μs, 17μs, 33μs and 66μs timeslices. Results are normalized to the 3μs timeslice.

Unlike the results without spin detection, increasing the timeslice has little if any effect on the number of committed instructions when using the SDB (Figure 4.6). Both user and OS instructions now remain stable across a range of timeslices. For `Apache`, `Zeus`, and `OLTP` on Solaris, even the baseline 3μs timeslice has significant spinning, which is eliminated with the SDB. For `barnes`, and to some extent, `OLTP`, often all three VCPUs that are assigned to a given core are spinning waiting for one or more VCPUs on some other cores. In this case, the spinning VCPUs will continuously rotate on their core, each executing 1024 useless instructions every time, slightly inflating the number of instructions for longer timeslices. When all VCPUs currently assigned to a core are idle, a more sophisticated Virtualization Controller could migrate non-idle VCPUs currently paused on other cores, but this additional complexity is not modeled.

As evident from Figure 4.7, however, increasing the timeslice improves cache locality and reduces virtualization overheads, providing runtime reductions of 10–20% for most benchmarks.

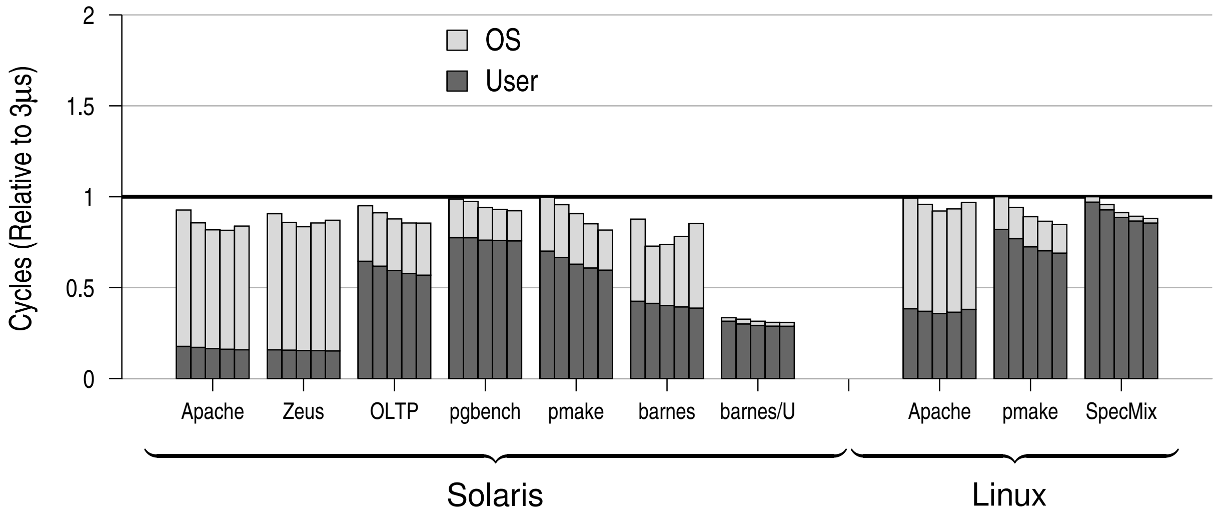


Figure 4.7 Normalized Runtime using the SDB. Five bars for each benchmark (from left to right) represent results for 3μs, 6μs, 17μs, 33μs and 66μs timeslices. Results are normalized to the 3μs timeslice.

Though the number of instructions is stable, the runtime of both OS and user code decreases with longer timeslices due to decreased cache misses.

The first column of Table 4.1 shows the average *effective timeslice* when using spin detection with a maximum timeslice of 200k cycles (the configuration shown in the rightmost bars of Figures 4.6 and 4.7). Workloads with the most spinning when not using the SDB (e.g., `Apache`, `Zeus`, `OLTP`, `barnes` and `barnes/U`) have the shortest effective timeslices, since whenever a spinning VCPU is rescheduled, it will only execute for 1024 instruction, or approximately 4k cycles (at average IPCs of ~0.25). On the other hand, workloads such as `SpecMix`, which observes practically no spinning, and `pmake` on Linux, which observes very little spinning, often allow their VCPUs to use the maximum timeslice of 200k cycles.

The second column of Table 4.1 shows the average latency to switch out one VCPU and switch in another. The switch requires 277 stores and 277 loads, which with a bandwidth of 2 each per cycle, take a minimum of 277 plus 15, or 292 cycles, if all lines are in the L2 cache (but not all are in the L1). Workloads with the shortest average timeslice access the VCPU state most

Workload	Effective Timeslice	Average Switch Latency
Apache	21k	291
Zeus	29k	293
OLTP	32k	294
pgbench	164k	333
pmake	82k	315
barnes	13k	286
barnes/U	20k	291
Apache (Linux)	113k	320
pmake (Linux)	174k	329
SpecMix (Linux)	199k	335

Table 4.1 Effective Timeslice and Switching Overhead using the SDB (cycles)

frequently, and are thus most likely to find all lines in the L2 and have the shortest latencies. Some benchmarks, notably `barnes`, often find most of their VCPU state in the L1, and are thus slightly faster than 292 cycles on average. The other workloads may need to go to the shared L3, or even off chip, to get some of the lines, and observe additional overheads of up to 43 cycles. These overheads are kept low because the order in which VCPU state addresses are accessed is optimized to pipeline cache misses. These latencies do not include draining the pipeline.

4.5 Related Work

There is a large body of work from virtualization research and practice. This section focuses on the two categories which are most closely related to the multicore virtualization presented in this section: hardware support for virtualizing VCPU state, and techniques that may also enable an overcommitted system.

4.5.1 Multicore Virtualization

Recently, both Intel and AMD have introduced hardware virtualization support with their Virtualization Technology (VT) [Uhlig et al., 2005] and SVM [Advanced Micro Devices, 2005] projects, respectively. Of particular relevance, both projects provide hardware support for VCPU switching and migration.

The primary difference between these technologies and the ones proposed in this chapter is that the hardware mechanisms are directly exposed to the software and exploited under the direct control of a traditional software VMM such as VMware or Xen. This chapter assumes that any hardware support is under the direction of a hardware/firmware virtualization layer that exposes a standard homogeneous multicore to the lowest level of system software.

One result of this difference in layering is that the VCPU state in this chapter is stored in reserved physical memory, whereas storage for the VCPU state in VT and SVM is created and manipulated by the software VMM. When the software VMM wishes to pause a VCPU, it executes an instruction to swap out the current VCPU, and another instruction to swap in the new VCPU. The effect of software managed storage, however, means that migrating VCPU state among cores requires significant software/hardware cooperation to ensure consistent state.

The microarchitectural mechanisms for managing and migrating VCPU state are not disclosed. But since VCPU state is stored in the VMM's visible physical memory, it would be reasonable to assume that migration is similarly implemented by utilizing the cache coherence protocol.

In an effort to maintain strict backward ISA compatibility, SVM tags TLB entries with VCPU IDs, similar to the approach proposed in Section 4.1.2, allowing entries from multiple VCPUs to coexist in the TLB, but not necessarily sharing them even if they are from the same guest VM. It is not clear whether Intel's VT does the same.

4.5.2 Enabling Overcommitted Systems

The primary issue when overcommitting a single guest VM is managing the overheads of software synchronization. As mentioned in Section 4.2.2, there are two basic techniques that can be used to do so: 1) avoid preempting a thread holding a lock or 2) pro-actively preempt a thread that is excessively spinning in favor of executing a more productive thread.

Uhlig, et al., have proposed one technique for avoiding lock-holder preemption, with the similar objective of flexibly executing a subset of a guest VM's VCPUs on a restricted number of available cores [Uhlig et al., 2004]. They make the observation that a VCPU executing in user mode is not holding a kernel lock, and can thus be safely preempted. These preemptible locations are referred to as *safe points*. A VCPU executing in the OS may be holding a kernel lock, and thus, the VMM will try *not* to preempt it at that point.

Speculative Lock Elision provides another way to avoid preempting a lock holder: infer the acquisition and release of a lock from the atomic operations typically used by software to implement the lock semantics [Rajwar and Goodman, 2001]. An atomic read-modify-write (e.g., `casa` or `ldstub` in SPARC) which return non-zero is assumed to be a lock acquire. A subsequent store to the same address is presumed to be a lock release.

Both of these techniques have the potential to identify when the OS (or user application for SLE) is holding a lock, and thus, can help prevent some of the synchronization overheads. However, both schemes fail to efficiently handle the frequent cross-calls seen in the applications running Solaris. A reimplementation and performance evaluation of Safe-Points is discussed below. SLE-style inference is not evaluated.

The second approach to reduce the synchronization overhead is to detect when a VCPU is excessively spinning on a lock and preempt the VCPU at that point. This is similar in concept to *helping locks* proposed by Hohmuth and Hartig [2001]. Most current solutions for identifying spin locks and loops (including the OS idle loop) involve OS-intrusive modifications or kernel PC annotations. Several existing projects use this method (often just for the idle loop), including IBM's Power5 Hypervisor [Armstrong et al., 2005], Cellular Disco [Govil et al., 1999, 2000], and one of the solutions offered by Uhlig et al. [2004].

Li, et al., proposed a different mechanism for hardware spin detection [Li et al., 2006], simultaneously with the proposals in this chapter [Wells et al., 2006]. Their technique detects a spin loop when it can be *guaranteed* that no forward progress is being made by an application. They rely on a significant hardware component in order to detect temporally silent register writes, but require that after each iteration of a loop, *no* changes have been made to the architectural state of the processor.

Both the SDB proposed in this chapter and the hardware proposed by Li, et al., aim to dynamically determine spin loops in hardware without any software modifications. But while the Li, et al., technique is sufficient for simple spin loops, it fails to capture the lack of forward progress occurring in the complex Solaris 9 idle loop, or in the adaptive mutex locks in Solaris 9, both of which perform loop counting and other accounting functions while spinning.

4.5.3 Comparison of Techniques

This section provides a comparison of the Safe-Points technique for avoiding preemption of a kernel lock holder [Uhlig et al., 2004], and the Li et al. [2006] spin detection hardware, with the SDB proposed in this chapter.

4.5.3.1 Implementing Safe Points

Though conceptually appealing, implementing an efficient version of Safe-Points is not straightforward. To avoid starvation of other VMs, Uhlig et al. [2004] suggest using a 1ms *grace period*, after which point a VCPU will be preempted even if it has not reached a safe point. When the number of VCPUs exceeds the number of cores, this grace period is also necessary to avoid deadlock in a single VM.

In Solaris, which frequently sends software interrupts to implement cross-calls, it is critical to schedule the interrupted VCPU as soon as possible to ensure forward progress of the sender. But when several running VCPUs are either sending cross-calls or attempting to acquire a lock held by one of the senders, most executing VCPUs start spinning in the kernel (and are thus considered unsafe to preempt). Often no core is available to run the interrupted VCPUs until the grace periods for these spinning VCPUs expire.

There are a range of possible strategies for dealing with this problem, including reducing the grace period when paused VCPUs have outstanding interrupts, or delaying scheduling of the interrupted VCPU until it can be scheduled on its currently assigned core (to maintain affinity). We find that the best overall policy is to maintain the 1ms grace period, but run the interrupted VCPU on the next available core, regardless of cache affinity. While this results in frequent VCPU migrations, it greatly reduces the synchronization overhead compared to the other alternatives.

Uhlig, et al., do not observe this seemingly pathological cross-call behavior with their evaluation methodology, since 1) this mode of synchronization is fairly uncommon in Linux, and 2) they use their technique to enable more flexible scheduling (similar to Chapter 6), but do not actually overcommit a single VM, and can thus concurrently schedule all VCPUs of a given guest VM when necessary.

To determine safe preemption points while inside the OS, Uhlig et al. [2004] also investigate injecting additional safe points into OS execution by installing a fake device driver and sending interrupts from the VMM to this driver. Since they assume the OS is not holding a lock while executing this driver, they can safely preempt the VCPU at that point. Our evaluation does not implement this additional complexity, which would not prevent the cross-call problem for occurring anyway.

4.5.3.2 Implementing Li, et al., Spin Detection

The Li et al. [2006] spin detection hinges on two conditions: 1) “The observable state of the thread [VCPU] for the period between t_a and t_b is the same at t_a and t_b,” and 2) “Any change made by the thread [VCPU] to its observable state between t_a and t_b is not observed by any thread [VCPU] outside processor [core] P.” Together, these conditions guarantee that no forward progress has been made, while allowing *temporally silent* [Lepak and Lipasti, 2002] writes to registers or memory.

To detect these cases, Li et al. [2006] suggest using hardware to inspect program state at a *backward control transfer* (BCT) point. A BCT must exist in a dynamically executing spin loop. This technique ensures that the register state is the same at one execution of the BCT as it is at the next execution (i.e., one loop iteration), and that only silent stores have been observed within this time. They recognize that nested spin loops can also be a problem, where one inner loop inside an outer spin loop may change state, only to be changed back before the end of the outer loop. To tolerate nested loops, they propose keeping a table to track multiple BCTs, and the register state present the last time each branch was executed. They claim 16 entries is sufficient in the table, though the evaluation presented below optimistically assumes an infinite-sized table.

4.5.3.3 Comparison Results

Figure 4.8 presents the relative instruction count with different policies for managing synchronization overheads. From left to right, the bars for each benchmark represent the SDB scheme with a base timeslice of 33μs, the Safe-Points scheme with a 33μs timeslice and a 1ms grace period, and the Li, et al. spin detection with a 33μs timeslice. Results are normalized to SDB. For the Linux benchmarks, Safe-Points is quite effective. In fact, it outperforms the SDB scheme for `pmake` on Linux since it incurs less frequent context switching. For the Solaris benchmarks, however, this scheme is often unable to find preemptible VCPUs during concurrent cross-calls (as

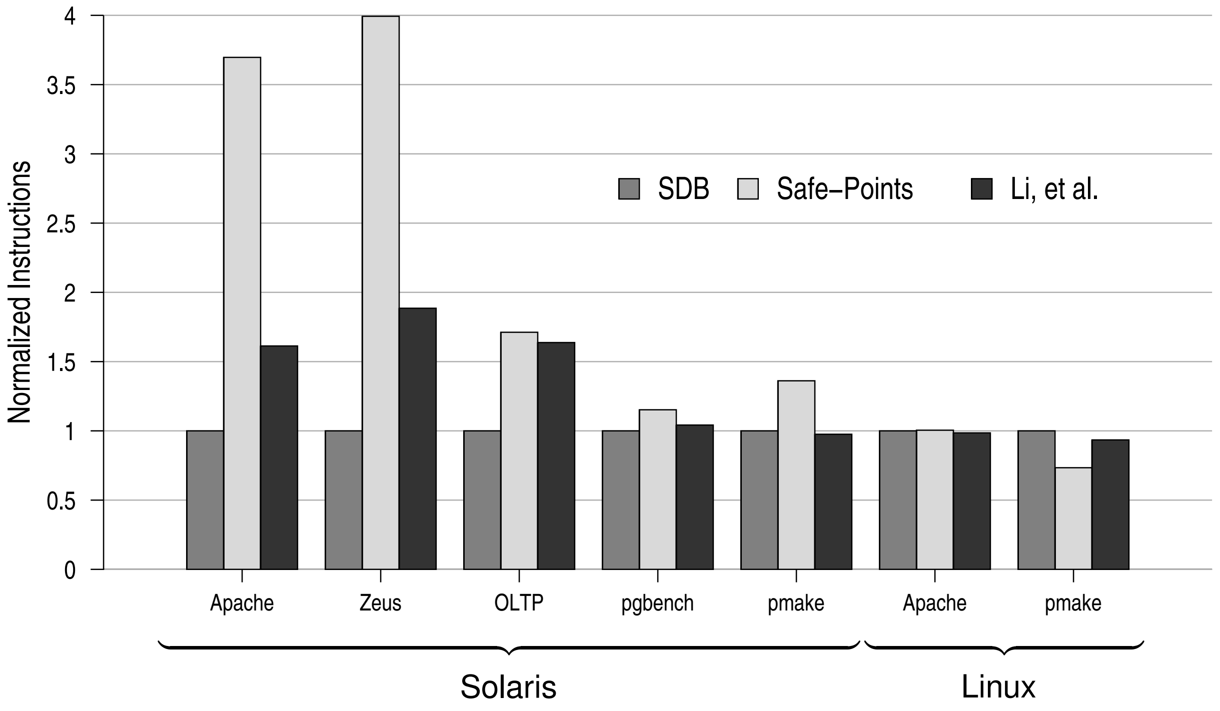


Figure 4.8 Comparison of Normalized Instructions for Related Work

discussed in Section 4.5.3.1), leading to a dramatic increase in spinning. The Li, et al., hardware spin detection is quite effective for several benchmarks. Their mechanism is able to detect the simple spin loops present in Linux, and in Solaris while waiting for cross calls, and thus does not see the same dramatic increase as the Safe-Points scheme for Solaris benchmarks. However, this mechanism is unable to detect spins in the more complicated Solaris 9 mutex lock or idle loop. For Solaris benchmarks with significant kernel lock contention, such as `Apache`, `Zeus`, and `OLTP`, it still incurs 50–80% more overhead than the SDB. The Li, et al., spin detection is said to capture these mutex locks in Solaris 8, the platform on which their paper was based [Li et al., 2006].

The relative runtime of these three configuration is shown in Figure 4.9. These results track very closely with committed instructions from Figure 4.8. The notable exception is the Safe-Points scheme for `Apache` on Linux. Although this scheme is particularly effective at eliminating

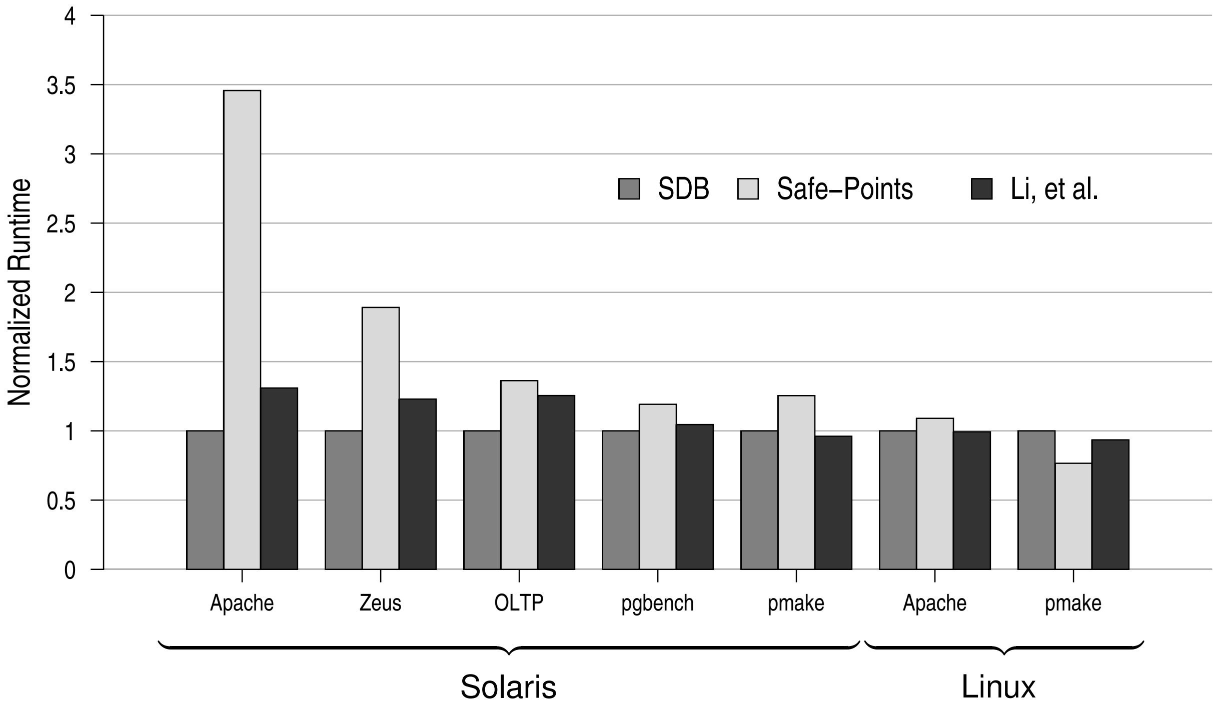


Figure 4.9 Comparison of Runtime for Related Work

spins, doing so requires VCPUs to frequently migrate among cores, hurting cache locality, and increasing runtime by 10%.

4.6 Chapter Summary

Many of the emerging challenges and opportunities of dynamic heterogeneity result in a chip where the number of cores that are available or appropriate for a particular type of computation are fewer than the number of VCPUs that a single OS (or guest VM) wishes to concurrently execute. Such a system is referred to as an *overcommitted* system.

Allowing VCPUs to move among cores in such a system requires techniques to virtualize the cores. This chapter proposes a technique to properly maintain and migrate the state, including registers, TLBs, and interrupts, associated with each VCPU, using limited hardware support. This technique was shown to be sufficiently low-overhead for relatively frequent VCPU transitions.

Overcommitting a single OS requires further support, since the OS incurs large synchronization overheads due to mutex locks and cross-calls when not all of its VCPUs are concurrently executing. This chapter proposed the *Spin Detection Buffer* (SDB), a hardware mechanism for heuristically identifying a spinning VCPU from its dynamic instruction stream. Combined with the other virtualization techniques, the SDB nearly eliminates the overhead of this synchronization. The SDB also outperforms two other techniques for mitigating the synchronization overheads.

Overall, the proposed SDB technique works well for virtualized environments as it automatically detects other cases where a VCPU is not doing useful work, such as the OS idle loop and spins in user code. Together, these techniques for multicore virtualization enable a multitude of applications for adapting to dynamic heterogeneity.

Chapter 5

Adapting to Intermittent Faults

Although transient and permanent hardware faults have been studied by a multitude of architecture researchers, few if any architects have previously investigated *intermittent* faults, an “in-between” class of faults that is likely to gain importance as technology continues to scale [Borkar, 2004; Borkar et al., 2003; Constantinescu, 2007, 2003]. Intermittent faults arise from a combination of physical variation (e.g., manufacturing variation or in-progress wear-out), and variations in the operating conditions (e.g., temperature or voltage). The result of this variation are bursty faults that occur for a period of time, and can then disappear as operating conditions change.

Intermittent faults exhibit some of the worst properties of both transient and permanent faults, in that they cannot be relied upon to either go away, as required by most techniques to tolerate transient faults, or to consistently stay, as required by several techniques to tolerate permanent faults. In this way, intermittent fault present new challenges to architects and system designers — not the least of which is the creation of rapidly changing dynamic heterogeneity in terms of different cores’ abilities to perform reliable computation.

Despite these challenges, several hardware schemes appear capable (with minor modifications) of detecting and recovering from a variety of intermittent faults (e.g., [Ernst et al., 2003; Gomaa et al., 2003; Hamilton and Orailoglu, 1998; Ismaeel and Bhatnagar, 1997; LaFrieda et al., 2007; Smolens et al., 2006; Weaver and Austin, 2001]). However, applying such techniques in isolation, without consideration for the nature and causes of intermittent faults, will require these schemes to be so complex and have such high overhead that they will be neither practical nor desirable in many cases.

Instead, this chapter proposes techniques to *adapt* to the effects of intermittent faults in a way that makes existing low-overhead detection and recovery techniques more practical and effective. In particular, this work argues that the ability to temporarily suspend program execution on a core that is sustaining intermittent faults will be an effective ploy for 1) reducing several of the factors contributing to the faults in the first place, 2) simplifying system design by reducing the fault coverage requirements, and 3) aiding in the diagnosis of permanent circuit damage. In this way, a core can simply be avoided when its capacity to correctly execute code becomes temporarily limited by a burst of intermittent faults.

Naively suspending a processing core is not a common practice because it is not transparent to software and can lead to serious system-level consequences. Fortunately, multicore processors provide unique opportunities for enabling techniques to adapt to the dynamically changing resource availability created by intermittent faults. A qualitative and quantitative comparison of three of the most logical adaptation techniques is presented, which are, 1) pausing execution on the faulty core without notifying the OS, 2) using spare cores, and 3) asking the OS to stop using the faulty core. Each of these techniques, however, is shown to be deficient with respect to one or more system design goals. To remedy several drawbacks of these three, this chapter propose a fourth technique: utilizing the multicore virtualization techniques developed in Chapter 4 to manage an *overcommitted system* during periods of intermittent faults.

5.1 Intermittent Faults: An Emerging Challenge for Architects

Intermittent faults, as their name implies, occur in bursts for a period of time, but can then seemingly disappear for a long period of time as well. They arise from a combination of *physical* variation, and variation in the *operating conditions*. These two factors are discussed in more detail below.

Physical Variation Manufacturing parameter variation, or in-progress wear-out, can give rise to *physical variation*.

Manufacturing variation is caused by the limitations of photo-lithography. As wires and devices become smaller, extremely small variation in the width of a gate or wire can constitute a significant fraction of the overall width of the device, creating a variation in the physical attributes (e.g., capacitance or resistance) [Borkar et al., 2003].

As devices age, they can experience wear-out effects such as electro-migration or gate oxide breakdown, which progress over the course of days to months [Smolens et al., 2007]. Eventually, wear-out can lead to a complete short or break in a wire, or an inoperable gate, becoming a permanent fault [Constantinescu, 2003]. *During* the progression of the wear-out, however, the device will again incur variation in its physical attributes.

These physical attribute variations eat into the voltage and timing margins that are built into the design. However, these margins must also become smaller as we scale technology [Eisele et al., 1996; Najm and Menezes, 2004]. Thus, these physical variation makes a design more susceptible to smaller and smaller variation in the operating conditions. These variations can result in timing errors even when operating conditions are well within the specified noise margins.

Operating Condition Variation Operating conditions, such as voltage and temperature, vary over time. Higher temperatures impart more resistance on wires, causing circuits to operate more slowly. Similarly, reduced voltages also cause circuits to operate more slowly. If timing margins are nearly violated at nominal temperature and voltage levels, timing faults will occur with higher probability during periods of temperature and voltage fluctuations.

Different software phases, which can exercise different components, and different critical paths through the circuits within those components, can also be considered a variation in operating conditions which affects the occurrence of these timing faults.

Intermittent hardware faults are hardware timing errors which occur in bursts for a period of time, due to the combination of these physical and operating condition variations [Borkar, 2004; Borkar et al., 2003; Constantinescu, 2007, 2003].

Because intermittent faults are affected by a large number of factors, the duration of bursty faults can vary across a wide range of timescales. For example, voltage fluctuations are typically short-lived, on the order of several to hundreds of nanoseconds [Borkar et al., 2003; Joseph et al., 2003; Powell and Vijaykumar, 2004]. Temperature fluctuations alter a device's timing characteristics over millisecond to second time scales [Powell et al., 2004]. Different software phases, which can change on the order of 100ms to several seconds [Sherwood et al., 2003], can exercise different components of a core, activating different intermittent faults. Finally, as wear-out progresses over the course of days [Smolens et al., 2007], it can cause intermittent faults to become frequent enough to be classified as permanent [Constantinescu, 2003].

5.1.1 Underlying Assumptions

Many uncertainties remain regarding the occurrence of intermittent faults in future technologies. However, based on technology trends, this dissertation makes three primary assumptions regarding these faults. These assumptions and the insights that led to them are discussed below.

1) Bursty intermittent faults will occur frequently. While the exact rates of various faults are not certain for future processors, current technology trends clearly indicate that even the design of commodity processors will be greatly affected by these faults. Section 2.2.1 provided a discussion of fault rates in future processors for permanent and transient sources, but there has been no known circuit and manufacturing study on intermittent fault.

The fault rates examined in this chapter are in the range of 0.1% *Fault Duty-Cycle* and up, meaning each core is affected by a fault 0.1% of the time or more. Though much higher than current processors, study such high rates are important to study for several reasons. First, by studying and proposing solutions that remain effective at such rates, process researchers can begin to understand what frequencies of hardware faults can be tolerated by higher layers in the system. Second, it is unclear whether intermittent faults between multiple cores will be correlated, though our evaluations assume course-granularity failures are independent.[1] Experimenting with high rates make it likely that multiple cores will fail at the same time, approximating a period of correlated faults. Finally, while these rates are well beyond the expectation for current systems, they are *not* beyond the public considerations of industry technologists [Borkar, 2004].

2) Practical circuits cannot mask all intermittent faults. Intermittent faults can affect nearly every component of a core, from register files and instruction window entries, to the bypass network and control logic, as well as clock and power distribution networks. While many techniques for tolerating various faults have been proposed [Blough et al., 1999; Ernst et al., 2003; Hamilton and Orailoglu, 1998; Ismaeel and Bhatnagar, 1997; Liang and Brooks, 2006; Mitra et al., 2006a; Nanya and Goosen, 1989; Shyam et al., 2006; Smolens et al., 2007], the ways in which faults manifest are likely to increase as devices become more unreliable. This will lead to a continued increase in the complexity and overhead of the techniques to tolerate the faults. We believe circuit, and higher-level, techniques will thus be employed to reduce the frequency of intermittent faults, but *cost-effective* techniques are unlikely to completely eliminate these faults, or prevent their occurrence from being noticed by system or application software.

[1]This assumption is only relevant to Section 5.4.4.

3) Intermittent faults manifest as circuit timing errors. In order to frame the continued discussion and evaluation in this chapter, the fault model is restricted to intermittent faults which manifest as timing delays in logic circuits within the core.

As operating conditions change, timing delays can become larger, leading to a fault when the delay is large enough that the pipeline latch improperly latches the previous value. Some circuit techniques can be used to detect and recover from errors resulting from circuits which operate too slowly. For example, Razor [Austin et al., 2004; Ernst et al., 2003, 2004] or the self-tuning circuits from Kehl [1993] use one or more pipeline latches to recapture the circuit's output using a delayed clock. The delayed latch can detect cases where the circuit output is late at the first latch, but resolves to the correct output before the delayed latch latches. Other techniques, such as BISER [Mitra et al., 2005; Zhang et al., 2006] similarly use a second pipeline latch, and can be modified to delay the *inputs* to the second latch [Hill et al., 2008]. By delaying the inputs, changes in timing can also be detected (at the possible expense of a longer clock cycle). The challenge with all of these techniques is that the delay cannot be too long, or the circuit outputs may already change to reflect the inputs for the next clock. Thus, these techniques can detect and recover from many timing errors up to the point that they become too large. Prior to that point, some higher-level corrective adaptation must have taken place in order to continue to provide reliable operation. This chapter assumes that operating conditions do not change so rapidly that all affected circuits go from operating with no appreciable delay during one cycle, to being slower than is detectable by the delayed latch in the next cycle.

One additional important assumption about the target multicore is made: that the memory hierarchy is reliable. Most architectural redundancy work (e.g., [Bower et al., 2005; Gomaa et al., 2003; LaFrieda et al., 2007; Mukherjee et al., 2002; Reinhardt and Mukherjee, 2000; Rotenberg, 1999; Shyam et al., 2006; Smolens, 2008; Smolens et al., 2006; Weaver and Austin, 2001; Zhou, 2006]), likewise focuses reliability efforts on the core logic, and makes similar assumptions about reliable caches. Research attention must certainly be paid to cache reliability, but the above assumptions seem reasonable because cache data and tag arrays are very regular structures, easily protected with parity or ECC. While the cache logic controller is less regular than the arrays, compared to the logic of the core itself, cache logic is comprised of smaller, simpler circuits that are likely easier to protect by other means.

5.2 Adapting to Intermittent Faults

The inability of circuit-level techniques to detect and recover from timing errors that become too large creates the need for *adapting* to the effects of intermittent faults. This need gives rise to the further contributions of this chapter. In particular, this chapter examines suspending the use of a core as a means of preventing timing-related errors during periods of intermittent faults from affecting the execution of the user code.

5.2.1 Suspending the Use of a Core

Suspending the use of a core may not be able to repair manufacturing variation or in-progress wear-out. However, temporarily suspending computation on a core *will* cause temperature and voltages to stabilize, reducing the further occurrence of any intermittent faults caused by these two major factors.

Suspending the use of a core when a burst of faults begins, or is expected to occur, can also improve the overall reliability of the system. Due to the factors influencing intermittent faults, correlated faults *within* a core or other localized area are very likely. Thus, the kinds of events that are most challenging for existing techniques to protect against, e.g., multiple concurrent faults or faults affecting critical structures, are most likely to occur together during temperature, voltage, or other fluctuations. All reliability techniques have a certain probability of manifesting unprotected errors to higher levels in the system. By reducing the number of faults they must protect, especially the ones they are least likely to protect, the number of faults they miss is reduced.

Current high-availability systems already take a similar approach by having service technicians replacing chips when correctable intermittent faults begin to occur [Constantinescu, 2003]. However, the granularity of failure in a multicore (*portions* as opposed to an *entire* chip), and the increasing frequency of these faults even for commodity processors, make chip-level replacement undesirable.

5.2.2 Recovering from Intermittent Faults

Several adaptation mechanisms require the virtual processor (VCPU) previously executing on the suspended core be moved to a different core. Though recovering the VCPU state from a suspended core may be possible in certain circumstances (e.g., [Litt, 1998]), it is clearly infeasible for others. At other times, it may be possible to detect or predict a burst of errors with sufficient timeliness to write out the VCPU state error free. However, this work pessimistically assumes that the fault recovery mechanism periodically creates checkpoints, similar to SafetyNet [Sorin et al., 2002], ReViveI/O [Nakano et al., 2006], or the techniques used by Shyam et al. [2006] or Dynamic Core Coupling [LaFrieda et al., 2007]. The checkpoints are stored into the cache every 10k cycles, and on I/O, and are consistent across the on-chip cores [LaFrieda et al., 2007; Nakano et al., 2006; Sorin et al., 2002].

Further discussions and evaluations in this chapter assume the use of circuit-level techniques such as Razor for detecting and recovering from many simple faults, whereas upon detecting a rash of intermittent faults on a core, these circuit mechanisms initiate a rollback to the previous validated checkpoint and then begin the adaptation mechanisms.

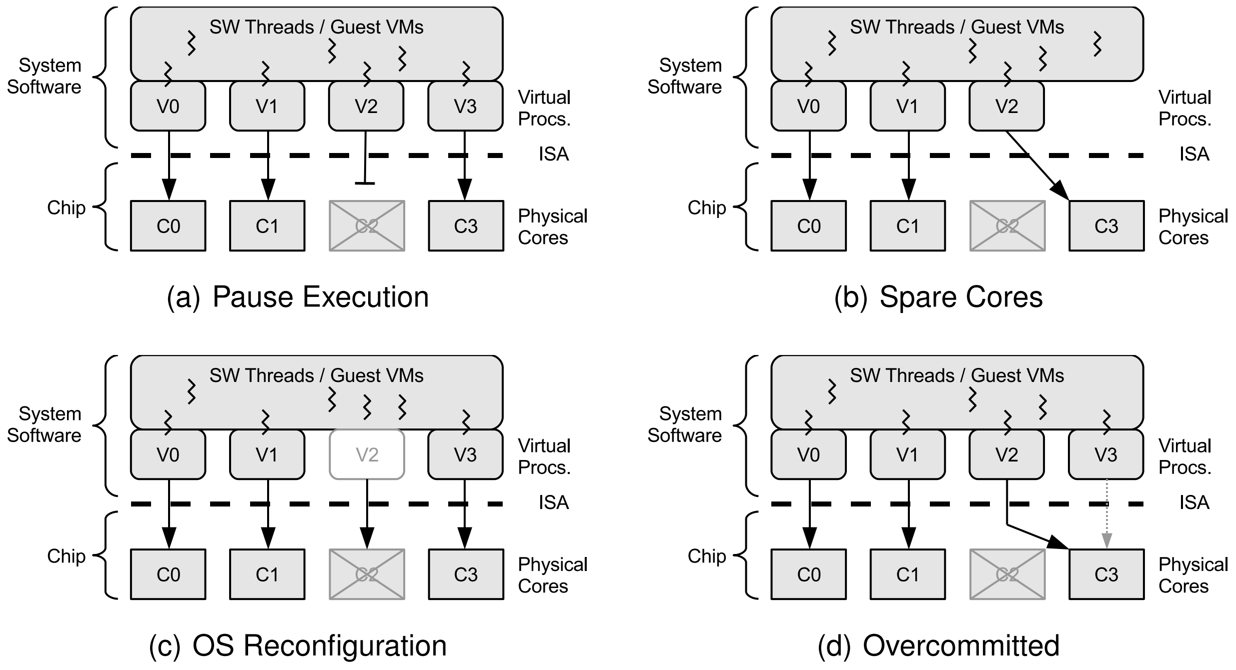


Figure 5.1 Core Suspension Techniques

5.3 Exploring Adaptation Techniques

Naively suspending program execution on a core is not a common practice because it is not transparent to software and can have serious system-level implications. Fortunately, multicore processors provide unique opportunities, including inherent redundancy, low on-chip latency, and high bandwidth, which enable several techniques for adapting to the temporary loss of one or more cores. This section discusses three such techniques which represent the current state-of-the-art, and proposes a fourth technique, which utilizes the multicore virtualization techniques from Chapter 4, to remedy serious drawbacks in each of the first three. Section 5.4 presents a quantitative comparison of all four techniques, considering throughput, effects on latency-critical applications, fairness, and overheads at different fault rates.

For simplicity, this chapters refers to the lowest software layer as the *operating system* (OS), which could be replaced with *hypervisor* throughout with no loss of generality.

5.3.1 Technique 1: Pause Execution

The first plausible technique for suspending the use of a core is to just pause the execution of instructions for a period of time. As shown in Figure 5.1(a), when a core (C2 in this case) sustains an intermittent fault, the microarchitecture pauses the execution of instructions from the VCPU assigned to that core (V2).

Pausing execution is the simplest technique examined in this chapter, and has been used, in a uniprocessor at least, for thermal management [Gunther et al., 2001]. In a multicore, other cores continue to execute instructions, thus pausing execution on one core will not drastically affect the other cores as long as they do not attempt to communicate with the thread assigned to the paused core. If communication is present, however, pausing one core can cause a cascading effect, livelocking other cores.

This technique is not *fair*, because threads scheduled on the paused virtual processor are starved, and it can similarly impact the latency of critical applications. Low throughput during a fault would be expected from this technique for workloads where threads frequently communicate, but for faults of short duration, this technique may be adequate for some applications.

5.3.2 Technique 2: Spare Cores

Unlike pausing, setting aside one or more cores as spares is expected to have little impact on software during a fault. For example, an eight-core chip might only expose seven cores to the OS. During a fault, the chip, using a hardware/firmware layer, can transparently remap the affected virtual processor from the faulty core to the spare. Core sparing is depicted in Figure 5.1(b).

Hot (powered up) spares are appropriate for short duration intermittent faults as circuit techniques can reduce leakage power when the core is not needed [Tschanz et al., 2003]. Since the performance degradation is negligible during a fault (as long as the number of faults do not exceed the number of spares), spare cores are also effective for long-duration or permanent faults. Partly for these reason, spares are used in real systems, at least for permanent faults [Bernick et al., 2005; Slegel et al., 1999].

The major drawback of setting aside spare cores, especially for commodity processors, is the high overhead of *not* using these cores during fault-free execution. In addition, this technique cannot tolerate more concurrent failures than the number of spares without an additional fall-back mechanism.

5.3.3 Technique 3: OS Reconfiguration

A third possible technique is to ask the OS (or hypervisor) to reconfigure itself to only use the remaining fault-free cores. This technique is depicted in Figure 5.1(c), where the de-configured virtual processor is not assigned any software threads or guest virtual machines to run. Some current OSs (such as Solaris) and hypervisors (such as those that run on the IBM zSeries) already contain this functionality [Armstrong et al., 2005; Sun Microsystems, Inc.]. For other systems, this technique requires intrusive software modifications.

Software reconfiguration can take several milliseconds, and can cause high overheads for faults of short duration. But the performance of the system should gracefully degrade once reconfiguration is complete, since the OS retains responsibility for scheduling threads, maintaining fairness, and achieving low latency for critical applications.

On the surface, this technique also appears to eliminate the need for hardware adaptation mechanisms. Unfortunately, that is not the case, since current operating systems requires the faulty core, and all other cores, to operate correctly until reconfiguration is complete [Litt, 1998; Sun Microsystems, Inc.]. For the evaluations in Section 5.4, the proposed overcommitted technique (discussed next) is utilized during reconfiguration, though it is not needed once reconfiguration has taken place.

5.3.4 Proposed Technique: Utilizing an Overcommitted System

A qualitative look at three existing techniques for suspending the use of a core has revealed several deficiencies: fairness and latency concerns, along with the possibility of cascading livelock; high fault-free overhead and the need for a fall-back mechanism; and OS-intrusive modifications plus the need for advanced notice of an upcoming fault.

To alleviate these drawbacks, this section proposes a fourth technique which uses a thin hardware/firmware layer to abstract the details of fault management from the system software, while presenting a view of continuous, fully-functional, reliable operation. Such abstraction is achieved during periods of intermittent faults by building on the multicore virtualization proposed in Chapter 4. Since this technique operates beneath the ISA, it is applicable to all system software that can be loaded on the machine.

VCPU Scheduling Applying the multicore virtualization techniques to intermittent faults allows the hardware/firmware layer to manage the mapping of VCPUs to cores, such that a given VCPU, unbeknownst to the system software, can be quickly migrated to a different physical core, or briefly paused.

Two (or more) OS-visible VCPUs must share a single physical core during a fault. Figure 5.1(d) shows an example of using this technique, with virtual processors V2 and V3 sharing core C3. V2 is currently executing, while V3 is not. Utilizing spin detection hardware allows the two VCPUs to be frequently switched as necessary to avoid the issues associated with the generic *Pause* technique.

Using an overcommitted system to preserve the illusion of continuous, reliable operation of all VCPUs can result in performance asymmetry when two or more VCPUs share a core, while other VCPUs execute alone on other cores [Balakrishnan et al., 2005]. To achieve continued symmetry and fairness during a fault, VCPUs that are co-assigned are rotated. For example, at some point, V2 may have C3 to itself while V3 and V0 share C0. Later, V0 will rotate and share C1 with V1. For the 8-core experiments in Section 5.4, VCPUs are rotated in this manner frequently enough such that symmetry is maintained over a 5ms window, which is slowly enough that cache affinity is largely maintained.

Hardware Complexity As described in Section 4.3, the proposed overcommitted technique involve modest hardware/firmware complexity. Required features involve a mechanism to context switch a virtual processor, a VCPU to core mapping table, spin detection hardware, and control logic. Overall, this is a modest amount of complexity, though certainly more than is required for the *Pause* technique. However, all of these components except spin detection are already required in order to use spare cores.

ISA Transparency By placing control over the use of faulty and non-faulty cores below the ISA, the abstraction of fault-free operation occurs transparently to both the OS and a traditional hypervisor such as VMware or IBM's Power5 Hypervisor. Such a model allows chip manufacturers to ship a chip that is expected to experience intermittent faults, but will continue to operate correctly regardless of the system software installed on the machine. The model provides several benefits for chip makers: first, the burden of correct hardware operation remains with the hardware vendor, not the system software vendor; second, the new chip automatically works with products from multiple system software vendors, and with legacy system software as well; and finally, as shown in Section 5.4, placing control of faulty cores beneath the ISA allows some of the functionality to be implemented in hardware, making it easier to quickly adapt to frequent changes in hardware configuration.

5.4 Evaluation

This section focuses on two kinds of experiments. First, system behavior is inspected in detail during a fault by measuring throughput, latency and fairness. After understanding these implications, the overall impact of these faults, including fault-free execution, for a wide range of fault durations and frequencies is inspected using both a simple mathematical model and execution-driven simulation. An overview of the experiments and methodology is described below.

5.4.1 Experimental Overview

Experiments were performed on an 8-core system, which exposes 8 VCPUs to the OS in most experiments. A mix of commercial and other workloads running Solaris 9 were evaluated. Details of the target system and workloads are provided in Chapter 3.

5.4.1.1 Methodology

In all simulations, we pause all cores for 10k cycles (3.3μsec) upon initiation of fault recovery to roll back to the latest verified checkpoint and account for the work lost.

Evaluating the *Pause* technique is straightforward. The virtualization of the *Overcommitted* technique is described in Section 4.4.1. The methodology for experimenting with *OS Reconfiguration* and the *Pause* technique are described below.

OS Reconfiguration To perform OS adaptation, Solaris is instructed to *unconfigure* one of its eight virtual processors, CPU4. In simulation, this operation starts similarly to software generating an interrupt. The timing simulator sets up a fake memory operation on CPU1, which is passed to the appropriate ASI handler within the functional simulator. Included in the interrupt packet is the PC of the interrupt handler on the destination CPU. The functional simulator then delivers the interrupt to the appropriate (virtual) CPU (in this case, a third processor, CPU3). The arriving interrupt causes control-flow on CPU3 to vector into the trap table, which sets up the processor state for OS execution, and then jumps to the specified interrupt handler included in the interrupt packet.

The interrupt handler is specified to be a small, otherwise blank portion of OS memory in which a short, hand-assembled code snippet has been placed by the timing simulator. The code segment simply acts as a launching point for calling arbitrary kernel functions as directed by the timing simulator. When the timing simulator detects that control flow has entered this launching point, it copies over the desired call parameters (registers and memory) and target PC, and continues simulation. Solaris then calls the desired function, which executes as normal. Upon finishing, Solaris simply returns to the launching point function, then the trap table, and finally back to the user or OS code that was executing before the interrupt took place.

The simulator forces the launching point function to call `sbd_ioctl()` with the necessary arguments to unconfigure CPU4. The function and arguments are the same as would be called by the command `cfgadm -C unconfigure CPU4`, but the interrupt mechanism creates the ability to call this function at arbitrary points without the overhead of the command. The Solaris `psradm` command, which can take CPU4 'off-line' is insufficient, as the processor is still required to process cross-calls. We use the overcommitted technique as the fall-back mechanism until the virtual processor is unconfigured.

Additional changes had to be made to the default Simics configuration to enable `cfgadm` to unconfigure a CPU. In particular, an on-board, CPU SRAM structure was added to the configuration. This SRAM is used in a real Sun system to allow an unconfigured CPU to execute code and spin in a special idle loop without creating any bus traffic. Before unconfiguring the CPU, the kernel copies a shutdown function onto this SRAM, and then jumps to the function.

After repeated attempts, further limitations in the Simics functional model could not be overcome to enable these experiments to be performed with newer versions of Solaris, or to perform the analogous experiment of reconfiguring a CPU. Posts to the Simics forum solicited no useful response from Virtutech.

Spare Cores Due to the methodology used, comparing runs using separate commercial workload checkpoints with different numbers of OS-visible VCPUs is impractical. The reason is due to the complex, variable nature of the commercial workloads, and the fact that only a small portion of the each workload's execution can be modeled in detail. In particular, there is little way to ensure that two workloads, set up and warmed up on separate machine configurations, are executing at nearly the same point, or that the OS disk caches have similar contents. These differences can impact performance by as much or more than the difference between seven and eight cores.

For the throughput experiments, the *Spare Cores* scheme is instead modeled using the overcommitted technique with oracle spin detection and without charging overhead for storing, migrating, or switching VCPU state. A seven-processor system *is* properly simulated with the microbenchmark for latency and fairness experiments, since the microbenchmark (discussed below) is very regular and is dominated by user code.

5.4.1.2 Experiments

Measuring Throughput For these experiments, one core experiences a detected fault at the beginning of execution; simulations are then run for a range of times from 100μs to 1 second.

Measuring Latency and Fairness For both the latency and fairness experiments in Section 5.4.3, a single 10ms fault is simulated beginning at 100μs of simulation, and then run for 11ms. A microbenchmark is used for these experiments, consisting of one thread per processor, where threads each execute short CPU-bound transactions and have no communication. Threads call `thr_yield()` every transaction to improve the OS's ability to maintain fairness, especially for the OS reconfiguration experiments. The *spare cores* experiments have seven threads and seven VCPUs, with eight threads and eight VCPUs for the rest.

Measuring Overall Performance Impact Determining the overall impact of intermittent faults requires accounting for periods of fault-free execution as well. Several experiments are run with faults randomly occurring at a particular rate. The fault duration is fixed in each experiment, but the inter-arrival time of faults is sampled, independently for each core, from a normal distribution of moderate variance.

Tracing Faults In order to better understand the results of other experiments, traces are take of the number of cores performing *useful work* during a fault of 100ms. *Useful work* is defined, for each processor, as whether or not any user instructions were executed in each 100μs period, and sum this Boolean value over all eight OS-visible processors. These traces are taken for the *Pause* an *OS Reconfiguration* techniques.

5.4.2 Throughput During a Fault

This section demonstrates the throughput of all four techniques *during* intermittent faults of various durations. Each technique is discussed in turn. The line at $\frac{7}{8}$ in throughput graphs represents the expected best-case slowdown of losing one core.

Pause Execution Figure 5.2 shows the throughput of each benchmark when pausing execution for faults of various durations. For the shorter 100μs and 1ms faults, all workloads observe throughput within 25% of the best case. Across a range of fault durations, `vortex` continues to have throughput similar to the best case, while `artOMP` is only slightly lower than that. The commercial workloads, on the other hand, which have significant OS activity and communication

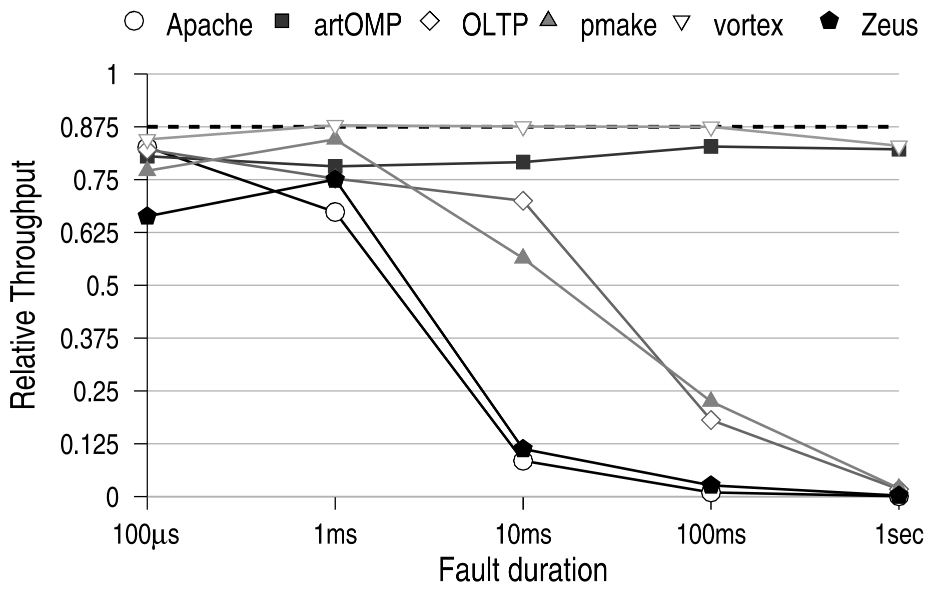


Figure 5.2 Throughput of Pausing Execution During a Fault

between cores, observe much lower throughput for faults of duration greater than 1ms — even approaching *zero* throughput for 100ms and 1sec faults.

Figure 5.3 helps explain this throughput loss for longer faults. This figure shows the first portion of a trace of the number of cores performing useful work during every 0.3ms of a 100ms fault. For all workloads, the number of cores performing useful work immediately drops to seven (or lower) after the fault. `vortex`, with eight independent processes, remains at seven for the duration of the fault. For `artOMP`, a second core stops performing work after 2ms because it has blocked waiting on a TLB shootdown request sent to the VCPU formerly executing on the paused core.

The other four workloads, however, have much more frequent interaction among cores, causing rapid degradation of the entire system's forward progress. For `Apache` and `Zeus`, nearly half of the VCPUs in the system stop making forward progress within 1ms of the fault. For the three commercial workloads in this graph, all VCPUs stop making forward within 3–11ms. The fault-free cores are simply executing OS spin loops waiting for either cross calls to complete, or

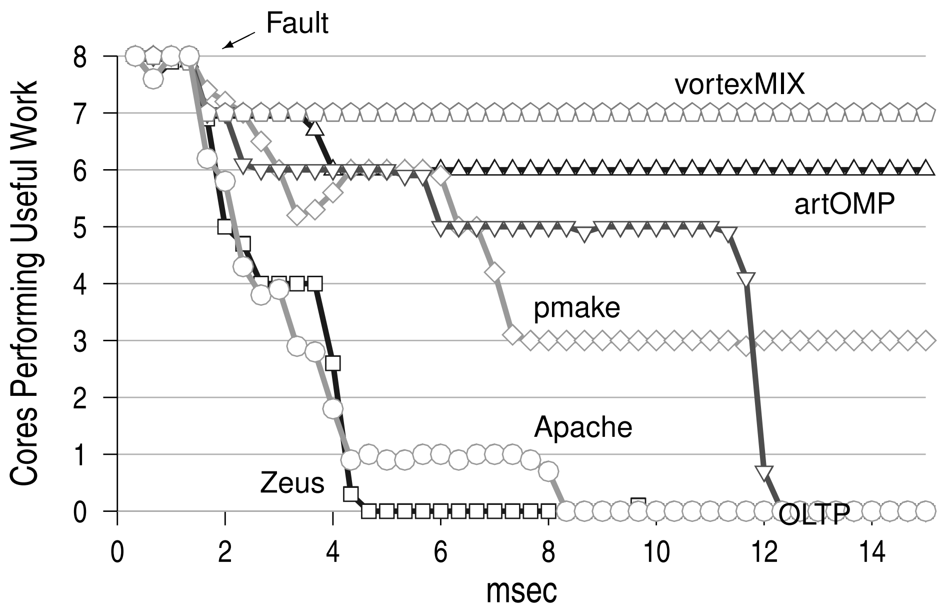


Figure 5.3 Cascading Livelock of Pause Scheme

locks held by the faulting processor. While not shown in the graph, all cores will eventually resume useful work after the paused core is re-enabled, provided the paused interval is short enough that the OS kernel does not panic (∼1 second for Solaris 9).

Despite its simplicity, Figures 5.2 and 5.3 show that the cascading livelock suffered by many workloads makes pausing execution unattractive for long faults. On the other hand, for short (<1ms) periods, this technique may be appropriate in some environments.

OS Reconfiguration To determine the performance of OS Reconfiguration, faults of various durations are again injected in the simulation. For these experiments, an interrupt is sent to the to the OS telling it to *unconfigure* the VCPU that was running on the core sustaining the fault.

The time required for reconfiguration to complete is shown in Table 5.1. During this time, the OS requires all eight VCPUs to continue to execute code, by either using a fall-back adaptation mechanism, or by continuing to execute code on the faulty core itself. In addition, this latency also represents the minimum length of time that overheads from reconfiguration will be incurred,

Apache	artOMP	OLTP	pmake	vortex	Zeus
2.12	1.69	2.12	2.40	1.89	4.09

Table 5.1 OS Reconfiguration Latency
OS Reconfiguration Latency (ms)

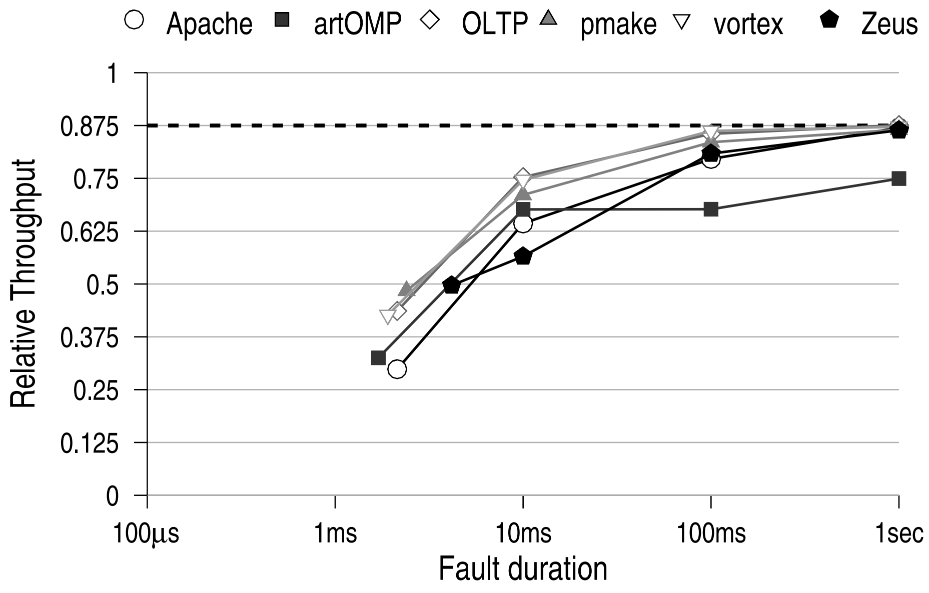


Figure 5.4 Throughput of OS Reconfiguration During a Fault

even if the suspended core is re-enabled in the meantime (since reconfiguration cannot simply be *stopped* once in progress).

Figure 5.4 shows the throughput of each benchmark when using *OS Reconfiguration* to continue execution during faults of various durations. The first point for each benchmark in this figure is placed on the x-axis (and measured against the baseline) at point in time that the VCPU is finally disabled. During the longer 100ms and 1sec fault durations, the cost of OS reconfiguration begins to amortize, and the throughput of all the workloads approaches the expected value of one less core compared to the baseline. For the shorter intervals, however, the cost of reconfiguration is not amortized — the loss in throughput is 2–6 times the loss expected from a single disabled core.

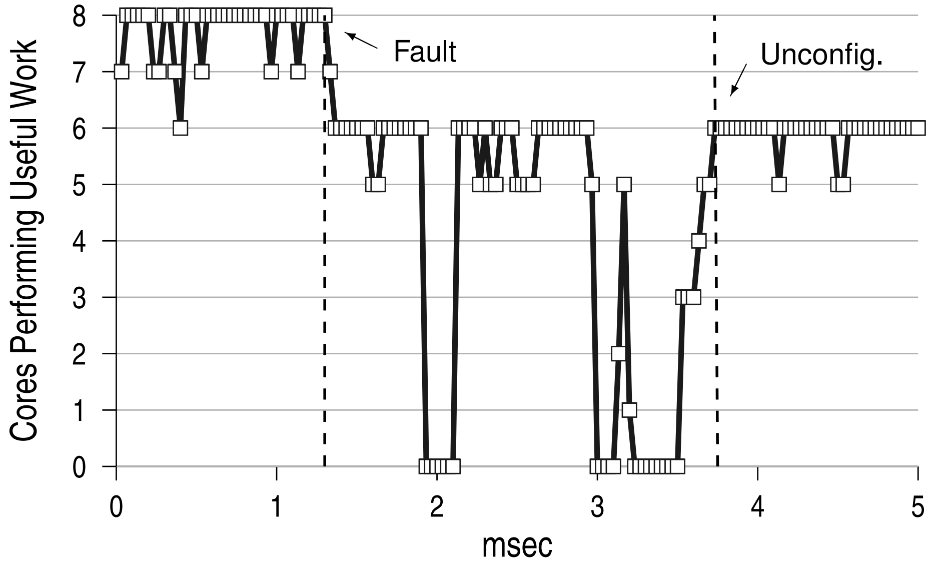


Figure 5.5 OS Reconfiguration of Zeus

Similar to Figure 5.3, Figure 5.5 explains this data by measuring useful work during various intervals. At 1.3ms (label 'Fault'), both the VCPU executing on the faulty core (Solaris's CPU4) and the recipient of the interrupt (Solaris's CPU3), stop committing user instructions. At 3.7ms (label 'Unconfig.'), CPU4 is finally unconfigured and enters a PROM idle loop. All processors in the system are quiesced twice to avoid deadlock arising from outstanding cross calls. The first is during the higher-level task of taking the CPU "off-line" [Sun Microsystems, Inc., 2008, line 1376], the second is during the lower-level task of actually "unconfiguring" the CPU [Sun Microsystems, Inc., 2006, line 424].

Note that the latencies in Table 5.1 are an average — the trace in Figure 5.5 took only 2.4ms.

Spare Cores Spare cores provide throughput during a fault that roughly matches the fault-free throughput (which also uses one less core than the baseline). Figure 5.6 demonstrates this fact. For the shortest fault, 100μs, the 10k cycles assumed for recovering from the fault introduces some overhead. Likewise, the process of transferring VCPU state and then incurring misses on

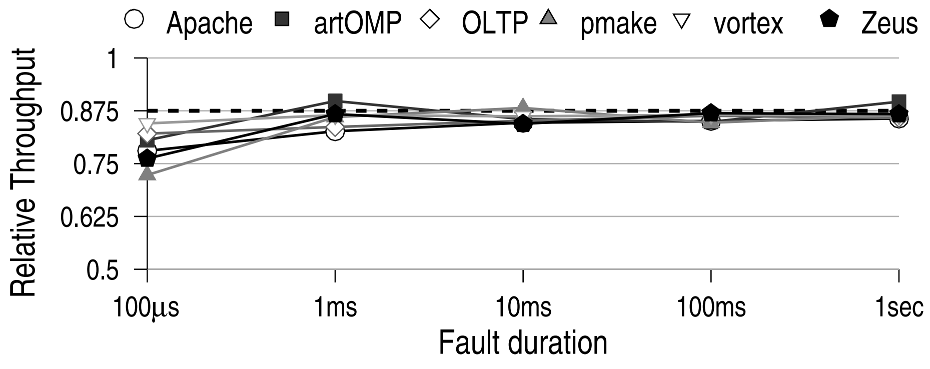


Figure 5.6 Throughput of Spare Cores During a Fault

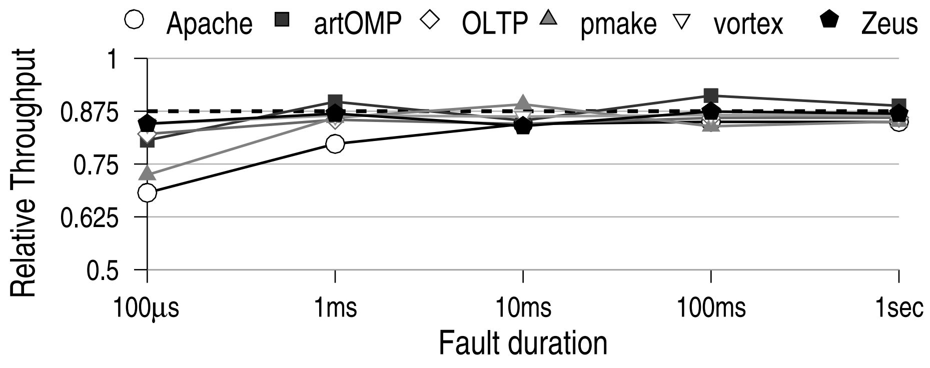


Figure 5.7 Throughput of an Overcommitted System During a Fault

all cached data causes additional initial overhead. For all the longer durations, however, there is practically no loss in throughput compared to the best expectation. `artOMP` appears to incur sub-linear slowdown for certain runs. This is an artifact of our methodology for simulating spare cores (see Section 5.4.1): a VCPU in the baseline system enters a spin loop waiting for all other VCPUs to acknowledge one of the aforementioned TLB shootdowns, causing our perfect spin detection to yield the core to a productive thread.

Overcommitted System Figure 5.7 demonstrates the high performance of the overcommitted system. Similar to using spare cores, this technique incurs some overhead for the shortest faults due to the recovery time and cache misses. However, this overhead is small and is quickly amortized for longer fault rates.

Using an overcommitted system with spin detection during periods of intermittent faults yields throughput similar to using spare cores. Unlike spare cores, however, this technique retains the ability to utilize the entire machine during periods of fault-free execution, and can handle concurrent failures.

5.4.3 Microbenchmarking Latency & Fairness

While throughput is important, other performance metrics are equally important for certain applications. For example, latency is critical for Multiplayer Online Games [Deen et al., 2006], or for telemetry applications, and fairness may be important for consolidated servers. Other metrics may be of interest as well. Ideally, we would use these target applications to measure transaction latency and fairness, but the complexity of building such workloads, combined with irregular or long transactions and the distorting effects of other software components, conspire to make such an evaluation difficult. Instead, a microbenchmark is constructed, as described in Section 5.4.1, to understand the underlying behavior of our four adaptation techniques. Experiments focus on a fault duration of 10ms, but results can be easily extrapolated to other fault durations.

5.4.3.1 Latency

Figure 5.8 shows the cumulative distribution of transaction latencies from each software thread for the microbenchmark. Both axes are logarithmic to highlight transactions that deviate from the common case.

In the baseline, fault-free system, 99.5% of transactions take 16μs or less, while several transactions take 40–100μs. Very similar data occurs when using a spare core, and when pausing execution, except that one transaction, the one on the paused core, takes over 10ms. Note that the

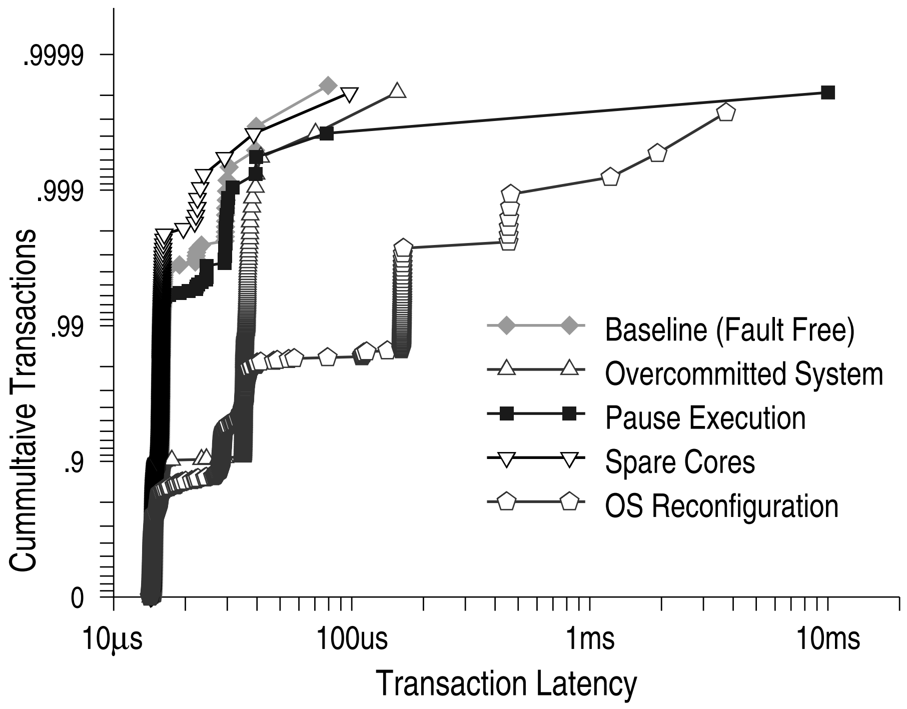


Figure 5.8 Microbenchmark Transaction Latencies

microbenchmark, dominated by user code with no communication, represents the best case for pausing execution. With OS reconfiguration, many transactions are delayed by 100μs–1ms, while the OS quiesces all VCPUs. Because the OS migrates threads off the faulty core, no transactions are delayed as long as the 10ms fault, but many outliers remain.

VCPUs that are sharing a core using the *Overcommitted* technique are allowed a maximum timeslice on the core before the the running VCPU is swapped for a paused one. As discussed in Section 4.2.1, this maximum timeslice is tunable by the system administrator. The length of this timeslice can potentially impact latency critical applications, but can be decreased if necessary for a small increase in switching overhead. For the experiments in this section the Virtualization Controller performs a VCPU context-switch at least every 20μs. Since transactions that are started but not finished before the timeslice is over will finish when that VCPU is rescheduled the next time. The SDB does not detect any spinning when running the microbenchmark, so each VCPU uses its maximum timeslice every time. Thus 12% of transactions take approximately 20μs longer than the baseline (since two VCPUs are vying for the same core). No outliers are delayed by more than 20μs longer than the baseline, however, unlike with the *Pause* and *OS Reconfiguration* techniques.

5.4.3.2 Fairness

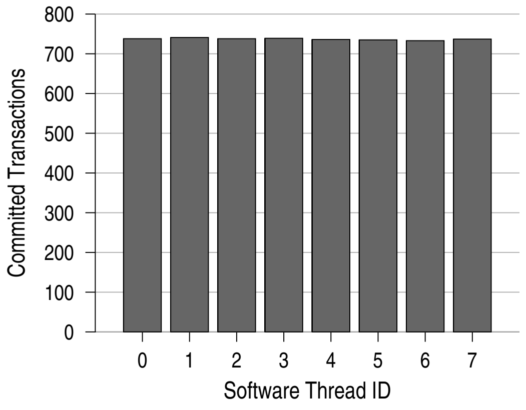
Using the microbenchmark, fairness is measured by examining the total number of transactions committed by each software thread. Figure 5.9 graphs this number for the baseline (fault free) and for each scheme. Again, a 10ms fault is simulated at the beginning of simulations that last for 11ms. Figure 5.9(a) shows that the baseline system commits a nearly equal number of transactions per thread. A system with a spare core also commits a nearly equal number of transactions per thread, though with one fewer thread (Figure 5.9(b)). Note that the graph for spare cores only has seven bars, while the others have eight. This result assumes that the application software can be easily partitioned seven ways, which is not the case for many scientific applications.

On the other hand, pausing execution causes one thread to be significantly impeded by the fault (Figure 5.9(c)). Since the OS is still scheduling software threads among all eight VCPUs, one application thread is starved when pausing.

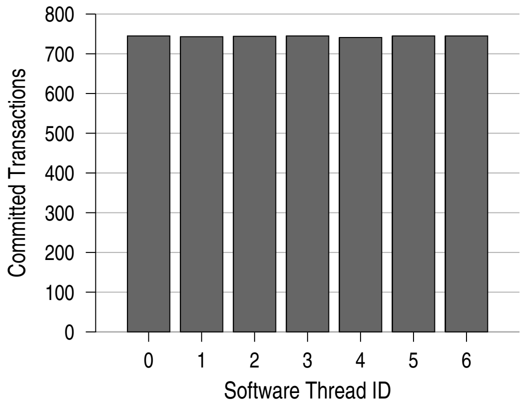
Due to the overhead of using at least one VCPU to orchestrate reconfiguration, and the quiescing of all VCPUs, OS reconfiguration cannot maintain fairness among software threads during the 11ms interval simulated (Figure 5.9(d)). The *OS Reconfiguration* scheme might fare better for longer fault durations.

The overcommitted system is able to provide conceptually similar fairness as spare cores and the baseline, even during the failure of one core (Figure 5.9(e)).

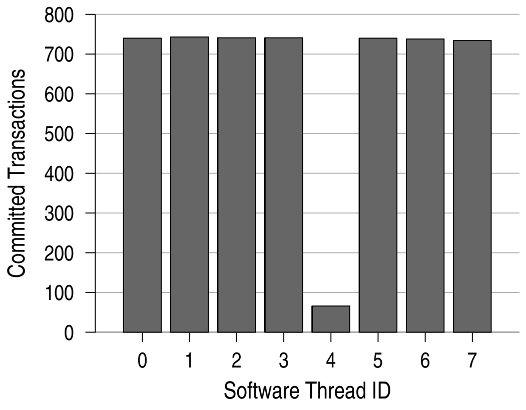
To quantify the degree of fairness, both the *fair speedup* (F.S.) metric used by Chang [2007], and the ΣM_0 metric from Kim et al. [2004] are examined. Results for the F.S. use the harmonic



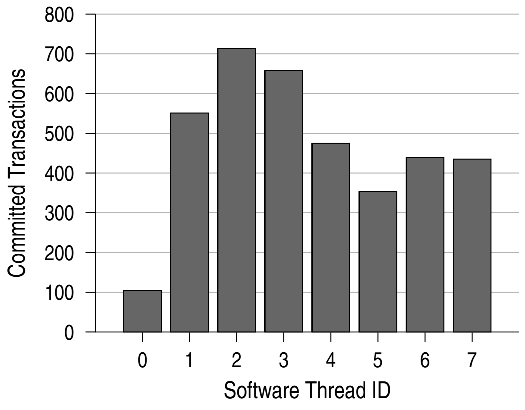
(a) Baseline



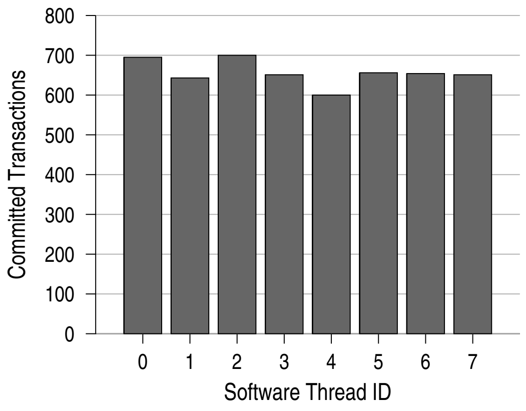
(b) Spare Cores



(c) Pause Execution



(d) OS Reconfiguration



(e) Overcommitted

Figure 5.9 Committed Microbenchmark Transactions from each Software Thread

	Base	**Spare**	**Pause**	**OS**	**OverC**
F.S. [Chang and Sohi, 2007] $\uparrow$	1.00	1.00	0.44	0.49	0.94
ΣM_0 [Kim et al., 2004] $\downarrow$	0.92	1.00	5.47	7.14	1.17

Table 5.2 Fairness of Metrics for Different Techniques

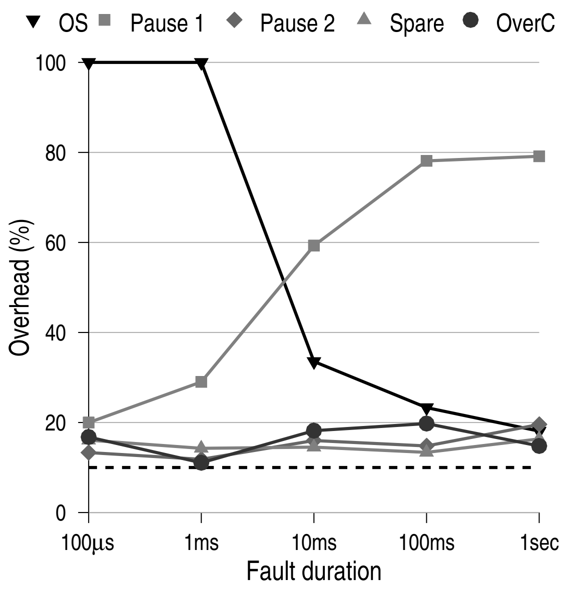
mean of the speedup between each software thread and the most productive thread in that experiment. ΣM_0 is derived from the sum of M_0 across all pairs of threads i, j, where $M_0^{ij} = \|X_i - X_j\|$, $X_i = \frac{Trans_i}{Trans_p}$, and p is the most productive thread.

For fair speedup, higher is better, and for ΣM_0, lower is better. These metrics are shown in Table 5.2. For both metrics, the baseline and spare cores are very close, while the overcommitted system is only slightly worse than both of them. Pausing and OS reconfiguration are significantly worse. Though the metrics differ in how much they penalize the OS and pausing schemes, both clearly show that these two techniques are inferior in terms of fairness.

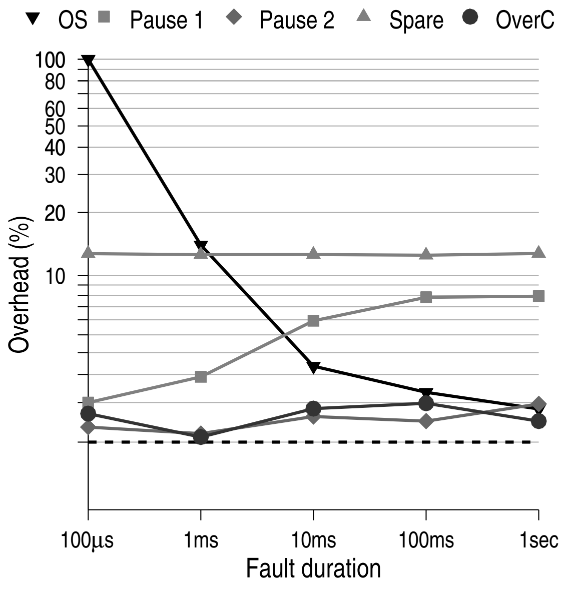
5.4.4 Overall Impact of Different Fault Rates

Results thus far have examined throughput during faults without considering intervening periods of fault-free execution. This section looks at the overall impact of the four techniques across a range of fault durations and frequencies.

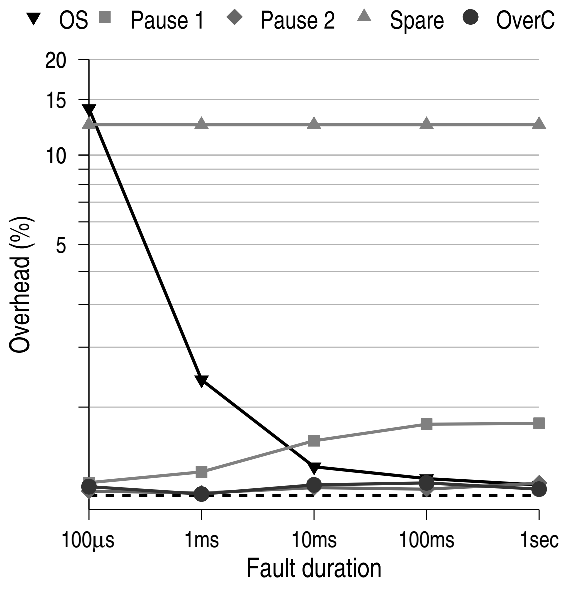
Using an analytic model, the throughput data from Section 5.4.2 is extrapolated to determine the overhead at various fault rates. The analytic model allows examination of the overheads in a more controlled environment, which is necessary because limitations in Simics prevent execution-driven simulation of OS reconfiguration, and limitations on the spare cores technique itself prevents it from handling more concurrent faults than spares (see Section 5.4.1). Execution-driven simulations using the other two techniques are used in the next section to validate the model and to explore multiple concurrent failures.



(a) 10% Duty Overhead



(b) 1% Duty Overhead



(c) 0.1% Duty Overhead

Figure 5.10 Overhead with Different Fault Duty-Cycles (Analytic Model)

5.4.4.1 Analytic Model

To get a basic understanding of the impact of different faults rates on overall performance, a simple analytic model was derived from data similar to the throughput data collected in Section 5.4.2. The experiments used for the model differ only in that the faulty core is re-enabled after the fault duration, and experiments are run for twice as long as that duration. For *OS Reconfiguration*, where Simics does not allow a core to be re-enabled, the model assumes that adding the faulty core back has 50% of the overhead of removing it.

To gauge overhead, the model examines a variety of *fault duty-cycles*, i.e. the expected fraction of time faults are affecting *each* core. For simplicity, we assume no concurrent faults.

The simple equation defining the overhead for each duty cycle d and fault duration l is as follows:

$$Overhead_{d,l} = (1.0 - AvgThroughput_l) \times d \times N \times 2$$

where $AvgThroughput_l$ is the throughput during a fault of duration l averaged together for all six benchmarks, and $N = 8$ is the number of cores. The reason the result is multiplied by 2 is that $AvgThroughput_l$ is for an experiment twice as long as the fault duration l. When averaging each benchmark, the *Pause* scheme is broken into two groups, *Pause 1*, containing the commercial workloads (i.e., `Apache`, `Zeus`, `OLTP`, and `pmake`) for which pausing works poorly, and *Pause 2*, containing `vortex` and `artOMP`, for which pausing works well. For *OS reconfiguration* with timeslices less than the reconfiguration latencies from Table 5.1, a throughput of zero is used. Data where the overhead is more than 100% are capped at 100%.

As an example calculation, the *Pause* technique for the group *Pause 1*, has nearly 0% throughput for a 1sec fault duration (Figure 5.2), and approximately 50% throughput for a 2 second run when the failed processor is reenabled after 1 second. For a 10% duty-cycle d, and $N = 8$, the overhead is thus $(1.0 - 0.5)(.10)(8) \times 2$, or 80%.

In Figure 5.10, each line in the graph keeps the *duty-cycle* constant. Thus, 100μs faults with a duty cycle of 1% are occuring, on average, every 10ms, and 1sec faults are occurring, on average every 100sec. Figure 5.10(a) represents a duty cycle of 10%, Figure 5.10(b) represents a duty cycle of 1%, Figure 5.10(c) represents 0.1%. Note that the y-axis is logarithmic in Figures 5.10(b) and 5.10(c). In all three graphs, a dashed line is drawn at the expected overhead represented by the appropriate fault duty cycle.

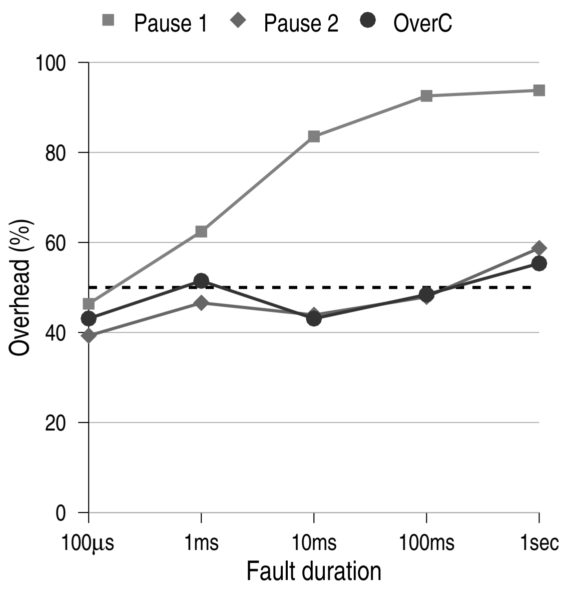
Because the model assumes no concurrent faults, *Spare Cores* incurs $\sim$12.5% overhead for 0.1–1% duty cycle, and slightly more for 10%. In all experiments, the group *Pause 2*, as well as the overcommitted scheme, incur overheads from 1–2 times the duty cycle. The same is true for the group *Pause 1* for 100μs faults, and for OS reconfiguration for 1sec faults. However, for longer faults, *Pause 2* observes overheads of approximately eight times the duty cycle, since a fault on each core affects the other eight as well. Similarly for OS reconfiguration, not only does a fault on one core affect the others, but the latency of reconfiguration causes overheads 2–3 orders of magnitude larger that the duty cycle for the shortest faults.

All techniques are expected to incur low overheads when fault rates are low, but even when fault rates create a duty cycle of 0.1%, care must be taken when invoking the *Pause* or *OS Reconfiguration* techniques.

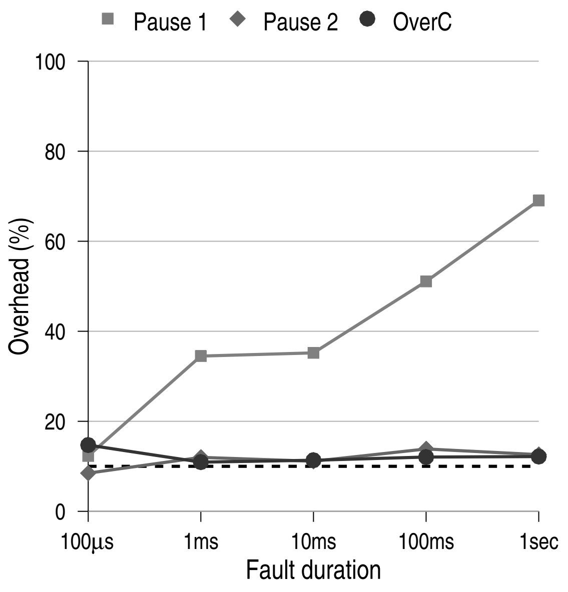
5.4.4.2 Execution-Driven Simulation

The simple analytic model in the previous section is unable to handle multiple concurrent failures. With higher fault duty-cycles, however, concurrent failures become much more likely. This section presents results of execution-driven simulation using randomly generated periods of intermittent faults.

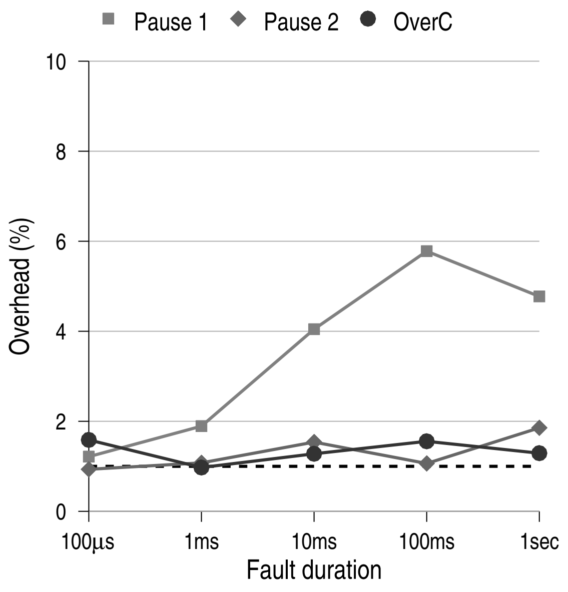
Figure 5.11 shows similar plots as 5.10, though results are directly driven by execution-based simulation. *OS reconfiguration* cannot be shown in this figure, since cores cannot be re-enabled



(a) 50% Duty Overhead



(b) 10% Duty Overhead



(c) 1% Duty Overhead

Figure 5.11 Overhead with Different Fault Duty-Cycles (Execution Driven Simulation)

in the simulation environment. *Spare Cores* is not show either, since this scheme itself is unable to tolerate more concurrent failures than the number of spares.

Figure 5.11(a) represents a duty cycle of 50%, Figure 5.11(b) represents a duty cycle of 10%, Figure 5.11(c) represents 1%. None of the y-axes are logarithmic. Again a dashed line is drawn at the expected overhead represented by the appropriate fault duty cycle in all three graphs.

A fault duty cycle of 50%, meaning that on average, half of the cores are experiencing a fault, is well beyond the expectation of systems in the near future. But a duty cycle this high demonstrates the scalability of overcommitting: for every fault duration, the *Overcommitted* technique is within 10% of the expected 50% overhead. The same is true for the infrequently communicating benchmarks in group *Pause 2*.

Confidence intervals were left off the graphs to improve clarity, however, variability in the 50% duty cycle experiments causes the 95% confidence intervals to reach $\pm15\%$ for some datapoints due not only to workload variability, but also variability in the randomly generated faults.[2] Thus, some data points appear lower than the expected duty cycle for *Pause 2* and *Overcommitted*, but both are near the duty cycle in all experiments. Confidence intervals for the 1% and 10% duty cycle experiments do not rise higher than $\pm8\%$.

For *Pause 1* benchmarks, a duty cycle of 50% creates overheads that rise to greater than 90% for longer fault durations. The reason is that little to no work is being performed for these benchmarks after a *single* core is paused for more than about 10ms (see Figure 5.2). For a duty cycle of 50%, the probability that no cores are faulty is $<1\%$. Despite the fact that at least one core is almost always paused, individual cores regularly come and go offline during simulation. This enables those cores to processes any pending interrupts and critical sections for which other cores

[2]Up to 24 trials were performed per benchmark per datapoint to reduce the impact of variability, but for a given sample variance, the confidence interval reaches an asymtote after $\sim$10 trials.

may be waiting, allowing some work to be performed. For shorter durations, *Pause 1* benchmarks are much closer to the expected overhead for a given duty cycle.

For longer faults, multiple concurrent failures actually benefit the pause scheme in comparison to the duty cycle, since other cores that might be affected by pausing one have some probability of already being paused themselves. This can be observed in Figure 5.11(b) by looking at the 100ms duration: *Pause 1* incurs a 55% overhead with simulation, but a projected 80% overhead from the analytic model. The reason for this discrepancy is that the analytic model treats the 10% duty cycle as though 8 faults (one per core) occur evenly spaced in an execution window 10 times as long as the fault duration. That is, in the analytic model, exactly one core is faulty exactly 80% of the time, or in other words, all cores are active only 20% of the time. In the execution driven experiments with a 10% duty cycle, all cores are active 43% of the time on average (0.90^8). A much smaller discrepancy is observed for the 1% fault duration, since both models predict all cores to be active ∼92% of the time ($0.99^8 \approx 1.0-(0.1\times8)$). The execution-drive experiments are more accurate for the higher fault rates, given the assumptions made about the fault distribution.

The benefits of using an overcommitted system become apparent from the experiments in this section. Even when half of the cores, on average, are faulty, this technique provides overheads commensurate with the duty cycle in all experiments. The same is not true for any of the other schemes.

5.5 Impact of Future Trends

Using spare cores becomes more viable as fault rates increase and the relative granularity of spares decreases. However, this technique still cannot easily adapt to long or short-term changes in the number of concurrent faults. For example, when using a laptop on an airplane, or when one section of a data center becomes too hot, fault rates may increase, requiring more spares. At other times, few if any spares may be necessary. Setting the number of spares too high introduces overhead, and setting it too low increases the probability of observing more concurrent faults than spares. An overcommitted system, on the other hand, has a distinct advantage since it can dynamically adapt to these changes.

Based on what we can assume about future multicores, we believe that the qualitative results of our experiments will generally hold. Future technologies will allow room for many more than eight cores, and this will undoubtedly have an impact on techniques for adapting to intermittent faults. If applications are partitioned so that they each use no more cores that they do in our simulations, we would expect the results for pausing execution to be similar. However, this technique could be devastating if a single application, with occasional communication, is using all cores of the chip. As long as all the cores are under the control of a single OS, or single hypervisor, the system software may still have to quiesce all cores to prevent deadlock, increasing the latency and overheads of software reconfiguration.

5.6 Related Work

Related work fall into several categories, from detection to adaptation and formal fault models. These are discussed below in turn.

5.6.1 Detecting Intermittent faults

Many circuit-level techniques for tolerating intermittent faults have been proposed [Blough et al., 1999; Hamilton and Orailoglu, 1998; Ismaeel and Bhatnagar, 1997; Nanya and Goosen, 1989], but they are generally applicable only to individual components. Consequently, they are likely to be useful for reducing the frequency, but not eliminating, intermittent faults. Similarly, thermal management techniques (e.g., [Brooks and Martonosi, 2001; Powell et al., 2004; Skadron et al., 2003]) can be used to reduce the frequency of faults by managing thermal variations. However, for future processors, avoiding intermittent faults with these techniques will require them to be overly conservative, thus providing low performance.

Shyam, et al., use Built-In Self Test (BIST) to detect, and course-grained checkpoints to recover from, permanent faults [Shyam et al., 2006]. Although fairly low cost, this approach cannot reliably detect intermittent faults for the same reason it makes no effort to detect transient faults: an intermittent (or transient) fault may appear during the computation, but not during the BIST phase.

FIRST [Smolens et al., 2007] detects wear-out-based intermittent faults by running a BIST at lower that normal noise margins. The adaptation mechanisms proposed in this chapter would allow these tests to be run without halting the entire system.

DIVA [Austin, 1999; Weaver and Austin, 2001] relies on a ‘checker’ processor to verify the output of the main processor before it commits instructions. Among DIVA’s advantages, it tolerate permanent, transient, and intermittent faults in the main processor, as well as design bugs. However, this proposal requires that a second processor be designed and verified, and that this second processor be fault free.

Application-level software [Reis et al., 2005a] and hybrid [Reis et al., 2005b] fault detection schemes typically assume limited fault models and have low fault coverage. It is also unclear how to apply these techniques to OS and hypervisor software.

Relaxing the Fault Model The fault model used for the evaluation assumes that intermittent faults will manifest as timing errors (Section 5.1). Relaxing this assumption may mean that circuit-level techniques such as Razor are insufficient, even when combined with the adaptation techniques presented in this chapter. The use of Dual-Modular Redundancy (DMR) as a detection and recovery mechanism might then be reasonable.

Many architecture papers consider using high-overhead Dual-Modular Redundancy for detection and recovery of particle-induced transient faults [Gomaa et al., 2003; LaFrieda et al., 2007; Mukherjee et al., 2002; Reinhardt and Mukherjee, 2000; Rotenberg, 1999; Smolens et al., 2006], however, transient faults do not require techniques for adaptation. Others architecture papers consider permanent faults [Bower et al., 2005; Govil et al., 2000; LaFrieda et al., 2007; Shyam et al., 2006], which requires permanent reconfiguration only. Weaver and Austin do mention intermittent faults [Weaver and Austin, 2001], but present no methods for adapting to them.

Although they do not discuss intermittent faults, several proposals, such as Reunion [Smolens et al., 2006], DCC [LaFrieda et al., 2007], and CRTR [Gomaa et al., 2003], execute two loosely-coupled threads on different cores. With minor modification, these techniques can thus be used to detect most intermittent and permanent faults, even with relaxed model assumptions, though they incur $\sim$2X overheads in terms of power and throughput. Several other techniques (e.g., [Reinhardt and Mukherjee, 2000; Rotenberg, 1999]) execute the checking thread on the same core as the original, and thus cannot detect most permanent or intermittent faults.

Since this work focuses on the effects these faults have on software, the results presented herein would remain unchanged if one considers DMR cores to form one logical processing core, and then performs the adaptation techniques on the logical core.

5.6.2 Reconfiguring after Device-level Faults

Several methods have been presented to continue use of a core despite permanent faults. These techniques involve fine-grained diagnosis and reconfiguration of a core's components [Bower et al., 2005; Shyam et al., 2006], or attempt to match a program's requirements and a core's capabilities, such as Core Salvage [Joseph, 2006]. We believe that the ability to suspend execution on a core in order to perform diagnosis and reconfiguration would likely be a simplifying addition to these techniques.

5.6.3 Alternate Adaptation Methods

There are other plausible techniques for adapting to the occurrence of bursty intermittent faults, without temporarily "losing" a core's ability to execute instructions. In particular, increasing voltages to speed the computations paths of a circuit, or lowering frequency to allow more time for a delayed path, would be logical to employ. However, going forward, fully suspending the use of a core is not necessarily a worse solution, especially after considering three main points. First, increasing voltages may actually worsen operating conditions such as temperature, creating a positive feedback loop, requiring even higher voltages. Second, reducing frequency is not transparent to software either, as it can create performance asymmetry [Balakrishnan et al., 2005]. Finally, as we enter an era where many more cores can be integrated onto the same sized die every few years, yet the number of cores that can be simultaneously powered does not increase to the same degree [Chakraborty et al., 2007], a handful of cores may be suspended at any time for with almost no loss in expected performance.

5.6.4 Fault Tolerance in Distributed Systems

Much distributed systems research has addressed fault tolerance for clusters of computers, e.g., [Bernick et al., 2005; Blough et al., 1992; Contant et al., 2004; Govil et al., 2000; Kalbarczyk et al., 1999; Lamport et al., 1982; McEvoy, 1981]. For most of this research, the unit of failure is an entire machine, including the CPU(s), memory, and system software. Such course-grained units are not applicable to systems comprised of only a few, or even one, multicore chip.

In addition, the comparatively short timescales of device-level intermittent faults render these software-based adaptation techniques ineffective because they cannot adapt quickly enough (see Section 5.3.3). For example, if certain cores on a chip observe intermittent faults every few seconds, software techniques will, by necessity, consider the entire chip to be permanently faulty.

Chameleon [Kalbarczyk et al., 1999] provides a reliable software-based fault tolerant system. They use the term *Adaptive Fault Tolerance* to describe a system that is flexible to the dynamic demands of applications, but not necessarily to the dynamic conditions of the hardware.

5.6.5 Formal Fault Models

Formal fault models are a useful tool for reliability researchers, as the can allow a researcher to prove their design covers all possible faults within the model (subject to certain assumptions). However, identifying or creating a good formal fault model is a tricky endeavor, walking the line between capturing all causes and manifestations of the faults under consideration, and being simple enough to allow effective reasoning. For example, the simplistic *Stuck-at* model and *Fail-stop* model [Schlichting and Schneider, 1983], are unable to capture the dynamic nature of intermittent faults. At the other extreme, the Byzantine Hardware Fault model [Nanya and Goosen, 1989], which places simplifying constraints on the Byzantine Generals Problem [Lamport et al., 1982], captures not only transient and permanent faults, but also the inconsistency of intermittent faults. However, this model is so overly general that provably correct solutions require at least triplicate redundancy [Lamport et al., 1982].[3]

The *Fail-stutter* fault model [Arpaci-Dusseau and Arpaci-Dusseau, 2001] attempts to bridge the gap between the simple Fail-stop and Byzantine models, by addressing *performance* faults in addition to fail-stop faults. From a higher level, the intermittent inability of a core to perform reliable computation could be considered an example of "unexpected and low performance of a component," but is of no use from the perspective of detecting and recovering from faults.

While this chapter makes every attempt to clearly articulate the device-level and architectural fault models, but for the reasons presented above, it does not formally define the models.

[3]Nanya and Goosen [1989] propose a circuit solution which only captures a small subset of the Byzantine Hardware Fault model they propose.

	Quantitative Goals				Qualitative Goals		Appropriate
	Fairness	Latency	Thrghpt.	Fault-free	Complx.	Concurrent	Timescales
Pause Exec.	X	X	X	√	Low	√	≤1ms
Spare Cores	√	√	√	X	Med.	X	100μs–1sec+
OS Reconfig.	X	X	X	√	High	X	≥100ms
Overcommit	√	√	√	√	Med.	√	100μs–1sec+

Table 5.3 Results Summary for Intermittent Fault Adaptation

5.7 Chapter Summary

This chapter takes a look at the system-level implications of *intermittent hardware faults*, an emerging reliability issue for future multicores. These are faults that arise from a combination of physical variation (e.g., manufacturing variation or in-progress wear-out), and variations in the operating conditions (e.g., temperature or voltage). The result of this variation are bursty faults that occur for a period of time, and can then disappear as operating conditions change. These bursty faults, in turn, create rapidly changing dynamic heterogeneity, in terms of different cores' ability to correctly execute instructions.

Although complex reliability techniques may tolerate many intermittent faults without affecting the rest of the system, we believe these approaches will require, or be greatly simplified by, the ability to temporarily suspend computation on a core during bursts of such faults. With this in mind, this chapter does not fully answer the question, "How do we build a reliable chip?", but rather answers the question, "Given that a reliable chip may need to suspend execution on a particular core, how can that be achieved?" To this end, the contributions of the work in this chapter are threefold. First, the emerging class of intermittent faults is discussed and brought to the attention of the architecture community. Second, three existing techniques for adapting to intermittent faults are qualitatively and quantitatively examined, exposing several deficiencies that are summarized in Table 5.3. Finally, a new adaptation technique, involving the virtualization mechanisms from Chapter 4, is proposed and evaluated. Utilizing an overcommitted system is the only mechanism to achieve high marks on all of the performance metrics across a range of timescales, gracefully handle multiple concurrent failures, and involve only moderate complexity.

By eliminating the system-level concerns through our proposed overcommitted system, researchers may find the ability to suspend execution on a core to be a useful tool — both to simplify the design and improve the coverage of reliable chips, and for other uses that have yet to be discovered.

Chapter 6

Dynamic Core Partitioning for Consolidated Servers

As mentioned in Chapter 2, *server consolidation* is a term used to denote the process of moving two or more services from multiple, separate machines onto one physical machine [Armstrong et al., 2005; Figueiredo et al., 2005; Sun Microsystems, 1999; VMware, 2006b; Waldspurger, 2002]. Server consolidation is a large and growing business, with many major computer companies, such as IBM, Sun, VMware jumping in to provide products, and major IT departments reporting to save millions of dollars in costs [Forrester Research, Inc., 2004; IBM, 2008; VMware, 2006c]. In addition, the U.S. Environmental Protection Agency lists server consolidation as one of the major tools that should be aggressively employed to reduce the environmental impact of datacenter energy usage [U.S. Environmental Protection Agency, 2007].

When configuring a server for running a single service, sufficient resources must be provisioned to allow the server to handle the expected peak demands on that service. However, during the expected case of non-peak operation, a majority of these resources can be left idle. Reclaiming these idle resources is the primary benefit of server consolidation. By joining multiple services that incur peaks at different times, resources are idle much less frequently, leading to lower capital costs, less power usage and machine room space, and often reductions in other operational costs, such as staffing.

A *system Virtual Machine Monitor* (VMM) is responsible for sharing the physical resources, such as processors, memory, disks, and network devices of the machine among multiple VMs, which might number 5–10 guest VMs [Waldspurger, 2002]. There are a multitude of issues that a system VMM must address in order to share these resources. This chapter focuses on the aspect of processor virtualization for single-chip multicore systems. In particular, the rapid switch to multicore processors provides new challenges and opportunities for the policies and mechanisms of processor virtualization on these servers. For example, multicores encourage an increase in the number of virtual processors (VCPUs) exposed to each guest virtual machine (VM). This increase is partly due to 1) the fact that the original services being consolidated may already be running on and configured for a multicore, and 2) the fact that increasing the parallelism of a workload is now required in order to increase its performance. The result is an increase in the number of VCPUs exposed to each guest VM, with a consequential increase in the complexity of managing these VCPUs.

In order to tolerate the complexity of managing multiple VCPUs of each guest VM, VMMs often adopt the policy of *gang scheduling* these VCPUs. Gang scheduling, or co-scheduling [Ousterhout, 1982], simply refers to the policy of either concurrently running *all* VCPUs of a given VM, or none of them. Though simple, gang scheduling is one of the major obstacles to efficient use of the processing resources in a consolidated server. This chapter proposed to utilize the multicore virtualization techniques from Chapter 4 to eliminate the need for gang scheduling, and by doing so, provides to opportunity to 1) efficiently handle the varying demands placed on each service, 2) improve performance by actively creating dynamic heterogeneity through the assignment of VCPUs to cores, and 3) quickly adapt to other dynamically changing characteristics, such as power, thermal, or reliability issues affecting future multicores.

6.1 Gang Scheduling: Conflicting Objectives

The VMM for a consolidated server has two primary objectives. First, it must maintain, for each VM, the degree of *performance* (e.g., throughput and request latency) and *predictability* (e.g. isolation from other VMs) required by the customer paying for the hosted server. If these performance goals cannot be met reliably, the customer will look for other alternatives.

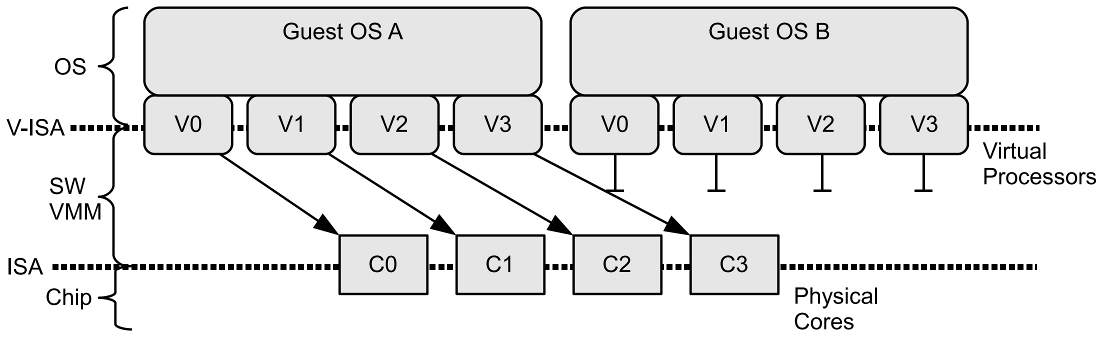
Provided the performance goals are met, the other main objective is to maximize the efficiency of the server. Additional performance can lead to lower static resource requirements (such as fewer machines and machine room space), and additional power efficiency can lead to lower dynamic requirements such as electrical delivery and cooling.

Maximizing efficiency first requires the VMM to adapt to varying demand on the services of each guest VM — this ability is the reason servers are consolidated in the first place. Adapting to demand implies that one VM must be allowed to utilize a large fraction of the available resources (e.g., physical cores) during periods of peak demand, but be able to offer those resources to a second VM when that second VM experiences high demand.

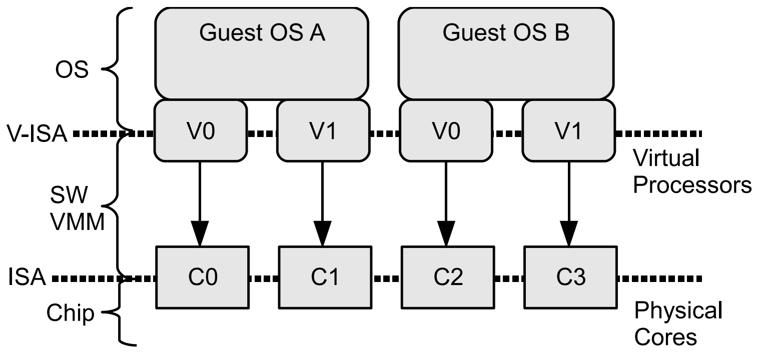
Second, VMMs can improve efficiency, as well as performance isolation, by improving the locality of per-core predictive structures, such as caches, TLBs, and branch predictors. By careful assignment of VCPUs to cores, these predictive structures can become dynamically specialized for a particular type of task, and can execute those tasks more efficiently.

Adapting to demand and maximizing locality are required to improve efficiency, but the challenges of these problems are exacerbated by the need of most VMMs to *gang schedule* the VCPUs belonging to each VM. Gang scheduling is used by VMware ESX server [VMware] and Cellular Disco [Govil et al., 1999], among others, in order to avoid the serious synchronization issues that arise when not all of the VCPUs of a given guest VM are concurrently executing (see Section 4.2).

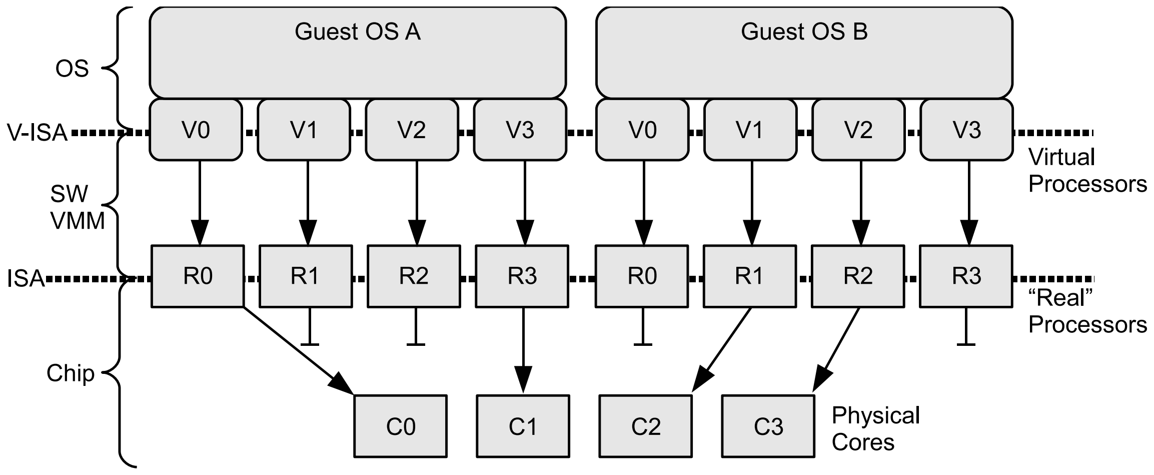
To satisfy the first goal of allowing a single VM to utilize all (or most) physical cores during high demand for the services running in that VM, a consolidated server can be configured to expose as many VCPUs to each VM as there are physical cores. This configuration is depicted in Figure 6.1(a) for a 4-core system. The figure depicts the hardware/software stack starting from



(a) Gang Scheduling



(b) Static Partitioning



(c) Overcommitting and Dynamic Partitioning with Multicore Virtualization

Figure 6.1 Techniques for Processor Virtualization of Consolidated Servers

the physical cores on the bottom, and two guest VMs, A and B at the top. In the middle, a software system VMM maps the VCPUs exposed to the guest VMs via the *Virtual ISA* (V-ISA) to the physical cores exposed by the chip's ISA. At this moment, the software VMM is running all VCPUs of A and none of B.

Such a configuration *time partitions* the processing resources of the machine. By adjusting the amount of time each VM executes before relinquishing control of the physical cores to other VMs (its *timeslice*), the VMM can dynamically adjust the fraction of resources consumed by each VM according to demand. Adjusting the timeslice also allows the VMM to optimize the request latency of each VM, i.e., the amount of time an incoming request must wait to be serviced.

The problem with gang scheduling in this manner, however, is that it is inflexible. In particular, the timesharing of each physical core among two or more unrelated activities erodes opportunities to optimize performance, efficiency, and maintain isolation due to two different problems.

First, when many VCPUs of a guest VM are idle, gang scheduling leads to inefficient use of the cores since idle virtual processors of one VM are scheduled even when non-idle virtual processors of another VM could run. To combat this problem, some gang schedulers use techniques to identify the idle loop of the guest OSs [Govil et al., 1999, 2000]. However, even with precise information about which VCPUs of VM A are idle, it is not straightforward to allow only some of VM B's virtual processors to execute, since they will be seriously affected by kernel synchronization unless *all* non-idle virtual processors of VM B can be co-scheduled.

Second, by timesharing multiple, unrelated operations onto each core, gang scheduling harms cache, TLB, and branch predictor locality. Depending on the timeslice offered to each VM, such destructive interference can be significant, and can lead to both performance and isolation problems (see Section 6.3). In an attempt to reduce the interference, the timeslice offered to each VM can be increased, directly analogous to single-OS scheduling [Torrellas et al., 1995]. However, longer timeslices come at the expense of increased latency for new requests arriving at paused

VMs, and most applications cannot tolerate an arbitrarily long delay in responses. For example, low response times for Multiplayer Online Game (MOG) servers are critical in order to provide a satisfactory user experience [Deen et al., 2006].

If isolation and/or cache performance are of primary concern for a particular system, an alternate configuration could statically partition the cores, such that each guest VMs is allocated a number of VCPUs that is only a fraction of the total cores in the system. A statically partitioned configuration is depicted in Figure 6.1(b), and involves *space partitioning* (or *physical partitioning* [Smith and Nair, 2005]) of the processing resources of the machine. Such a configuration is enabled by IBM's Logical Partitioning (LPAR) [Borden et al., 1989], or Sun's System Domains [Charlesworth, 1998]. Here, the same two guest VMs are each configured with only two VCPUs, allowing each VM to utilize half of the physical core resource of the machine. By repeatedly running each VM on only a subset of the cores, the predictive structures of each core becomes specialized for executing that VM efficiently. That is, dynamic heterogeneity is created from physically homogeneous cores. Structures that are private to each core, such as TLBs, branch predictors, L1 caches, and for the target multicore in this dissertation, L2 caches as well, are unaffected by the execution characteristics of the other VM, leading to improved isolation and performance predictability.

Static partitioning reduces the impact of interference, improves performance isolation, and can also insulate one VM from hardware or other failures on cores allocated to a different VM [Govil et al., 2000; Whitaker et al., 2002]. In addition, static partitioning as depicted in Figure 6.1(b), can reduce the overheads of processor virtualization when there is a one-to-one mapping between VCPUs and cores, as in this example. However, static partitioning trades-off the possibility of adjusting a guest VM's ability to handle periods of high demand and doesn't allow one VM to use cores left idle by another VM. Neither gang scheduling or static partitioning easily allows the system to adapt to underlying changes in the chip, such as the emerging power, thermal, or reliability challenges discussed in Chapters 1 and 5.

6.2 Overcommitting and Dynamically Partitioning

The reason VMMs typically adopt gang scheduling is that serious synchronization issues arise when not all the VCPUs of a given guest VM are concurrently executing (see Section 4.2). By using gang scheduling, an execution environment similar to a non-virtualized system is created, resulting in OS synchronization overheads that are similar to a non-virtualized machine.

This chapter proposes to utilize the multicore virtualization techniques from Chapter 4 to break the trade-offs inherent with gang scheduled consolidated servers, by allowing flexible assignment of VCPUs, from any VM, to any set of cores. A system designer and/or administrator can then choose a policy that best serves the specific needs of that specific system.

To avoid the conflicting objectives that plague gang scheduling, this chapter proposes *dynamic partitioning*, which uses the multicore virtualization techniques from Chapter 4 to enable the ability to respond to changes in both workload demand and the changing characteristics of the underlying chip, as well as the ability to optimize performance, efficiency, and isolation. To do this, a number of VCPUs equal to the number of cores is exposed to each guest VM, allowing it to execute those VCPUs on the entire machine when necessary during peaks in demand. Then, during periods of normal demand, the physical cores are dynamically partitioned among guest VMs, allowing the VMM to only execute a subset of each VM's VCPUs at a time. Since guest VMs are allocated fewer cores than VCPUs, the system becomes overcommitted.

Dynamic partitioning can be performed with an unmodified, traditional software VMM, by implementing the type of multicore virtualization proposed in Chapter 4 underneath the ISA. In this case, the chip can expose more processors to the software VMM than there are physical cores. These processors are called *real processors* to distinguish them from the virtual processors exposed to the guest OSs by the software VMM.

Dynamic partitioning is depicted in Figure 6.1(c). As shown in the figure, multicore virtualization logically adds a second layer of indirection between the virtual processors exposed the the guest VMs, and the physical cores on the chip. The software VMM is responsible for mapping the virtual processors (VCPUs) to the real processors, and the multicore virtualization implemented on the chip itself is responsible for mapping the real processors to the physical cores using the techniques in Chapter 4. Here, the physical cores are partitioned between guest VMs, but each partition is overcommitted, since there are more VCPUs exposed to each guest OS than there are available cores. However, unlike the static logical partitioning in Figure 6.1(b), the partitions in Figure 6.1(c) can easily and dynamically change so that Guest OS B can execute on zero, one, two, three, or four cores depending on its workload demands compared to those of Guest OS A. When using an unmodified software VMM and operating systems, the use of the basic virtualization and spin detection is required in order to execute efficiently.

Dynamic partitioning can lead to better cache, TLB, and branch predictor locality by improving constructive interference in the private structures of a core (within a partition) while eliminating destructive interference from the other VM. Locality is especially improved when the guest VM is running a shared memory application where different VCPUs from the same VM share instructions and data.

Though it need not necessarily be the case, in Figure 6.1(c) and the experiments in this chapter, there is a one-to-one correspondence between VCPUs and real processors. Thus the responsibility of managing the virtual processors has, in effect, moved from the software VMM to the chip. The chip is responsible for processor virtualization, and the software VMM remains responsible for the other aspects of system virtualization. The net result is that there is no additional overhead from processor virtualization when adding another layer to the stack, since there need be no overhead from the software VMM's mapping VCPUs to real processors. Even if a gang scheduling policy was still used, runtime overhead of processor virtualization is actually likely to be lower when adding multicore virtualization, since several common functions are implemented in hardware.

By adjusting the number of cores allocated to each VM, dynamic load balancing can be achieved without changing the timeslice, and hence response latency, of the other guest VM. One method to for dynamic load balancing over short timescales is to simply schedule non-idle VCPUs from one VM onto cores used by a second VM, when that VM has no non-idle VCPUs to run. Since the spin detection hardware automatically detects these idle VCPUs, only minor changes to the scheduling logic are needed. Throttling can be used to prevent VCPUs from switching cores too frequently to maintain cache locality.

A discussion of dynamic partitioning with para-virtualized VMMs is presented in the related work (Section 6.4).

6.2.1 Interrupt Handling and Transaction Latency

When a new transaction request arrives at a server, it typically comes in the form of a network I/O packet, which is handled by the processor when it receives an interrupt from the network controller. When performing processor virtualization, the VMM (whether a software VMM or the proposed multicore virtualization) has a choice when such an interrupt arrives for a particular VCPU that is not currently running. The VMM can either queue the interrupt for delivery once the VCPU is next scheduled as normal, or it can attempt to schedule that VCPU as soon as possible and deliver the interrupt more quickly.

Queuing results in extra delay for the transaction request. When gang scheduling two VMs, queuing interrupts increases the latency of request arriving when the VM is paused by (on average) half of the scheduling quantum. This average time is the expected amount of time before that VM is rescheduled and can process the interrupt, and is, for example, 25ms for a default VMware ESX Server 3.0 setup [VMware, 2006a].

To avoid this extra delay, the VMM could alternately schedule the paused VM as quickly as possible when an interrupt arrives for a paused VM. But with gang scheduling, the entire VM must be rescheduled, which destroys the locality of private, per-core structures.

Due to the sharing of cache, TLB, and branch predictor entries between VCPUs of the same VM, and hardware support for quickly moving the state of a VCPU in and out of the core, the overhead of context switching VCPUs from the same VM is small compared to the overhead of switching VCPUs of different VMs. In addition, by not requiring gang scheduling, only one VCPU needs to be switched on an interrupt to process that interrupt as quickly as possible. The result is that with dynamic partitioning, the latency of responding to interrupts is much closer to that of an unvirtualized system.

6.3 Evaluation

In this section, several aspect of dynamic partitioned are evaluated in comparison to gang scheduling, with a particular focus on the locality of per-core predictive structures such as branch predictors, TLBs, and caches. Details of the methodology for evaluating consolidated servers is described in Section 6.3.1, followed by a locality study in Section 6.3.2 and its effects on throughout and performance isolation in Sections 6.3.3 and 6.3.4. The impact of sharing read-only pages between VMs is discussed and evaluated in Section 6.3.5, and finally, the effects of scheduling policy on individual transaction latency is demonstrated in Section 6.3.6.

6.3.1 Methodology

An evaluation of consolidated servers was performed using the same *ms2sim* infrastructure described Chapter 3. Studies with consolidated workloads assumes the use of a software VMM, similar to VMware ESX Server, which virtualizes I/O, memory, and the execution of privileged instructions. Without access to such a VMM that supports the SPARC Enterprise 8000 platform used in this dissertation, the execution of consolidated workloads is emulated without modeling the overhead of this software VMM.

To create a consolidated workload, the same benchmark checkpoints used for single-VM experiments in previous chapters were combined together to create a single checkpoint which simultaneously loads both benchmarks. Each benchmark represents a *guest VM*, and contains its own Solaris 9 image and own application. The original benchmarks were set up and configured using the entire 8-core system. Each guest VM is configured with its own I/O devices and physical memory space, just as in the original workloads. From the perspective of the Simics functional simulator, simulating a consolidated workload simply means that it is loading multiple machines from the same checkpoint. In the absence of any intervention from the timing simulator, both VMs would concurrently and independently execute.

A Perl script was used to combine the checkpoints, and rename devices, such that when the Simics functional model load the single, combined checkpoint, it seamlessly loads all of the system components for two separate machines, and can execute them simultaneously. The timing model then manages the multiplexing of physical resources between the guest VMs.

The timing simulator models the VMM, which is responsible for sharing the physical resource of the target machine among guest VMs. By articulating which VCPUs from which VM run on which cores at what time, the timing simulator causes VMs to dynamically share all components modeled by the timing simulator, including the cores, caches, and TLBs.

The two guest OSs are allocated enough physical memory so that the VMM does not need to swap the *real* memory allocated to each guest VM to disk. For the experiments in Sections 6.3.2–6.3.4, the timing simulator translates real addresses to non-overlapping physical addresses using a simple linear map.

Section 6.3.5 examines the effects of sharing real pages among multiple VMs. Accesses to read-only pages that are potential candidates for sharing are identified when the virtual address is translated to the physical address within the Simics functional TLB. All pages in the instruction TLB are read-only. Read-only pages in the data TLB are identified by a cleared *Writable* bit. On a successful translation, the TLB informs the timing simulator that the page was read-only. The timing simulator then determines whether to share this page or not, depending on the policy of the experiment. For experiments in Section 6.3.5, sharing is accomplished by simply mapping shared pages from VM 2 into the physical memory space used by VM 1.

The methodology and tools developed for this dissertation were subsequently used in other research with consolidated servers [Marty and Hill, 2008, 2007].

Consolidated Server Workloads Several sets of workloads were created for experiments in this chapter. The first combines two guest VMs, where each guest VM is warmed up and fully utilizes all 8 VCPUs (i.e., it achieves near 0% idle time when run by itself on the 8-core processor). This full utilization set represents a worst-case load for the consolidated server. The second set combines two guest VMs where each guest is warmed up, but only utilizes approximately 50-80% of the 8 cores when run by itself. The lower-utilization set of workloads represents the expected common case, since servers are consolidated expressly for the purpose of increasing the utilization of the physical resources.

Creating workloads with a target level of utilization is a time consuming, trial and error process. First, for most workloads, the fraction of time processors spend idle does not maintain a linear relationship compared to the number of threads, or user think time, present in the workload driver. Second, the level of CPU utilization changes with the IPC of the simulated processor. During fast, functional simulators, processors have an IPC of one. During detailed out-of-order simulation, IPCs range from 0.3–0.8. Thus the speed of the processor changes compared to the speed of the disks and timer events for which the processor is idle waiting, and therefore the fraction of idle time changes. Third, idle time, as reported by the operating system cannot be determined until 1–2 seconds of simulated time have elapsed — longer than can reasonably be simulated with the specific IPC of the detailed timing simulator. The bottom line is that lower utilization workloads all experience some fraction of idle time, but the fraction is imprecise.

Heterogeneous workloads were created by pairing different benchmarks, executing in different VMs, and homogeneous workloads by pairing the same benchmark executing in different VMs. These first sets of workloads are used for experiments that examine throughput the cache behavior of predictive structures in Sections 6.3.2, 6.3.3, and 6.3.5.

A set of workloads is used in Section 6.3.4, which pairs one of the commercial benchmarks with one of two microbenchmarks in order to examine the effects of performance isolation. The first microbenchmark, `Comp`, is computationally intensive with a small cache footprint. The second, `Stream` is extremely cache intensive: it repeatedly streams through a large array, effectively replacing the entire private L2 caches (but only a fraction of the shared L3).

Finally, a third microbenchmark, μ`Server`, is used for experiments in Section 6.3.6 for measuring the transaction processing latency of guest VMs using different scheduling policies. A guest VM running μ`Server` is paired with a second guest VM running the `Comp` benchmark. μ`Server` measures the latency of both the time a transaction spends processing, and the time an arrived transaction spends waiting to begin processing. This microbenchmark was created in order to isolate the effects of scheduling policy, without observing the variability and additional overheads of OS network processing and the limited bandwidth of the simulated Ethernet devices and OS drivers. Instead, client and server communicate directly via memory-based queues. Each of 8 threads has its own queue. The client periodically sends transactions to the server by writing each request into the appropriate queue. A thread of the μ`Server` benchmark will begin running a transaction immediately if it is idle waiting for requests, or will hold the transaction in its queue if either the thread is processing another request, or the VCPU running that thread is paused. Transactions and are sent and enqueued both when the μ`Server` guest VM is running and when it is paused. μ`Server` runs CPU-bound floating-point transactions that each take 96k cycles ($\sim$32μs) on a lightly-loaded single-VM machine. The timing simulator itself acts as the simulated client. It sends transactions at a rate slightly less than twice the transaction latency, i.e., 8 transactions are sent every 200k cycles (67μs) for *half utilization* experiments, and at a rate slightly less than four times the transaction latency, i.e., 8 transactions every 400k cycles (133μs) for the *quarter utilization* experiments.

The out-of-order processor model was used for all experiments in this chapter. Workloads are run for 40ms (120 million cycles).

6.3.2 Locality of Per-core Structures

This section compares the branch predictor, cache, and TLB locality of configurations with two guest VMs that are configured like Figure 6.1(a) and gang scheduled with various timeslices, and that are configured like 6.1(c) to be overcommitted and dynamically partitioned.

Baseline Data Most figures in this section present data normalized to the 100μs gang scheduled configuration. For this baseline configuration, the YAGS direct branch misprediction ratio (mispredictions divided by total predictions), and TLB and cache miss rates (per thousand instructions) are shown in Table 6.1 for reference. Since experiments are run for a certain number of cycles, they can perform different amounts of work during that time. Normalized results factor in the relative amount of work performed, which is defined as *user committed instructions* as described in Section 3.4.

Basic Virtualization Overheads To cover workloads from different domains, gang scheduling is evaluated with timeslices ranging from a short 100μs timeslice, to ensure low response time in

	Workload	Br Mis %	TLB Miss	L1 Instr	L1 Data	L2 Miss	L3 Miss
Homo	Apache/Apache	2.7	6.9	58.4	108.7	65.7	8
Full	OLTP/OLTP	2.7	5.3	38.5	107.6	48	4.8
	pgoltp/pgoltp	1.7	2.2	33.9	36.9	23.7	5.8
	pmake/pmake	3.5	1.4	22.4	49.5	12.4	3.2
Hetero	Apache/Zeus	2.9	6.2	57	95.8	57.4	11.1
Full	OLTP/pmake	3.5	2.9	39.4	89.8	31.8	5.3
	OLTP/Zeus	3.1	5.6	43.5	91.9	45.7	9.3
	Zeus/pmake	3.2	3.7	53.4	92.1	36.4	11.1
Hetero	OLTP/pmake	2.5	2.1	128	67.3	21.2	2.5
Lower	OLTP/Zeus	3.2	3.9	98.2	95.7	44.9	1.8
	Zeus/pmake	3.3	2.3	103.1	84	29	3.1

Table 6.1 Baseline Branch Misprediction Rates (%), and TLB and Cache Misses per Thousand Instructions (Gang Scheduling 100μs)

environments like Multiplayer Online Gaming (MOG), to a longer 10ms timeslice, which is used by many OSs as a scheduling timeslice. A 1 ms timeslice is also evaluated.

Overcommitted dynamic partitioning experiments use a maximum scheduling timeslice of 100μs (300k cycles), meaning that VCPUs are context switched at least as frequently as they are in the the 100μs gang scheduled timeslice. As discussed in Sections 4.2.1 and 4.4.4, however, the *effective timeslice* actually used by a VCPU can be much shorter than the maximum allowable timeslice if spins (including OS idle loops) are detected.

The middle column of Table 6.2 shows the average effective timeslice for each of the benchmarks for the overcommitted and partitioned configuration. Similar to the data from Section 4.4.4, using overcommitting with dynamic reconfiguration causes VCPUs to switch on average much more frequently than the maximum timeslice. The gang scheduled configurations observe *effective timeslices* that are exactly as long as specified by the gang scheduling policy, which is 100μs, 1ms, or 10ms.

	Workload	Effective Timeslice	Switching OH
Homogeneous	Apache/Apache	11k	1.3%
Full Utilization	OLTP/OLTP	28k	0.5%
	pgoltp/pgoltp	66k	0.2%
	pmake/pmake	12k	1.2%
Heterogeneous	Apache/Zeus	15k	1%
Full Utilization	OLTP/pmake	31k	0.5%
	OLTP/Zeus	29k	0.5%
	Zeus/pmake	17k	0.9%
Heterogeneous	OLTP/pmake	10k	1.5%
Lower Util.	OLTP/Zeus	13k	1.1%
	Zeus/pmake	14k	1%

Table 6.2 Average Effective Timeslice for Overcommitted and Dynamic Partitioning (cycles), and Overhead for Switching (% of runtime).

The runtime overhead of basic virtualization is shown in the rightmost column of Table 6.2, and includes only the cost of transferring VCPU state into and out of the cores on a switch as described in Section 4.1.1. The overhead of saving and restoring VCPU state at the timeslices used for gang scheduling is negligible.

6.3.2.1 Branch Prediction Behavior

When multiple VCPUs are mapped to a single core, they can cause interference in the branch predictor of that core due to *aliasing* (i.e., conflicts) and to *capacity* pressure. Similar to aliasing between OS and user code [Chakraborty et al., 2006; Li et al., 2002], aliasing can occur between two VCPUs accessing the same hardware structures. Aliasing arises because the branch predictor is indexed and tagged using virtual addresses, which overlap among VCPUs from different VMs. When those VCPUs are executing similar code, e.g., the same OS, or same user application, that aliasing can be constructive if branches are biased in the same direction for both VCPUs. Entries in the predictor for such branches are essentially shared. When VCPUs are executing different

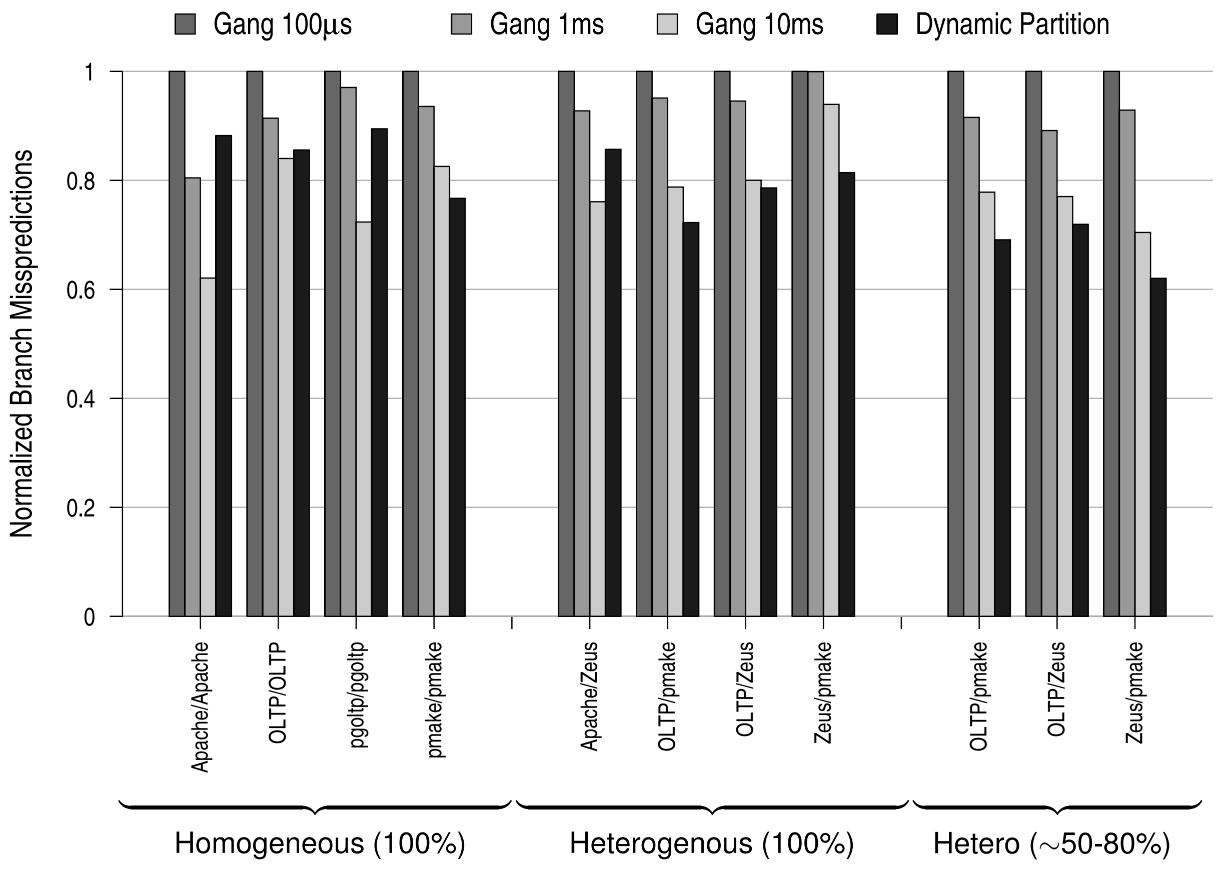


Figure 6.2 Branch predictor performance of consolidated workloads with different scheduling policies

code, branches that alias are more likely to be biased differently, since they are more likely to refer to different static branches. These entries are not shared, and the aliasing creates conflicts.

Capacity pressure arises when branches that do not alias (i.e., have different virtual addresses) either evict entries that map to the same index in the tagged "exception" tables of the YAGS predictor, or overwrite the bias stored in the untagged "choice" table.

When the timeslice that each VCPUs executes before switching to the next VCPU is short, both capacity and aliasing are larger problems, since the second VCPU will overwrite or evict many of the non-shared (i.e., conflicting) entries used by the first VCPU, so that when the first VCPU is rescheduled, many entries in the predictor are incorrect. As the timeslice increase, we would expect interference from the other VCPUs to diminish, and therefore branch misprediction rates to improve.

Figure 6.2 shows the branch misprediction rates (misrates) for the four scheduling policies, normalized to the baseline 100μs gang scheduled configuration. Indeed, for all but one workload, the branch misrates drop significantly. With the exception of the full utilization `Zeus/pmake`, workloads see a 3–19% reduction in mispredictions for the 1ms timeslice, and 16–38% improvement for the 10ms timeslice.

Across all benchmarks, dynamic partitioning decreases the misprediction rates by 12–28%. Despite a short average timeslice, the overcommitted and partitioned configuration multiplexes multiple VCPUs from the same VM onto each core, creating constructive instead of destructive aliasing. For seven of the eleven workloads, the branch prediction performance of dynamic partitioning rivals that of the 10ms gang-scheduled configuration, and beats the 1ms gang-scheduled configuration in all but one workload.

Surprisingly, there is not a significant difference between the homogeneous (same application) workloads and the heterogeneous workloads. It would be reasonable to expect that the homogeneous workloads would provide constructive aliasing, since all VCPUs in both VMs are executing the same OS and application. While this constructive aliasing might be occurring, given the slightly lower baseline misrates for the homogeneous workloads, as shown in Table 6.1, it does not make a sizable impact on the relative performance of the different scheduling policies compared to the heterogeneous workloads.

The homogeneous `Apache/Apache` workload sees the largest improvement from longer gang scheduling timeslices. The reason is likely due to the fact that `Apache` has a very large OS and user instruction footprint, sees significant interference within a single VM from these two components, and shows the most improvement among similar workloads when spreading OS and user entries across separate structures [Chakraborty et al., 2006]. It is thus no surprise that this benchmark would also show significant interference between different VMs.

6.3.2.2 TLB Behavior

As with the branch predictor, different VCPUs can cause interference in the TLB of a core. When those VCPUs are from the same VM, and when using the policy of sharing entries among those VCPUs, the interference is again capacity or conflict. We would thus expect interference to diminish for longer timeslices. Due to the policy of invalidating unlocked TLB entries when switching to a VCPU from a different VM, entry-by-entry interference does not occur. The TLB overhead for different gang scheduled timeslices simply becomes a measure of the ability to amortize the cost of the switch.

Figure 6.3 shows the TLB misses normalized to that of the gang-scheduled 100μs timeslice. TLB misses for this baseline are provided in Table 6.1. As expected, when moving to longer timeslices, the number of TLB misses with gang scheduling is reduced. For most of the full utilization workloads, the fraction of TLB misses for the 1ms and 10ms gang-scheduled and the dynamically partitioned configurations are very similar. The improvement for workloads containing `pmake` (i.e., `pmake/pmake`, `OLTP/pmake`, and `Zeus/pmake`) is the most pronounced. This is because `pmake` does not put as much pressure on the TLB as the other benchmarks, as evident from Table 6.1. Thus, a more significant fraction of the misses in the baseline, 100μs configuration are from invaliding on a VCPU switch rather than demand misses. This benchmark does incur an order of magnitude more page faults due to never-before mapped pages than other benchmarks, however.

Unlike the full utilization workloads, dynamic partitioning does not provide as much benefit for the lower utilization workloads. This is because the cores switch VCPUs more frequently as VCPUs become idle (see Table 6.2), creating additional interference.

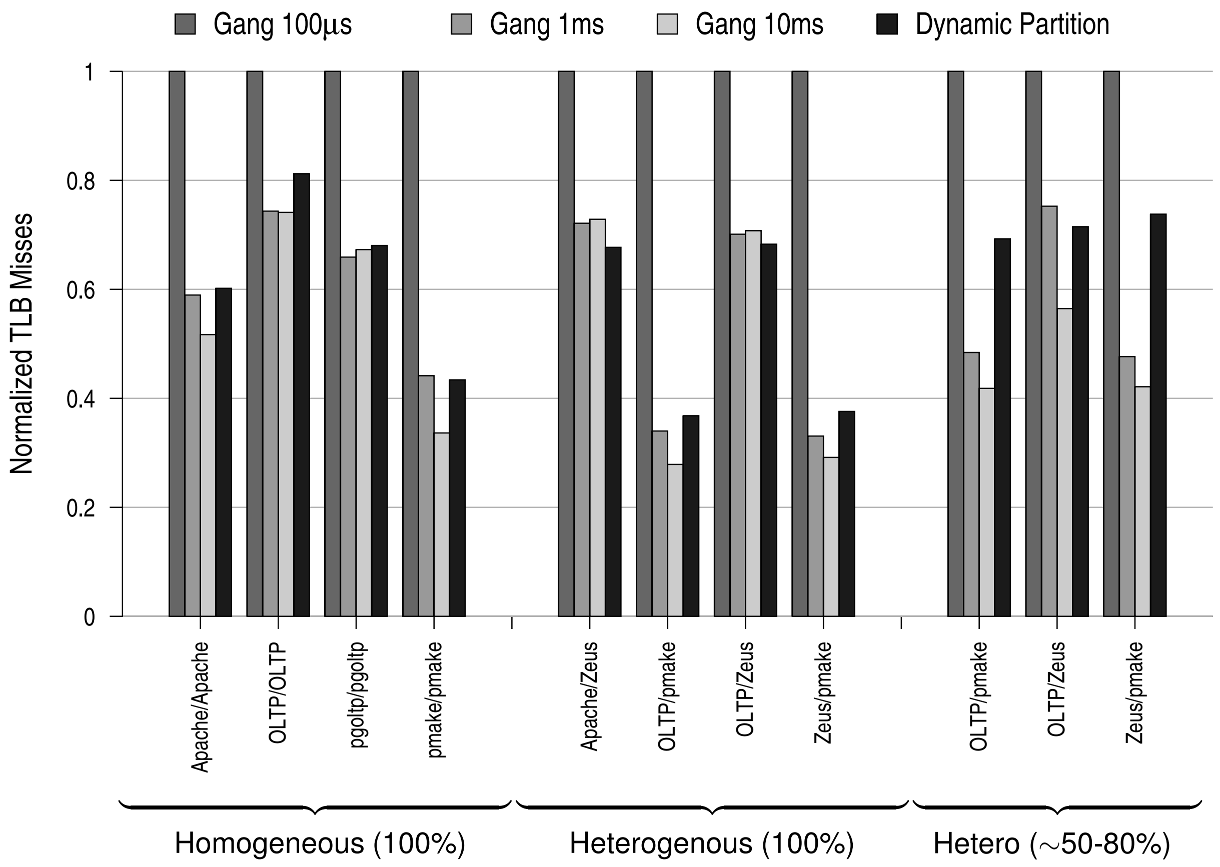


Figure 6.3 Normalized TLB misses of consolidated workloads with different scheduling policies

Using another policy for managing the TLB contents of different VMs, such as transferring the entire TLB with the rest of the VCPU state, would certainly be possible. However, the *Transfer* policy incurred more runtime overhead than did invalidating (Section 4.4.2).

6.3.2.3 Cache Behavior

Figure 6.4 shows L1 instruction and Figure 6.5 shows L1 data cache misses for the four scheduling policies. Misses are again normalized to the 100μs gang-scheduled policy.

For most workloads, the fraction of instruction cache misses is very stable across the range of scheduling policies. Most of the gang scheduling policies show minimal changes in L1 instruction

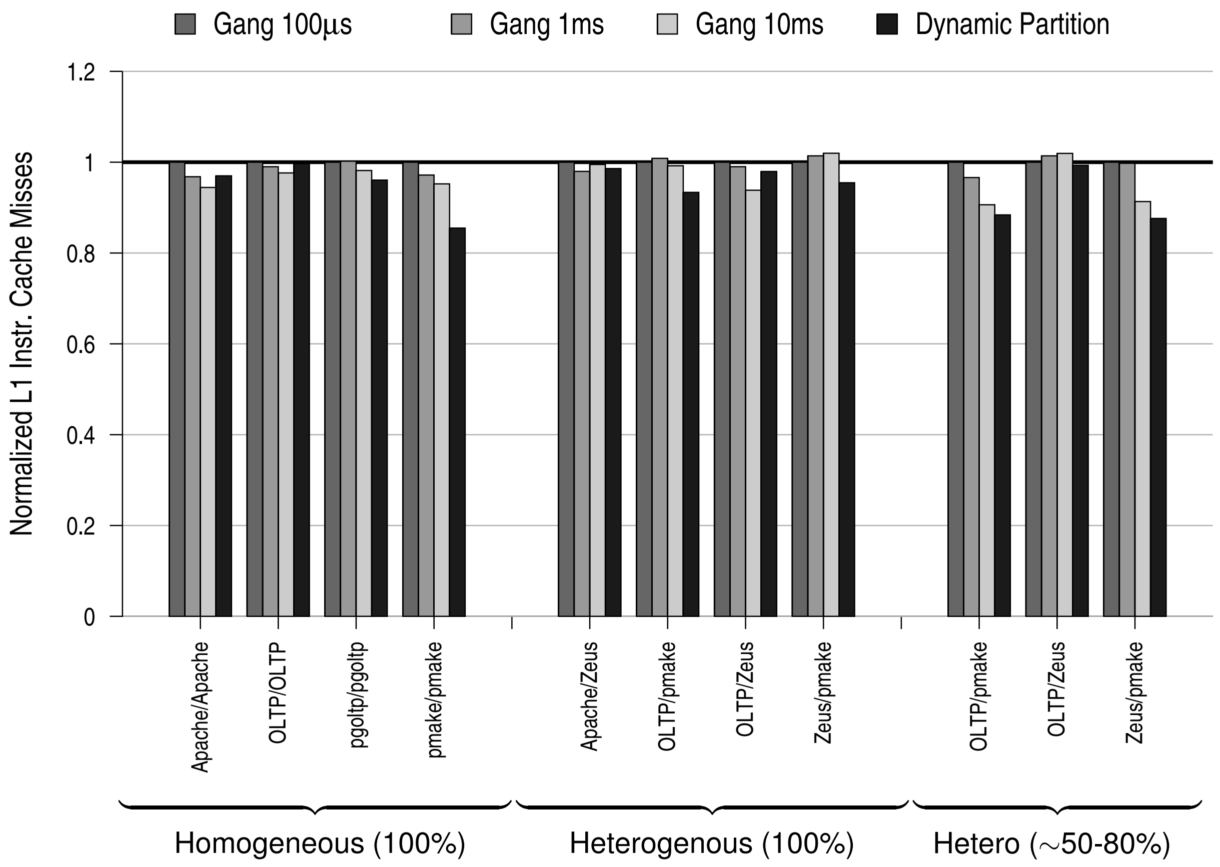


Figure 6.4 L1 Instruction Misses from Different Scheduling Policies. Results are normalized to gang scheduling with a 100μs timeslice (lower is better).

cache behavior. For these benchmarks, the small (16 KByte) L1 caches are replaced multiple times within even the shortest gang scheduling timeslices.

With dynamic partitioning, especially with workloads containing `pmake`, instruction cache misses go down by up to 16%. The reason `pmake` is most affected is that its instruction cache footprint and baseline miss rate are smaller than the other benchmarks.

With the L1 data cache, as shown in Figure 6.5, longer gang scheduling timeslices help more: many see a 5-10% decrease with 1ms and 10ms timeslices, and also significant decreases for dynamic partitioning. `pmake/pmake` and `pgoltp/pgoltp` have fewer L1 data misses compared to the other benchmarks (see Table 6.1), and therefore see a larger relative change.

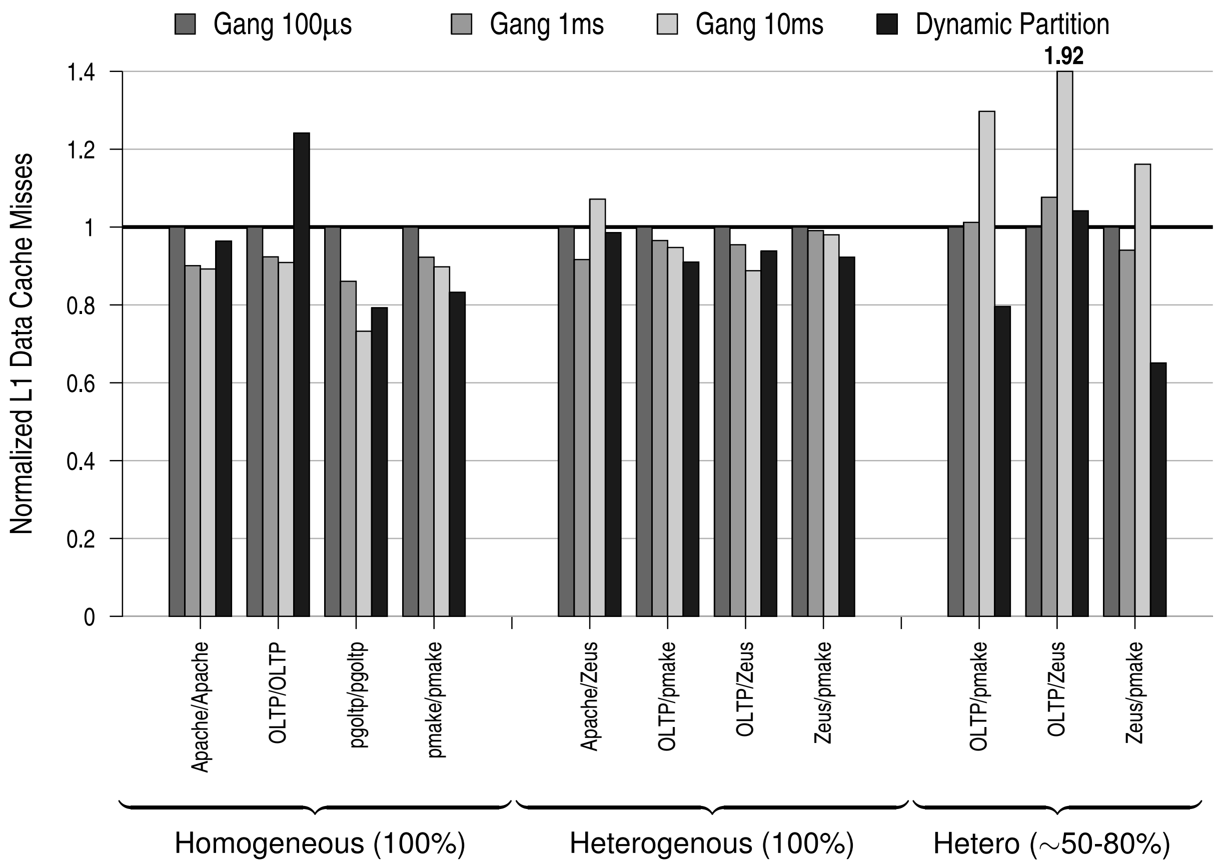


Figure 6.5 L1 Data Misses from Different Scheduling Policies. Results are normalized to gang scheduling with a 100μs timeslice (lower is better).

With dynamic partitioning, data cache misses increase for `OLTP/OLTP` by 22%. Even though VCPUs share the address space, they do not have as much data in common in the L1 working set as do the other workloads. For the lower utilization workloads `OLTP/pmake` and `Zeus/pmake`, dynamic partitioning results in a significant drop in data cache misses. This is due partly, again, to the fact that `pmake` has low baseline miss rates, and due partly to the fact that the dynamically partitioned workloads are not repeatedly entering and leaving the OS idle loop and incurring misses in the OS scheduler.

The most obvious outliers in Figure 6.5 are the lower-utilization workloads `OLTP/pmake` and `OLTP/Zeus` with gang scheduling for a 10ms timeslice. These benchmarks see a significant increase in OS activity, and subsequent increase in L1 misses (but not L2 misses). The other 50% utilization workload, `Zeus/pmake`, as well as the full-utilization `Apache/Zeus` also see 7–18% more data cache misses for gang scheduling with 10ms. Due to the complexity of the workloads and their interactions with virtualization, the root cause of this increased activity is not entirely understood, but appears to be related to the handling of hardware generated interrupts. One hypothesis for these additional misses relates to the fact that the OS and hardware timers are not virtualized, e.g., multiple hardware interrupts (including timer interrupts) can become queued up (or NACKed and retried) when a VM is rescheduled after being paused for 30 million cycles. Time (e.g., additional OS instruction) and effort (e.g. additional data cache misses) are are required in order to process these additional interrupts. Full-utilization workloads with the longer gang-scheduling timeslice are not affected as significantly.

Unlike the small L1 caches, the larger, 512KByte L2 caches do see a significant change in behavior with different scheduling policies. The reason is that the L2 caches, similar to the branch predictors and TLBs, exhibit locality on long enough timescales to be affected by the VCPU timeslice.

Figure 6.6 shows the total L2 misses for different scheduling policies, and breaks down these misses into their various components. Similar to the previous graphs, the four sets of bars for each workload represent, from the left, the baseline 100μs gang-scheduled configuration, to which others are normalized, the 1ms gang-scheduled, 10ms gang-scheduled, and on the right, the dynamically partitioned configuration. The total height of the bars represents the total L2 misses. The dark bars at the bottom represent L2 misses that are satisfied by a cache-to-cache transfer from another L2. Above that, the light bars in the middle represent misses satisfied by the shared L3 cache, and at the top, the medium gray bars represent L2 misses that go to main memory.

Overall, as the timeslice for gang scheduling increases, the total number of L2 misses drops dramatically for most workloads. A 25–47% reduction is observed for the 10ms timeslice. Even

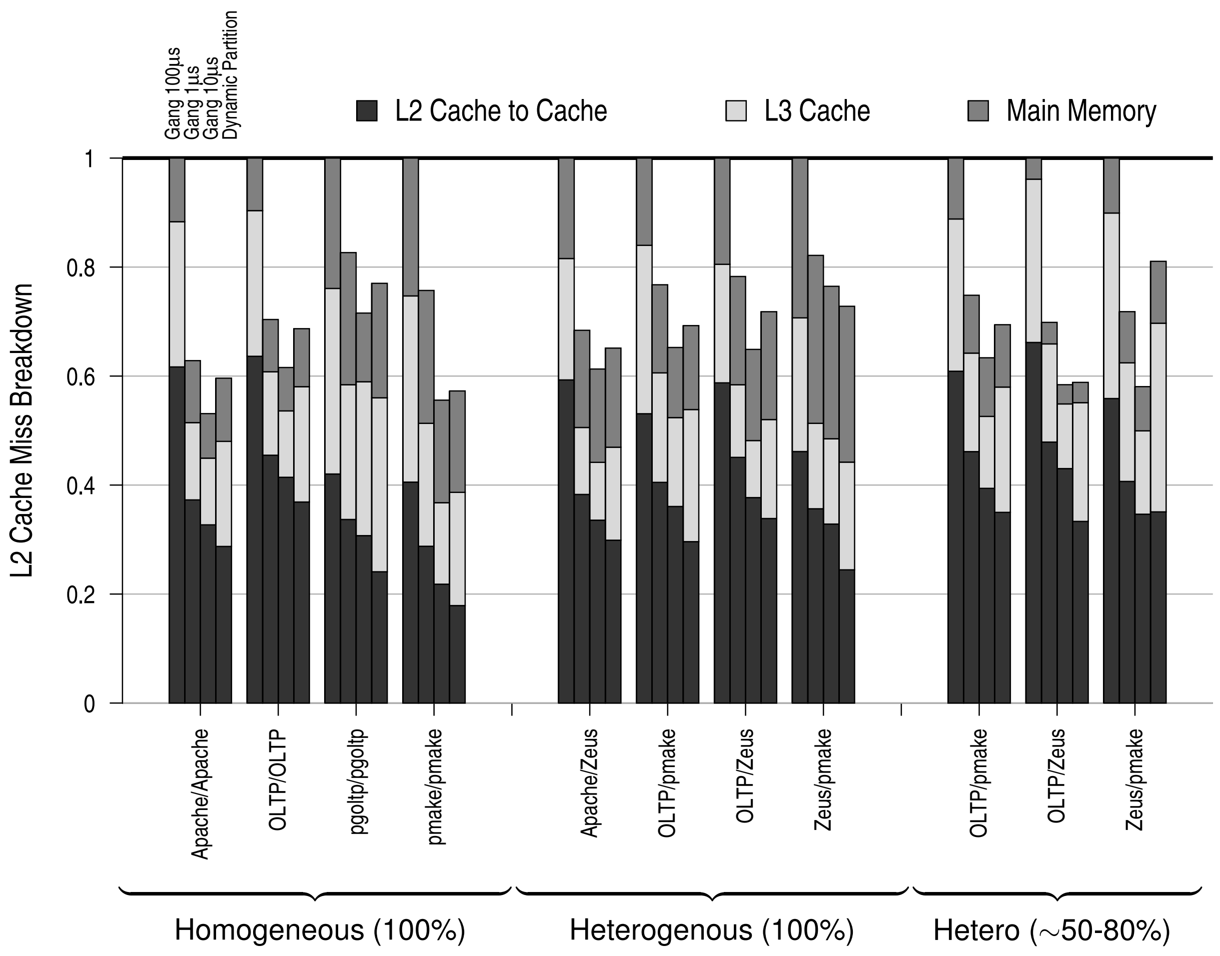


Figure 6.6 Breakdown of L2 Misses for Different Scheduling Policies. Results are normalized to gang scheduling with a 100μs timeslice (lower is better).

though the timeslices for dynamic partitioning are on average much smaller, the number of misses is less than the 1ms gang-scheduled timeslice for all but one workload, and often comparable with the 10ms timeslice.

Cache-to-cache transfers satisfy nearly half of all L2 misses for most workloads. The reason is that the L3 cache is exclusive with the L2s, thus a miss to any shared data (such as instructions) must be satisfied through a transfer. As the timeslice increases for gang scheduling, the interference in the private L2s is reduced, and thus shared data is more likely to be retained, preventing a miss. Cache-to-cache transfers are especially reduced for the dynamic partitioning configuration, since only half as many private L2s are potentially able to source the data. Compared to 10ms gang scheduling, the number of misses satisfied by the L3 increases in every benchmark to make up for fewer cache-to-cache transfers.

For most benchmarks, the number of off-chip misses remains stable across all scheduling policies. Note, however, than `pgoltp/pgoltp` observes only *half* of the off-chip misses for the 10ms gang scheduling configuration compared to the other 3 configurations. The reason for this dramatic drop is that 1) the working set requirements for `pgoltp/pgoltp` are only somewhat larger than the 12MBytes of on-chip space, and 2) this workload observes significant temporal locality in the 8MByte shared L3 cache, but does so only over long timescales. With each 1ms timeslice, the L3 cache observes enough misses to evict one third of the entries in the L3, but the new lines that are installed are not reused again for several more milliseconds. Thus, the 10ms configuration sees abundant reuse, whereas the shorter timeslices do not.

6.3.3 Throughput Performance

Given the effect of scheduling policy on the predictive structures, we now turn to look at the overall impact on performance. For most experiments, we would expect the overall performance to track closely with the changes in the behavior of the predictive components. The notable exception is for the lower utilization workloads, where each of the gang scheduled VMs observes VCPUs entering the OS idle loop for some fraction of the time, whereas the spin detection hardware used by the overcommitted technique can detect an idle VCPU and schedule a more productive one.

Figure 6.7 shows the overall performance of each VM for the consolidated workloads for the four scheduling policies. Total performance is normalized to gang scheduling at 100μs. Speedup, for two experiments $ExpA$ and $ExpB$ is simply the average speedup of the two VMs, and is

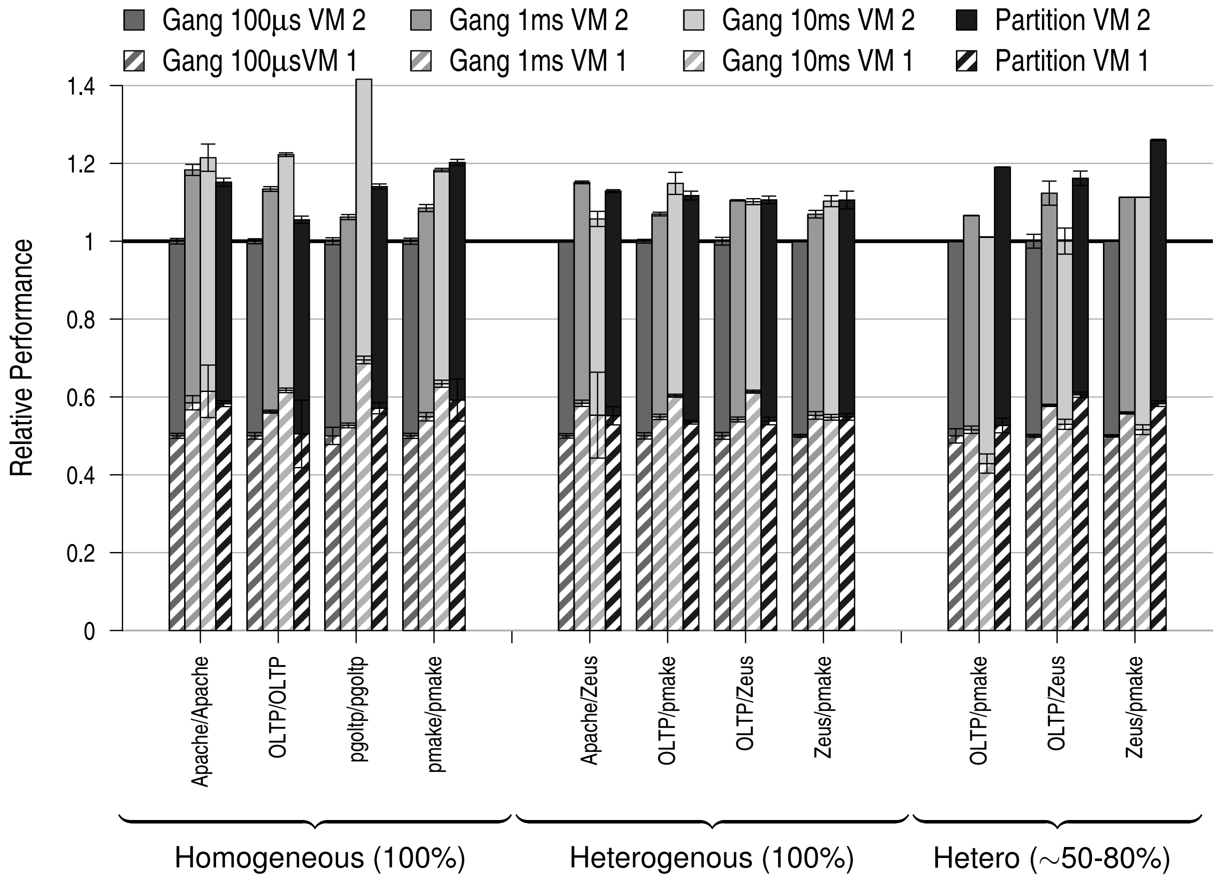


Figure 6.7 Normalized Performance of Different Scheduling Policies. Results are normalized to gang scheduling with a 100μs timeslice (higher is better).

calculated as:

$$Speedup = \frac{1}{2}\left(\frac{UserIPC_{VM0}^{ExpB}}{UserIPC_{VM0}^{ExpA}} + \frac{UserIPC_{VM1}^{ExpB}}{UserIPC_{VM1}^{ExpA}}\right)$$

where $UserIPC$ is the number of committed *user* instructions over the *total* number of cycles for the reasons discussed in Section 3.4. The graph breaks down the speedup component from each VM. Striped bars represent VM 1, and solid bars, VM 2. Error bars represent the 95% confidence interval, which is calculated independently for each VM.

As expected from the previous cache, TLB and branch prediction results, overcommitting and dynamic partitioning provides speedups from 10–20% for all but one full utilization workloads, and a slightly higher 18–25% for the lower utilization workloads, where it can recover some of the time VCPUs spend idle. Dynamic partitioning for `OLTP/OLTP`, which incurs additional L1 data misses, and doesn't improve the TLB as much as many workloads, observes only a 5% speedup, which is considerably worse than the 1ms and 10ms gang scheduled configurations. Interestingly, the overcommitting and dynamic partitioning proposal is able to achieve these speedups while context switching VCPUs on average every 2–20μs.

Using a 1ms timeslice for gang scheduling improves performance in all experiments by 5–19%, although this comes at the price of additional latency on each request.

In most cases, gang scheduling with a 10ms timeslice improves performance similarly to, or marginally better than a timeslice of 1ms, as would be expected from the previous data. The first exception is `pgoltp/pgoltp`, which observes a 40% speedup. As noted in Section 6.3.2.3 this benchmark observes a 50% drop is off-chip misses, directly impacting performance. The other exceptions are the lower-utilization `OLTP/pmake` and `OLTP/Zeus` workloads, which see an increase in L1 data cache misses, as discussed is Section 6.3.2.3.

Although the speedups for dynamic partitioning are modest, these experiments demonstrates that more flexible scheduling algorithms, enabled by the ability to overcommit the cores used by a single VM, can be important by providing the throughput of a long gang scheduling timeslice, while providing the expected transaction latency of a short timeslice.

6.3.4 Performance Isolation

Any resource shared among VMs can become a point of contention. When one VM disproportionately utilizes the resource, it can significantly impact the performance of the other VMs. When one VM suffers low performance at the hands of another, a consolidated server's overall efficiency can be affected. Also important for customers paying for the service is *predictability* of the performance their applications provide. If performance of a particular application is always lower

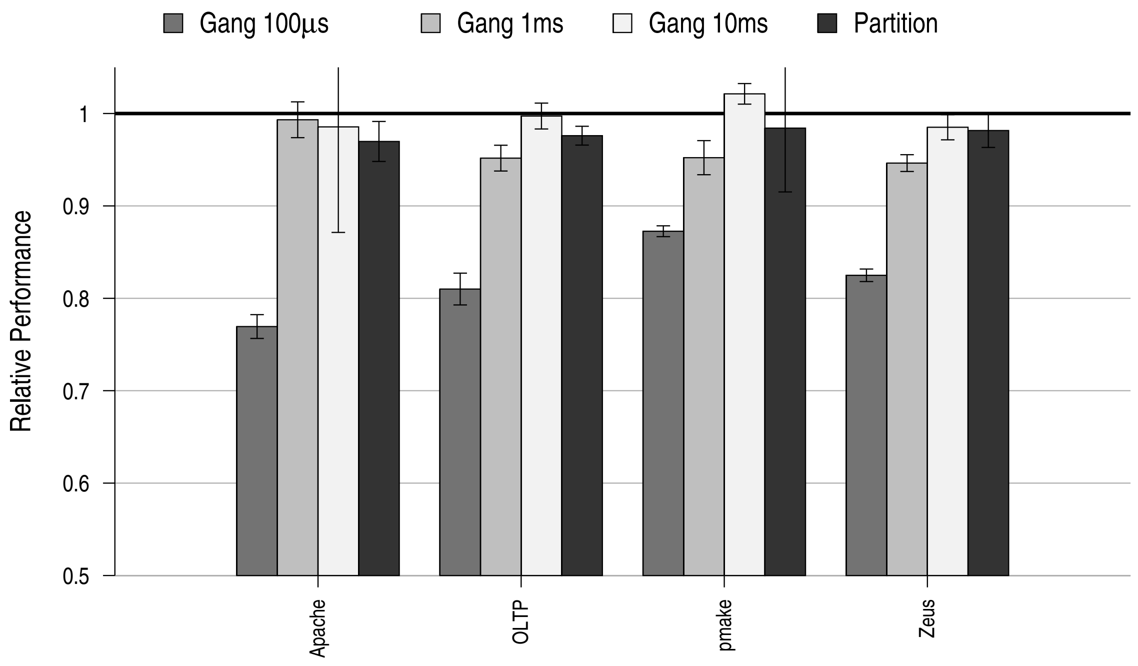


Figure 6.8 Performance Isolation. For each scheduling policy, bars represent the difference in performance between pairing the commercial workload in one VM with either a computationally-intensive (*Comp*), or cache-intensive (*Stream*), microbenchmark in the second VM. Higher is better.

than needed, more computing power can typically be purchased. But if performance is sometimes adequate and sometimes not, customers will look elsewhere.

Performance isolation refers to the ability of a system to insulate one application, or guest VM, from performance artifacts attributed to another application or guest VM. Numerous resources, including memory, I/O devices, and caches can affect the performance isolation (or lack thereof) of a system. For this reason, consolidated servers often partition these resources (statically or dynamically) among guest VMs [Borden et al., 1989; Charlesworth, 1998; Jann et al., 2003], or use other mechanisms to prevent interference [Waldspurger, 2002].

Of interest to the study in this chapter is the impact of interference from the on-chip predictive structures. In particular, this study examines the private L2 caches, where the scheduling policy mapping VCPUs to cores can create or minimize interference. With short gang scheduling timeslices, guest VMs share the private L2 caches, with both VMs frequently accessing lines and evicting each other's data. When one VM heavily utilizes these caches, it can significantly interfere with the cache performance of the other VM. Longer gang scheduling timeslices allow one VM to evict even more of the other VM's data, though longer timeslices also allow this cost to be amortized. As evident from Section 6.3.2.3, however, the the ability to flexibly assign VCPUs to cores by overcommitting and partitioning can naturally reduce this interference. The shared L3 cache is still subject to interference from different VMs, but could adopt other policies to improve isolation [Kim et al., 2004; Rafique et al., 2006; Suh et al., 2004].

Figure 6.8 compares the ability of the four scheduling policies from Section 6.3.2 to isolate the effects of sharing the L2 caches. Each bar represents the performance of the commercial workloads running with the `Stream` microbenchmark relative to the performance of the same scheduling policy with the same workload paired with the `Comp` microbenchmark. A value close to 1.0 means that the workload is nearly unaffected by cache interference caused by the `Stream` microbenchmark. Only the performance of the workload is measured, not that of the microbenchmarks.

When combined with the *Stream* microbenchmark, the commercial workloads' performance suffers substantially with gang scheduling at the 100μs timeslice (by 13–23%). Longer timeslices amortize the interference created the stream benchmark, resulting in 1–6% loss for the 1ms, and nearly zero loss for 10ms.

Overcommitting and dynamically partitioning the L2 caches between the guest VMs can also nearly eliminate the interference from the `Stream` microbenchmark, resulting in a small performance loss of 1.5–3% compared to being paired with the `Comp` microbenchmark. This performance is better than 1ms gang scheduling for all workloads except `Apache`, and similar or slightly worse than 10ms gang scheduling. As with the previous sections, this positive result for dynamic partitioning is obtained without the need for a long timeslice, and therefore without impacting the latency of requests arriving for the VM.

`pmake` appears to speedup with the `Stream` microbenchmark for the 30ms gang scheduling timeslice, though this is just an artifact of workload variability.

6.3.5 Sharing Real Pages Between VMs

Similar to processor virtualization, memory virtualization in a system VMM also adds another layer of indirection between the virtual memory visible to a user application and the physical memory implemented in the hardware. This intermediate *real memory* is the memory that each guest OS sees as the physical memory of its own machine. The OSs are responsible for managing the mapping between virtual and real memory, and the VMM manages the mapping between real and physical memory [Smith and Nair, 2005].

Multiple guest VMs that are executing the same OS, and/or running the same applications, have a tendency to use real memory pages that contain the exact same contents [Bugnion et al., 1997; Waldspurger, 2002]. The reason is that pages containing instructions, or read-only data structures, are expected to be identical among instances of the same OS and applications. A high-performance VMM then has the opportunity to share the physical memory used by such pages among the multiple real memory pages from multiple machines. Waldspurger [2002] reports that 10–43% of real memory is shareable among 5–10 real-world guest VMs with VMware ESX Server.

VMware identified identical real pages by scanning their contents. It allows such pages to be shared among guest VMs using standard copy-on-write techniques within the VMM [Waldspurger, 2002]. This technique is able to identify not only instruction and read-only data pages, but also writable data pages that happen to have the same contents.

Sharing identical real pages reduces the burden on physical memory, allowing a VMM to further overcommit the physical memory, while reducing the amount of time spend swapping real pages to disk. Of interest to the study in this chapter, real page sharing can also increase memory access locality and reduce the burden on the caches, though Waldspurger [2002] reports that the effect is generally small. The reason the effect of cache locality can be different than the fractional physical memory savings is because of the data access patterns: if proportionally more accesses arrive for private rather than shared pages, cache locality is only minimally enhanced, and vice-versa.

Examining the Impact Several experiments were performed to examine the impact of sharing real memory among guest VMs. In the experimental setup in this chapter, physical memory is not overcommitted, and thus sharing real memory can only improve cache locality. Rather than examine the contents of each page, these experiments assume that only (real) pages marked read-only by the guest OS are candidates for sharing. Read-only pages include all instruction pages, and a small but significant fraction of data pages, both in the OS and several user applications.

Since all workloads are running the same OS (Solaris 9), all accesses to read-only OS pages, including all instruction fetches, are shared. For homogeneous workloads, where the same application is running in each VM, read-only pages are shared for user pages as well.

Table 6.3 reports the fraction of instruction fetches and data loads for shared real pages. All instruction fetches are shared for homogeneous workloads. All OS fetches are shared in other workloads. Since many workloads spend a significant fraction of their time executing OS code, 33–90% of fetches are shared even for heterogeneous workloads. Since instruction misses contribute significantly to the overall memory footprint and cache misses of these workloads [Ailamaki et al., 1999; Chakraborty et al., 2006], such significant sharing could have an impact on performance.

		% Shared	
	Workload	**Fetches**	**Loads**
Homogeneous Full Utilization	Apache/Apache	100%	5.8%
	OLTP/OLTP	100%	3.1%
	pgoltp/pgoltp	100%	1.8%
	pmake/pmake	100%	13%
Heterogeneous Full Utilization	Apache/Zeus	89.8%	2.5%
	OLTP/pmake	33.9%	1.3%
	OLTP/Zeus	68%	1.9%
	Zeus/pmake	54%	2.7%
Heterogeneous Lower Util.	Zeus/pmake	71.2%	3.2%
	OLTP/pmake	68.4%	0.9%
	OLTP/Zeus	80.5%	2.8%

Table 6.3 Fraction of Requests Shared Among Guest VMs

Data sharing is much more limited however, even among homogeneous workloads with VMs running the same user applications. When two VMs of `pmake` are running, 13% of loads are shared. In this benchmark, most of the shared loads go one of two read-only segments in the `cc` compiler binary. In addition, `pmake` observes numerous loads to a handful of read-only kernel pages in part of a data segment used by the UFS file system kernel module. Other homogeneous workloads, on the other hand, see less than 6% of loads being shared.

All heterogeneous workloads observe 3.2% or fewer shared loads, since only OS pages are shareable. With the exception of `pmake`, the fraction of loads to shared data is expected to have little, if any, impact on cache performance.

The amount of real memory pages shared among VMs if not expected to further increase, as a fraction of total allocated real memory, when more than two VMs are consolidated [Waldspurger, 2002].

Figure 6.9 examines the cache impact of shared real pages. The figure shows L2 misses normalized to the 100μs gang-scheduled experiment *without* the sharing of real pages. Normalized

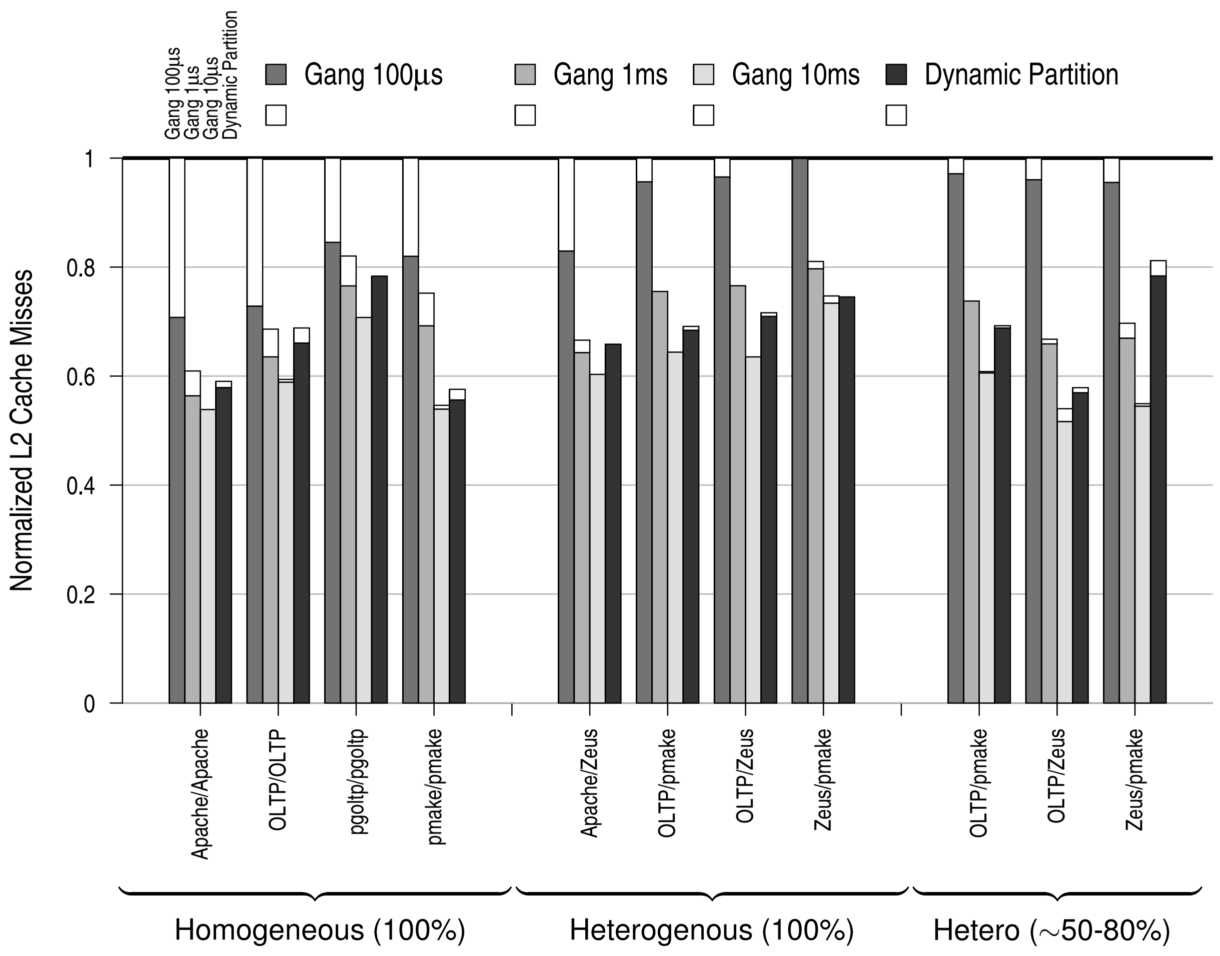


Figure 6.9 Normalized L2 Misses when Sharing Real Pages. Results are normalized to the non-shared 100μs gang scheduled configuration (lower is better). White, outlined bars represent L2 misses without sharing.

misses from the shared experiments are shown in colored bars. The outlined, white bars on top represent the non-shared experiments (the same data from Figure 6.6).

For homogeneous workloads on the left, sharing read-only pages make a considerable difference in the number of L2 misses for the 100μs timeslice, and a 5–7% different in misses for the 1ms timeslice. This reduction is expected since on a VM switch, the new VM can potentially use any of the instructions brought into the L2 cache. Also as expected, sharing has little impact on the dynamically partitioned experiments, since VMs do not compete for private L2 space. It similarly has little impact on the 10ms timeslice, since 30 million cycles is long enough to amortize the cost of a VM switch.

When sharing real pages, and consolidating the same applications on the same OS, the timeslice and scheduling policy has significantly less impact on cache performance than when real pages are not shared. The differences in L2 cache misses are much less pronounced, however, when consolidating workloads running different applications. The `Apache/Zeus` workload observes 18% fewer misses, all the rest see less than a 5% change. Branch predictions and TLB misses remain unchanged, since both operate on virtual addresses.

6.3.6 Transaction Latency

As discussed in Section 6.2.1, the choice of scheduling policy can have implications on the latency of transactions arriving at the machine. To determine the effects of scheduling policy on transaction latency, experiments were performed using the μ`Server` microbenchmark described in Section 6.3.1. Transactions arriving for idle threads on running VCPUs are processed immediately. Transactions arriving for threads already busy processing a transaction are enqueued, as are transactions arriving when the VCPU running the thread is paused.

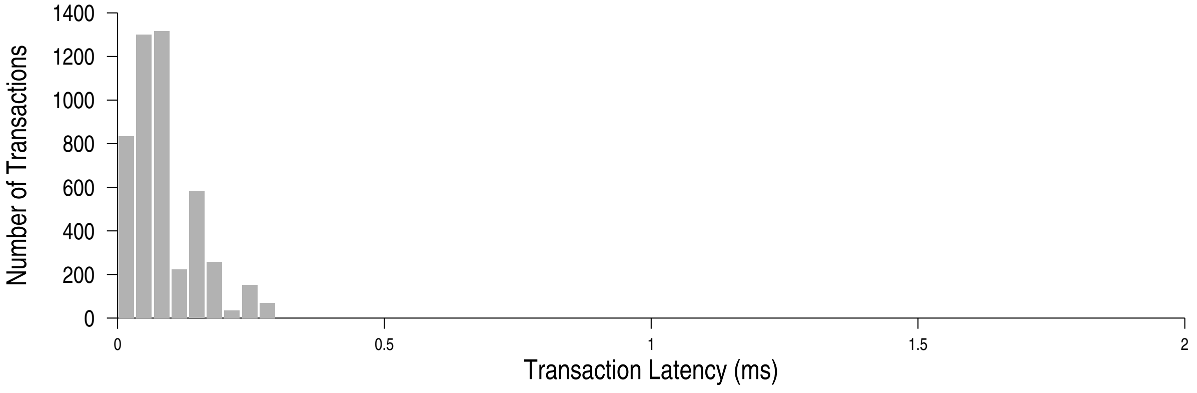
Experiments were performed using two transaction arrival rates. In the first, *Half Utilization* experiments, transactions arrive at just less than the processing rate of the consolidated guest VM (i.e, half the processing rate of a single-VM machine). In the second, *Quarter Utilization* experiments, transactions arrive at just less than the half processing rate of the consolidated guest VM, or one quarter the processing rate of a single-VM machine. Similar to Section 6.3.4, the performance of the second guest VM is not monitored.

Figure 6.10 shows histograms of the transaction latencies for the three gang scheduling timeslices, 100μs, 1ms, and 10ms, for the half utilization experiments. Note the that scale of the x-axis is different for the 10ms configuration. Each bin represents 33μs for Figures 6.10(a) and 6.10(b) and 180μs for Figure 6.10(c). Transaction execution time when running on the core is approximately 32μs. Note also that the y-axis is different for the different gang scheduling timeslices (but the same scale is used for each respective configuration in both Figure 6.10 and Figure 6.11).

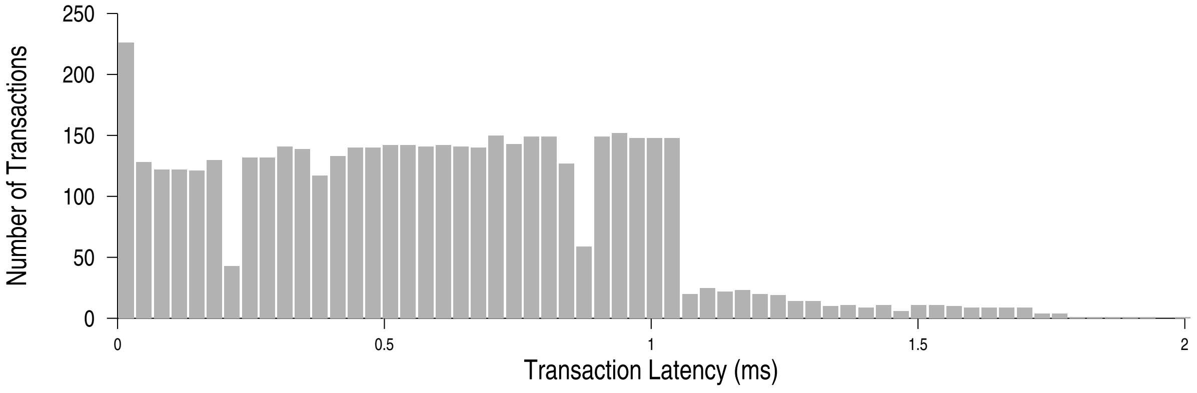
When using a 100μs timeslice, most transactions finish in less than 100μs, though many are delayed up to 265μs due to both time spent waiting for the μServer guest VM to be rescheduled, and for any previously enqueued transactions to be processed. This effects of these two delays are more obvious for the 1ms timeslice configuration in Figure 6.10(b). Many transactions, those arriving when the server is scheduled and mostly idle, finish within 33μs. Many more are roughly uniformly spread out for the next 1ms, due to their arrival during periods when the guest VM is paused, and several transactions take up to 2ms due to the additive effect of previously enqueued transactions. For gang scheduling with a 30ms timeslice (Figure 6.10(c)), 36% of transactions are processed within the 33μs execution latency, the rest are spread out to just over 10ms.

The half utilization configurations roughly represent the worst-case latencies for a gang-scheduled server that is still capable of sustaining the load. The quarter utilization configurations provide much more favorable latencies. In the quarter utilization experiments, half of the transactions, on average, are expected to arrive when the μServer guest VM is paused, and will still experience delay, but transactions arriving when the μServer VM is scheduled have a much lower probability of waiting for the processing of older queued transactions.

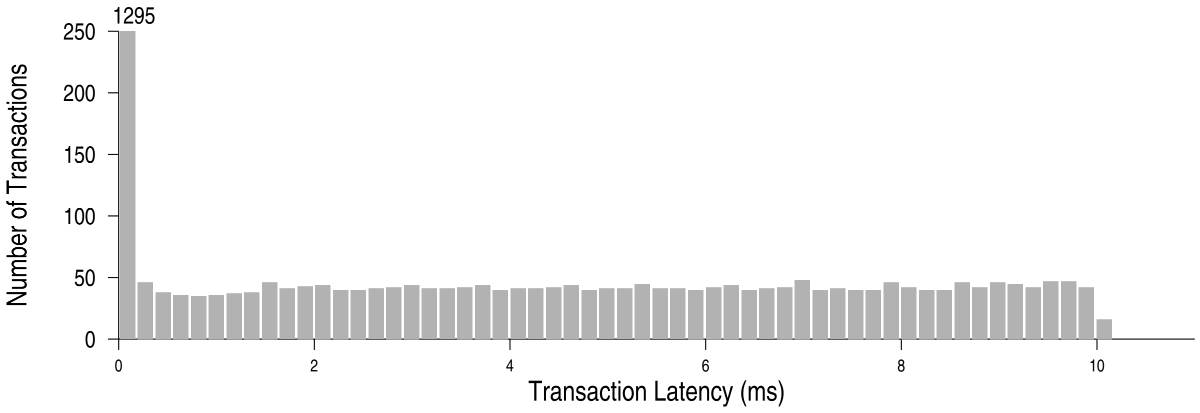
Figure 6.11 shows the same data for the quarter utilization experiments. For all gang scheduling timeslices, many transactions are still delayed by long amounts of time, although the number (and fraction) of transactions being delayed drops. For the 100μs timeslice, 53% of transactions are finished within the first 33μs. All transactions finish within 140μs (roughly the 100μs timeslice plus 33μs processing time). The total number of committed transactions is half that of Figure 6.10(a).



(a) Gang Scheduling with a 100μs Timeslice

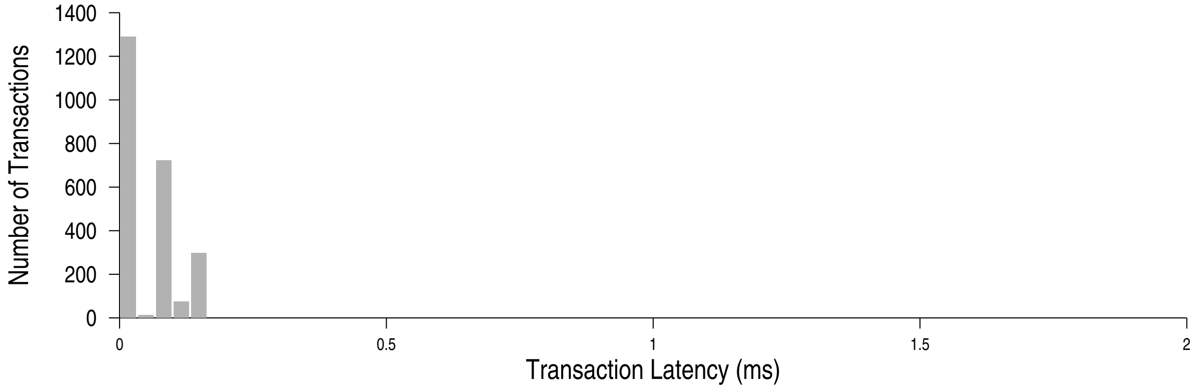


(b) Gang Scheduling with a 1ms Timeslice

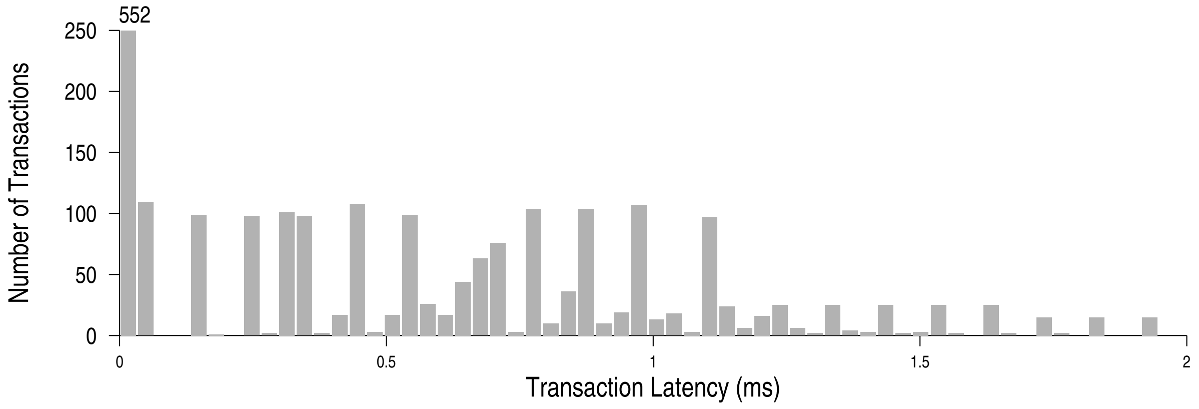


(c) Gang Scheduling with a 10ms Timeslice

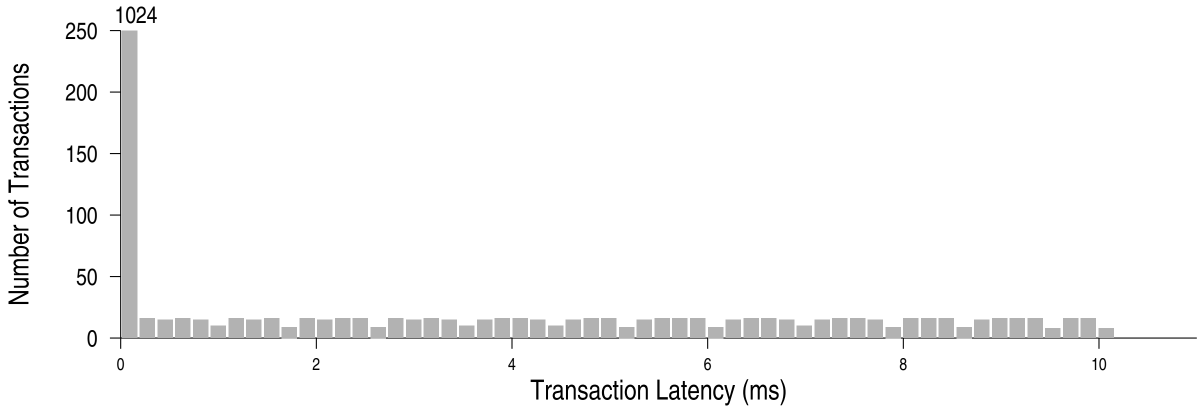
Figure 6.10 Transaction Latency Histograms for Gang Scheduling Policies (Half Utilization)



(a) Gang Scheduling with a 100μs Timeslice

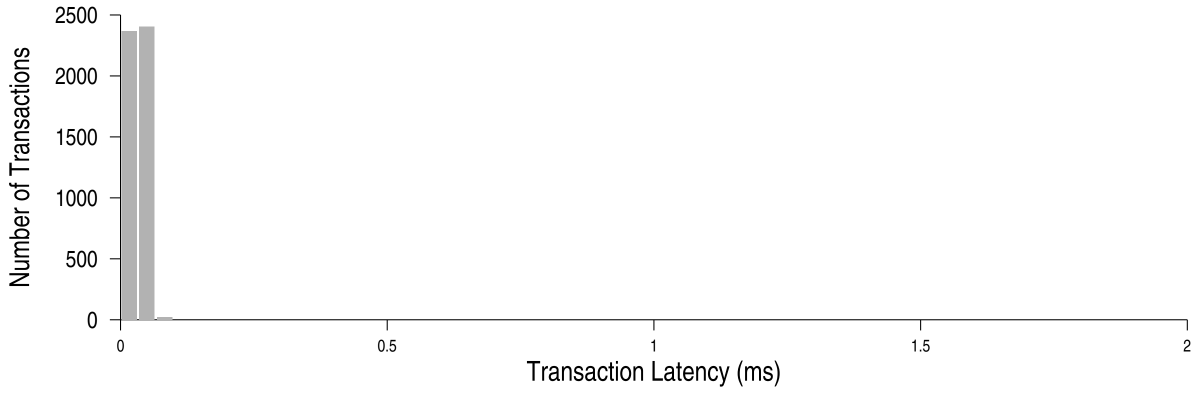


(b) Gang Scheduling with a 1ms Timeslice

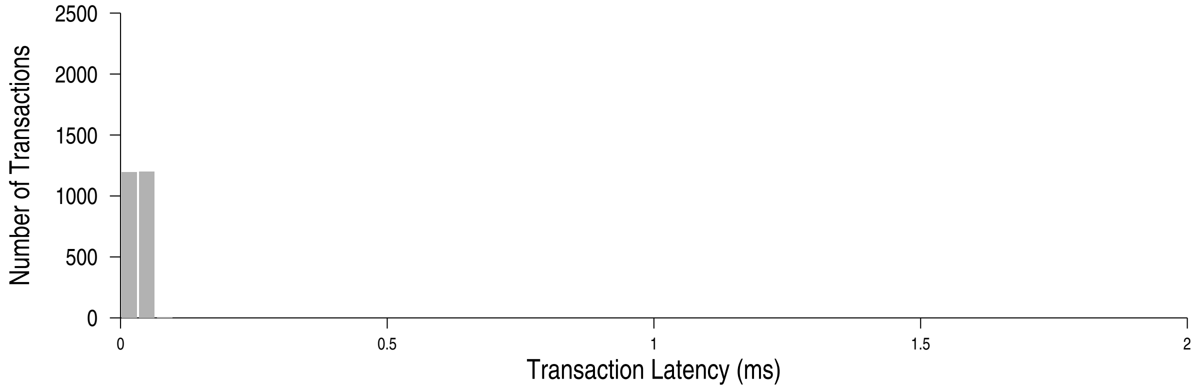


(c) Gang Scheduling with a 10ms Timeslice

Figure 6.11 Transaction Latency Histograms for Gang Scheduling Policies (Quarter Utilization)



(a) Half Utilization



(b) Quarter Utilization

Figure 6.12 Transaction Latency Histograms for Overcommitted and Dynamically Partitioned

For the 1ms timeslice, twice as many transactions finish within the 33μs processing time, even though half as many transactions are executed overall. Transactions are still spread out over the 2ms, but the fraction that are delayed drops by half, and the number that are delayed drops to one quarter. For the 10ms timeslice, slightly fewer transactions finish within 33μs with the quarter utilization as opposed to half utilization experiments, but still nearly twice as large of a fraction (57%) of transactions finish with little to no delay. The remaining 43% are still spread out uniformly until just over 10ms.

Finally, Figure 6.12 shows transaction latency histograms for the overcommitted and dynamic partitioning configuration. Figure 6.12(a) shows the half utilization experiment, and Figure 6.12(b) shows the quarter utilization experiment.

In stark contrast to all of the gang scheduling configurations, nearly all transactions are completed within 67μs of the request's arrival for both utilizations (though fewer transactions are committed for the quarter utilization experiment). Because the guest VM is always running, no transactions need to wait for the VM to be rescheduled. And also because the guest VM is always running, it can sustain the load nearly all the time without transactions being delayed waiting for currently executing transactions.

Like the rest of the overcommitted and dynamically partitioned experiments in this chapter, these experiments use a maximum timeslice of 100μs. Transactions running on, or queued up to run on, a paused VCPU can thus observe additional delays of up to 100μs. This data may be surprising then, since almost no transactions are delayed by that long, even for the half-utilization experiments. The reason is twofold. The first part is that the uniform transaction arrival rate does not cause bursts of transactions to queue up waiting for processing, instead, transactions arrive at just under the rate at which they are processed. The second part of the reason is that when a thread finished processing a transactions, it spins waiting for another transaction if there is not already one on its queue. Since there are rarely transactions already queued for that same thread, the spin is detected and the VCPU executing that thread is descheduled shortly after completing the transaction.

Thus, the dynamic partitioning experiments are easily able to process nearly all transactions shortly after they arrive, providing a transactions latencies much better than gang scheduling with long timeslices, and even provide latencies shorter than gang scheduling with a 100μs timeslice.

6.4 Related Work

Several companies [Armstrong et al., 2005; Sun Microsystems, 1999; Waldspurger, 2002] and academic researchers [Bugnion et al., 1997; Govil et al., 1999, 2000; Whitaker et al., 2002] use software system VMMs to consolidate multiple services, each running its own OS and applications in a guest VM, onto a single physical server. All of these VMMs, with the possible exception of the IBM Power5 hypervisor, require the use of gang scheduling and would potentially benefit from the dynamic partitioning proposed in this chapter.

IBM's Logical Partitioning [Borden et al., 1989] allows the static partitioning of processors, and well as other resources, as depicted in Figure 6.1(b) and discussed in Section 6.1. IBM uses the term *partitions* to refer to both the guest VMs and the resources allocated to those VMs.

Of particular relevance to the proposals in this chapter are other VMMs which enable dynamic partitioning to varying degrees. IBM's Dynamic Logical Partitioning (DLPAR) [Jann et al., 2003], and Sun's Dynamic System Domains [Charlesworth, 1998], both allow physical processors to be moved from one guest VM to another, either manually by the system administrator [Charlesworth, 1998] or automatically as requested by a guest operating system [Jann et al., 2003]. Both systems thus allow the physical processing resource to be portioned between guest VMs and allow the number of processors in those partitions to change over time.

There are two primary differences between these existing systems and the proposed dynamic partitioning enabled by multicore virtualization. First, the proposals in this chapter work with completely unmodified operating systems. IBM and Sun, on the other hand, have each invested heavily in significant OS modification to AIX and Solaris, respectively, to enable dynamic reconfiguration [Charlesworth, 1998; Jann et al., 2003].

Second, the dynamic reconfiguration in both systems is a heavy-weight procedure, involving a complex set of steps to reconfigure the first OS to use fewer processors (similar to Section 5.4), reconfigure the VMM, and then reconfigure the second OS to add the processors [Charlesworth, 1998; Jann et al., 2003]. Each of these steps can involve significant overhead and latency (again, see Section 5.4), meaning that they may be applicable for long-term shifts in user demand, but not to adapt to short term heterogeneity created by power, thermal, or reliability concerns (e.g., Chapters 1 and 5).

Unlike Sun and IBM, but like the proposal in this chapter, Uhlig et al. [2004] allow the cores currently assigned to a single VMs to be overcommitted (as long as the total cores in the system are not overcommitted by a single guest VM). Their paper is similarly motivated by the need to both partition cores among simultaneously executing VMs, and adapt to changes in workload demands. For a discussion of the limitations of their proposal, see Section 4.5.

In a para-virtual environment, the interface is changes to better support virtualization. With para-virtualization, a VMM could be created to allow the cores used by a single guest VM to be overcommitted relative to that VM's VCPUs, and hence gain several additional benefits of the proposed rapid dynamic partitioning. As mentioned in Section 2.1, when operating system and software VMM developers cooperate on a new interface between these components, the OS is in a position to pass useful information directly to the VMM. For example, system such as the Power5 [Armstrong et al., 2005] have an interface that allows an OS to indicate when it is not performing useful work. Such para-virtualized interfaces can enable a software VMM to overcommit the cores used by a single guest OS is much the same way that the proposed multicore virtualization allows, but without the necessity of hardware spin detection. Such interface modifications can allow next-generation software VMMs and operating systems to take advantage of the dynamic partitioning policies proposed in this chapter. We are unaware of software VMMs which actually do this, however.

Numerous studies have examined partitioning and performance isolation of on-chip caches within the context of multiple applications running within a single VM [Chang, 2007; Kim et al., 2004; Rafique et al., 2006; Suh et al., 2004]. Several of these proposals could be applied to the shared L3 cache in the target multicore to further improve performance isolation. These techniques could also be applied to individual L2 caches which are time-shared among multiple VCPUs.

6.5 Chapter Summary

At present, server consolidation a big business, with multiple economic and environmental drivers encouraging consolidation of multiple services onto a single physical host. Maintaining the efficiency objectives of such a server, however, becomes increasingly challenging as the number of VCPUs exposed to each guest VM increases in the multicore era. In particular, the necessity of most existing software VMMs to gang schedule, or co-schedule, the VCPUs of each VM creates conflicting objectives with respect to the efficiency of the server. A gang scheduled configuration can choose to either adapt to changing workload demands, or optimize locality through careful assignment of VCPUs to cores, but not both.

This chapter proposes to utilize the multicore virtualization techniques from Chapter 4, including the basic virtualization and hardware spin detection, to simultaneously permit both of these objectives, while working with unmodified software system VMMs and unmodified, commodity operating systems running within them.

This chapter demonstrates that dynamic partitioning of the cores among guest VMs, while overcommitting the cores within each partition, can be beneficial in three ways. First, it creates more cache, TLB, and branch predictor locality compared to gang scheduling at short timeslices, by dynamically specializing these structures for executing VCPUs from a particular VM. This dynamic heterogeneity leads to significant improvements in both throughput and performance isolation. Second, short effective timeslices does not increase the latency of requests arriving for either VM. Third, by exposing as many VCPUs to each VM as there are total physical cores, the partitions can be dynamically adjusted, quickly and easily, to changes in the demand for the services. Furthermore, the flexibility arising from the ability to assign any subset of VCPUs to any subset of cores, allows the system to quickly adapt to other emerging challenges of future multicores, including power, temperature, or reliability issues.

Chapter 7

Mixed-Mode Multicore Reliability

Technology experts continue to warn about the increasing rates of hardware faults in processor components, due to a variety of transient, intermittent, and permanent sources [Borkar, 2004; Borkar et al., 2003; Bowman et al., 2002; Constantinescu, 2007, 2003; Semiconductor Industry Association, 2005; Shivakumar et al., 2002]. Chapters 2 and 5 provide more details on these trends. Today, certain applications and users already desire high reliability and the peace of mind that comes with it — and are willing to pay extra in terms of performance and machine cost [Aggarwal et al., 2007; Bernick et al., 2005; McEvoy, 1981; Slegel et al., 1999]. If these reliability trends continue, as expected, many more users are likely to need or want extra levels of protection in the future.

A multitude of circuit and microarchitectural techniques have been proposed to tolerate various hardware faults, while preserving the view of continuous, reliable operation that the system and application software have come to expect. One technique in particular, Dual-Modular Redundancy (DMR), can provide very high coverage from many difference sources of faults. Yet when leveraging a previously proposed microarchitectural DMR technique [Smolens et al., 2006], experiments in this chapter demonstrate that applications can observe up to a 50% reduction in the IPC of each thread, and nearly a 4X reduction in overall throughpout.

While a growing number of applications may be be willing to trade reduced throughput and increased latency for reduced risk of hardware errors, other circumstances compel software users to forgo extra levels of reliability in favor of performance. One such example is a media player that can tolerate both visual artifacts due to simple data corruption, and occasional application crashes due to control-flow or other errors. Another example is a consolidated server hosting multiple guest virtual machines (VMs) for multiple customers with different service-level agreements (SLAs). Some customers may require very high reliability (at a premium price). Other customers may be willing to tolerate occasional data corruption and down-time due to crashes, while paying an economy price.

These scenarios result in a system where one set of application, the *reliable* application, need the protection of DMR, while another set, the *performance* applications, need the high performance offered by avoiding DMR. This chapter makes the observation that ***a user may want to run both types of applications on the same machine at the same time***.

To address this need, this chapter proposes a design of a *Mixed-Mode Multicore* (MMM), a system for supporting *mixed-mode reliability*, i.e., the ability to simultaneously execute reliable and performance applications.

The basics of an MMM seem simple: use DMR for reliable applications, and turn off DMR for performance applications. Several architectural DMR proposals suggest that DMR can easily be turned on and off (e.g, [Mukherjee et al., 2002; Walcott et al., 2007]), though they do not actually investigate the issues involved in doing so.

A key contribution of the research in this chapter is that dynamically switching between DMR and non-DMR operation within a single system is not as straightforward as it might first appear. In particular, this chapter observes that 1) care must be taken both during execution, and during a mode switch in order to protect the integrity of the system, and 2) providing a simple interface to software complicates the scheduling of VCPUs to cores. To address the first problem, this chapter proposes to maintain a small amount of redundancy for non-DMR applications by re-validating permission for any store misses to the trusted components of the cache hierarchy. To address the second problem, this chapter proposes to use the multicore virtualization techniques from Chapter 4 to flexibility assign VCPUs to cores.

The final MMM system is able to protect the integrity of reliable applications needing DMR, improve the latency and throughput of performance applications which do not, and preserve a simple interface to the system software.

7.1 Mixed-Mode Objectives and Design Overview

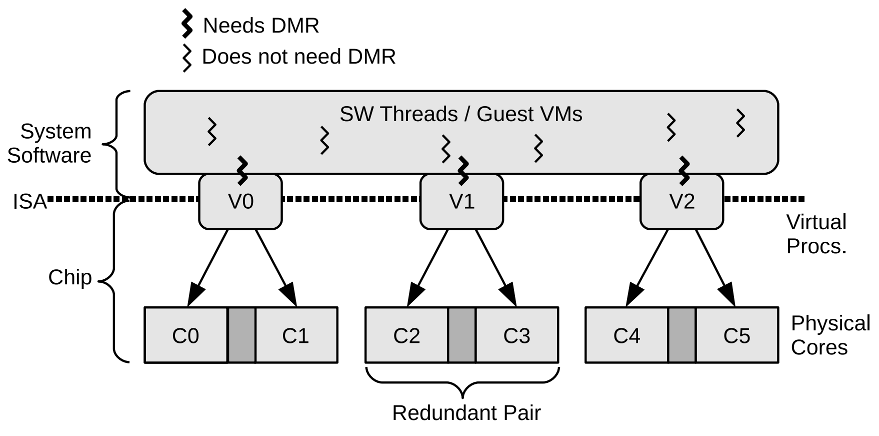
This section begins the discussion of mixed-mode reliability by examining several design objectives and providing an overview of the proposed MMM system. Details of the implementation are provided in Section 7.2. The primary objectives of an MMM are threefold. First an MMM should provide a simple, clean interface to software. Second, an MMM must protect the integrity of the system despite running certain applications with lower reliability. Third, an MMM needs to improve the performance of the system when running less reliable applications — that is the whole reason to operate in mixed-mode, as opposed to always DMR.

The first and third objectives are discussed together in Section 7.1.1, while the second is discussed in Section 7.1.2.

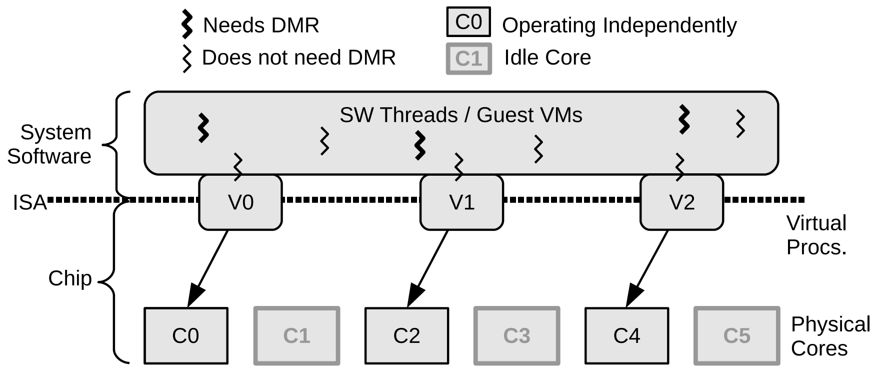
7.1.1 Improving Performance

The reason for implementing an MMM in the first place is to provide reliability for applications that require it, while reducing DMR overheads and improving the performance of for application which do not.

DMR overheads arise primarily from two sources: *throughput* and *latency* overheads. For example, threads running in DMR mode observe additional single thread latency (i.e., loss in IPC) of 35–50% because redundant threads must periodically synchronize execution with each other (Section 7.3.2). When operating in less reliable mode, this latency overhead can be eliminated.



(a) Operating in Dual Modular Redundancy (DMR) Mode



(b) Static Mixed-Mode Improving Latency

Figure 7.1 Techniques for Performing Mixed-Mode Execution

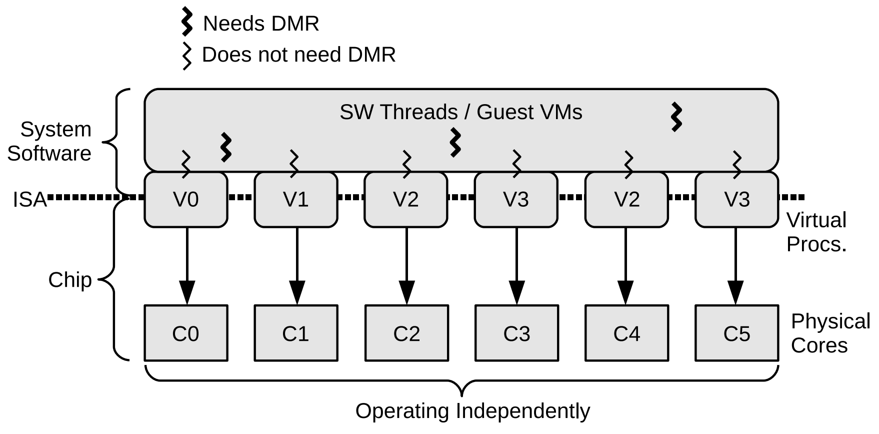
This scenarios is depicted in Figure 7.1 for a 6-core multicore system. Figure 7.1(a) shows an MMM operating in a standard DMR configuration. Three VCPUs are exposed to the OS, each of which is executing on two physical cores joined together to make a redundant pair. The system, as currently used by the OS, acts like an extra-reliable 3-core system.

The OS (or system VMM) dictates whether each VCPU needs extra reliability or not depending on which application threads are scheduled on that VCPU. As threads are rescheduled, the VCPU's need for extra reliability can change dynamically, changing the number of cores required to execute that VCPU. Thus, the OS's scheduling decisions create dynamic heterogeneity in terms of the requirements for executing the software. Figure 7.1(b) shows the same system after the OS has scheduled three threads onto those VCPUs (V0, V1, and V2) that do not need extra reliability. Each VCPU observes $\sim$35–50% improvement in performance since the core continuing to execute that VCPU is not required to synchronize execution with its redundant pair. Statically mapping VCPUs to a pair of redundant cores is a simple way to handle scheduling for an MMM. When DMR mode is enabled for a VCPU, both cores redundantly execute the VCPU. When DMR mode is disabled, one core executes the VCPU, and the other enters an idle state. For a multicore processor, different VCPUs can be executing in different modes at different times.

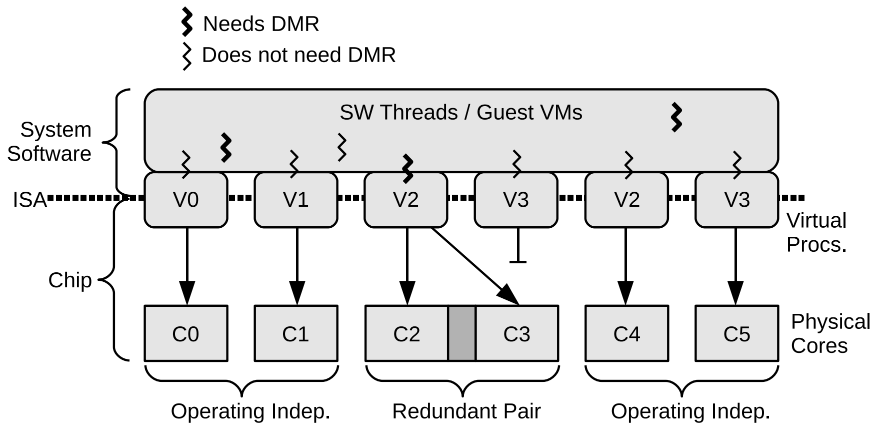
As an additional benefit, when operating in less reliable mode, the extra cores that would be used for executing the redundant threads can be used to execute new threads, improving throughput for scalable applications. To improve throughput, the desired system may look something like Figure 7.2(a), which shows a similar system to Figures 7.1(a) and 7.1(b), except that six VCPUs have been exposed to the operating system, which has scheduled six software threads that do not need the extra reliability of DMR. The six cores each independently execute one of the VCPUs, just like in a standard, non-DMR system. Throughput in this dynamically scheduled MMM system can be improved by nearly 100% compared to Figure 7.1(b), and by 3–4 times Figure 7.1(a) (Section 7.3.2).

Comparing the systems in Figures 7.2(a) and 7.1(b), however, demonstrates a problem: the number of VCPUs that must be exposed to the OS in order to improve throughput increases. Dynamically adjusting the number of VCPUs in the system, however, is a cumbersome process, as discussed in Section 6.4, and shown in Section 5.4.

By using the multicore virtualization techniques in Chapter 4, however, the cores can be overcommitted, such that more VCPUs are exposed to the OS than there are available *pairs* of physical cores. When many VCPUs wish to execute with dual-redundancy, some of them are unable to run.



(a) The Goal: Improving Throughput and Latency



(b) The Solution: Overcommitted Mixed-Mode Execution

Figure 7.2 Improving Throughput in a Mixed-Mode Multicore

But when many VCPUs do not require dual-redundancy, all of the cores can be utilized to increase throughput.

An overcommitted mixed-mode system is depicted in Figure 7.2. Here, one VCPU (V2) requires DMR, and is executing redundantly on cores C2 and C3. V3 is paused since there are no cores available to execute it. Using an overcommitted mixed-mode system allows throughput to be improved compared to a static mixed-mode, while still improving the latency of any non-redundant VCPUs.

Single-OS and Consolidated Server Systems Mixed-mode execution can offer differentiated service to either different applications within a single OS system, or to different VMs within a consolidated server system. In either case, the lower level of system software (i.e, the OS or system VMM, respectively) is responsible for deciding which virtual (or real) processors require redundancy at any given time. Most of the issues with these two systems are the same. Thus, without loss of generality, most of the following discussions focus on a single-OS system.

7.1.2 Protecting System Integrity

When performing mixed-mode reliability, a second key challenge is to protect the integrity of the system while executing a core in the less reliable mode.

During fault-free execution, or when executing with dual redundancy, correct unprivileged (i.e., non-OS) software will not access or modify state for which it does not have the appropriate permission; incorrect software will get caught doing so. However, when executing without dual redundancy, simple hardware faults, such as a bit flip in the *privileged mode* bit, checking logic, or TLB can result in an successful privilege check. Such faults can allow buggy software, or another fault, to corrupt other system state.

While an MMM should allow certain software to execute with lower reliability and higher performance, it should not allow that software to corrupt the state of the system, or corrupt the state of any applications running in DMR mode. Thus, a mechanism is needed to protect critical state when executing without DMR. The overhead and complexity of this mechanism should be small enough for it to be advantageous over always executing in DMR mode.

The three key components of the proposed implementation involve 1) always executing privileged software (i.e., the OS or system VMM) in DMR mode, 2) carefully handling register and cache state during mode-transitions, and 3) re-validating permission for stores when they leave the untrusted private L1 caches. A detailed discussion of these components is provided in Section 7.2.2.

If the integrity of the system cannot be maintained when running less-reliable software, users will enable reliable mode at all times, and forgo any possible performance gains from mixed-mode execution.

7.2 Mixed-Mode Implementation

The previous section outlined the objectives and high-level mixed-mode design. This section presents the implementation details of a mixed-mode multicore (MMM). First, an overview of the Reunion DMR proposal [Smolens, 2008; Smolens et al., 2006], and necessary adaptations for mixed-mode use, are provided. This section then focuses on the other aspects mixed-mode implementation: the mechanisms for protecting system state, and the proposed software interface. The virtualization mechanisms required for an overcommitted MMM have been described, discussed, and evaluated in detail in Chapters 4, 5, and 6.

7.2.1 Reunion Overview and Adaptations

The Reunion proposal [Smolens, 2008; Smolens et al., 2006] was chosen as the base DMR implementation for several reasons. First, it allows any pair of cores to be joined together as one logical DMR pair. Second, it involves only modest hardware changes within each core. Third, it works with a directory-based coherence protocol, and requires only modest changes to the protocol. Finally, it is the only DMR proposal of which I am aware to have investigated and tackled several of the issues involved with using DMR on multithreaded commercial workloads.

Reunion defines a logical processing pair as two cores which redundantly execute the same instruction stream, and are presented to the system software as one logical core. The *vocal* core, i.e., the *master*, implements full coherence, and communicates with other cores and caches in the system as normal. The *mute* core, i.e., the *slave*, loads data from its own private cache hierarchy, but does not expose new values outside of that hierarchy.

An additional in-order pipeline stage, *Check*, is added to each core after execution and before retirement. When entering *Check*, instructions compute a *fingerprint*, or hash of their results, and send these fingerprints to the other core. Each instruction waits in the *Check* stage until it receives the other core's fingerprints for the same instruction. The instructions are then committed to the architected state of each core. Fingerprints capture all outputs, branch targets, and store addresses and values. Multiple instructions can update a single fingerprint to reduce bandwidth. Fingerprints are communicated to the other core via either the normal on-chip network, or a dedicated fingerprint network. Each core independently detects a fingerprint mismatch, flushes its pipeline, and proceeds through a re-execution protocol described below.

Because load values are not strictly replicated, mute and vocal cores can each read different, but correct, sequentially consistent values from the memory hierarchy. This phenomenon is called *input incoherence*, and triggers the detection of an error. In addition, a mute core is not actually required to maintain coherence with the rest of the system. Instead, all requests emanating from the private cache hierarchy of a mute core are labeled as *phantom* requests, which do not change the state of the line in the directory or any other caches, but for which the cache hierarchy makes a best-effort attempt to provide the correct value. Should that attempt fail, input coherence and a fingerprint mismatch will result, which will be detected and corrected. Using a best-effort approach, input incoherence is reported to occur very infrequently: less than 1.5 events per million instructions for 7 of 8 workloads, and 21 events per million for the 8th (DB2 DSS Q1) [Smolens et al., 2006]. A discussion of mute coherence and mixed-mode execution is presented in Section 7.2.1.1.

In order to guarantee forward progress from reoccurring input incoherence, a re-execution protocol is proposed. This protocol involves each core non-speculatively single-stepping execution up to the first memory request. A synchronizing memory request is then sent from both cores to the on-chip directory, which obtains a coherent copy of the block on behalf of both cores and sends the copy to each core. Each core is then guaranteed to load the same value (in the absence of a hardware fault).

Certain errors, such as fingerprint aliasing, cannot always be detected before the mute core has corrupted its architectural state. For these cases, the vocal core is responsible for maintaining the "safe" register state from which both cores recover when such a fault is detected by a mismatch on subsequent instructions.

Relaxed-input replication allows vocal and mute threads to be scheduled on any cores, but they have the constraint of being co-scheduled.

7.2.1.1 Mute Coherence

Reunion suggests that the private caches of mute cores need not, and should not, maintain coherence with respect to the rest of the system. When operating in mixed-mode, however, a core switching from operating as a mute core of a DMR pair to operating independently to run another VCPU faces private L1 and L2 caches with incoherent data. These caches must then be flushed on a mode switch.

If mode switches are very infrequent the cost of flushing the cache can be amortized when switching from a VCPU that requires DMR to one that does not. Such a scenario can arise for a gang-scheduled consolidated server providing differentiated service to two VMs. It the timeslice is long, the new guest VM is likely to evict most of the contents of the cache during its timeslice anyway.

When mode switches are frequent, however, the cost of flushing the cache can become prohibitive. This scenario can arise when performing mixed-mode on a single-OS system, where traps into the OS require a switch to reliable mode. As an optimization for mixed-mode operation with frequent switching between DMR and non-DMR mode, private caches of a mute core can be kept coherent. A read from a mute core returns a line whose data and permission are coherent with the rest of the system. A write from a mute core invalidates the line in the private hierarchy if present, without writing the data. A subsequent read to a written line will cause a coherence miss, which will return the correct, coherent data from the vocal core.

Such an optimization improves the cost of switching cores, but comes at the price of more coherence for load-modify-store operations on the vocal core: A load miss for an unshared line in this case should not (cannot) be returned in *Exclusive* state, since the mute core will soon (or has already) acquired a *Shared* copy of the line. The following store from the vocal core must now acquire *Exclusive* permission before writing. This coherence miss on the vocal core must at least travel another round-trip to the directory to acquire permission. The vocal's store may not require a 3-hop transfer all the time, however. The reason is that when the mute core's store arrives at its own L2, it will automatically invalidate the line and notify the directory that it no longer owns a copy. Should the mute's invalidation happen before or during the vocal's coherence miss, some or all of the 3-hop latency will be overlapped.

With either option, another issue arises when performing loads and stores of VCPU state during a VCPU context switch or mode switch. These requests from a mute core must be processed as normal, regardless of whether the mute maintains coherence for other memory requests, or implements Reunion's mute incoherence policy. This means that the cache at a mute core can simultaneously consist of both incoherent lines brought into the cache via phantom mute requests, and lines (containing VCPU state) which are coherent with the system. A bit is added to the state field of the each line indicating whether or not the line is coherent with the system.

As a result of mixing coherent and non-coherent lines, flushing lines during normal Reunion mode is not as simple as gang-invalidating the cache. Instead, lines must be inspected one by one to see if they are dirty and need to be written back. This operation proceeds at cache request bandwidth.

7.2.1.2 Target Multicore Assumptions

It should be noted that little attempt was made to optimize the cache and core configurations of the target multicore used elsewhere in this dissertation for use with Reunion or mixed-mode. In particular, the 3-level cache hierarchy adds complexity and overhead to DMR operation, especially for an MMM. Had such reconfiguration been performed, several issues relating to mixed-mode and DMR would have been simpler. On the other hand, the chosen path of taking a given design and determining what changes need to be made may better reflect the approach often undertaken in industry.

Two changes to the target multicore were made, however. First, the L1 caches are assumed to be write-through, non-allocate, not write-back as in the previous chapters. This change facilitates switching of modes without flushing the untrusted L1 caches. Second, the multicore is assumed to have 16 physical cores to facilitate executing in DMR mode with the same 8-processor workloads used elsewhere in this dissertation.

As with Chapter 5, shared components of the memory hierarchy are assumed to implement other reliability techniques. Unlike Chapter 5, however, the L1 caches are not assumed to be reliable. While in DMR mode, accesses to the private L2s are replicated,[1] but any coherent line placed in the L2s must remain coherent and uncorrupted. That said, accesses to all parts of the cache, and even off-chip memory, could be replicated if the need arose.

[1]Accesses to VCPU state are not replicated to the *same* address, as are accesses to normal program state, but are instead redundantly performed to *different* addresses.

7.2.2 Protecting System Integrity

When performing mixed-mode reliability, a key challenge is to protect the integrity of the system while executing a core in the less reliable mode. When executing with dual redundancy, correct unprivileged (i.e., non-OS) software will not access or modify state for which it does not have the appropriate permission; incorrect software will get caught doing so. However, when executing without dual redundancy, simple hardware faults, such as a flip in the *privileged mode* bit, checking logic, or TLB can result in a successful privilege check. This can allow buggy software to corrupt other system state. Thus, a mechanism is needed to protect critical state when executing without dual redundancy.

7.2.2.1 Protecting System Software

The first step to protect the integrity of the system is to ensure that the system software itself is not corrupted. All privileged state is conservatively assumed to be critical. The result of this observation leads to the policy of always executing privileged software in reliable mode. This privileged software may be the OS in a single-OS system, or the software VMM or hypervisor in a consolidated server system.

As a consequence, a user application executing in less reliable mode must switch to reliable, DMR mode before executing the OS code. For many commercial applications, system calls occur as frequently as every 10-100k instructions. Thus the latency of switching modes can become significant, as we will see in Section 7.3.3. A core in performance mode cannot execute any privileged instructions without causing a transition to reliable mode.

A second consequence of this policy is that applications, such as Apache, which spend 80–90% of their time in the OS, are unlikely to observe significant savings when operating on a single-OS system. On a consolidated server, however, a guest VM running Apache may still see improvements, since the guest OS running in the VM does not require reliability mode. The system VMM or hypervisor must still excecute in reliable mode, but does not typically execute as frequently as does the guest OS.

7.2.2.2 Protecting Memory

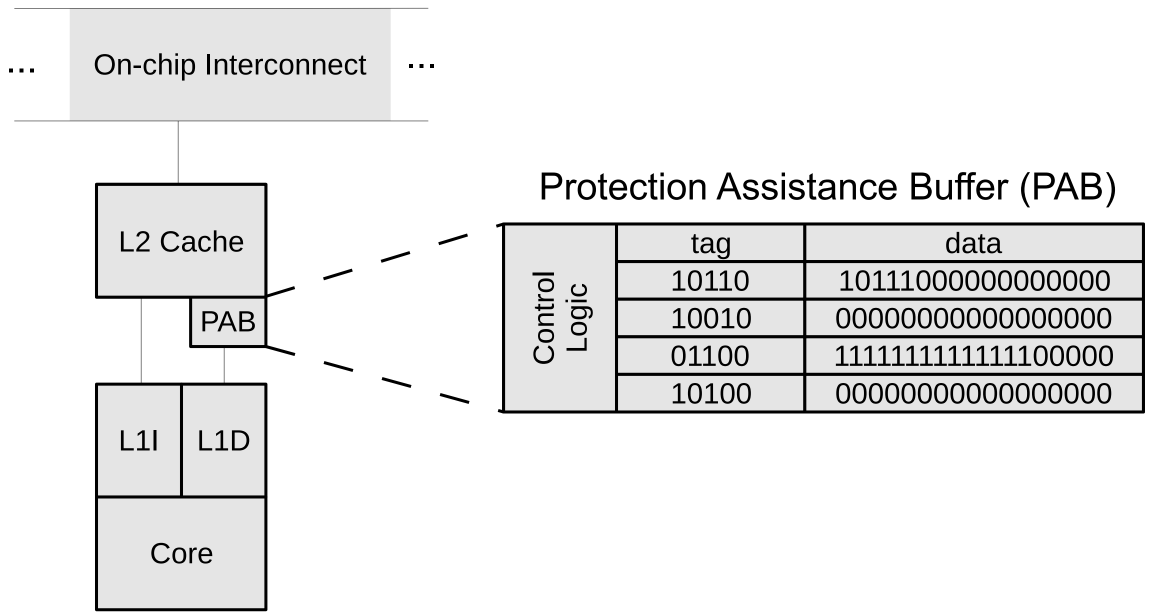
The TLB maintains sufficient information to prevent any user application from illegally accessing memory state. However a hardware fault in the TLB array, checking logic, privileged registers, or L1 cache can allow such an access.

One option may be to harden these structures by making the transistors larger, slower, and/or liberally applying other circuit-level reliability techniques. The problem is that all of these structures are on the critical load path, and hardening could impact the latency of every memory access. If hardening is not an option, then an MMM will require a separate mechanism to prevent against illegal memory accesses.

To prevent arbitrary software from accessing memory for which it does not have permission, this section proposes to duplicate the TLB protection check using a separate structure, called the *Protection Assistance Buffer* (PAB). The PAB is placed between the private L1 caches and the private L2 cache. When the core is executing in non-DMR mode, the permission of each L1 write-through is rechecked, either before or in parallel with the L2 access.

For non-malicious code, it is sufficient to only prevent erroneous stores from illegally writing memory. For security reasons, preventing erroneous fetches and loads from reading illegal memory addresses may also be required, but is not addressed in this work.

The PAB is a physically tagged and indexed structure whose entries contain one bit. This bit indicates whether code operating in less reliable mode is allowed to access this page. The implication of only using one bit, instead of a larger address-space identifier, is that less reliable software can potentially corrupt the memory of other less reliable software, but not that of the OS



tag	data
10110	10111000000000000
10010	00000000000000000
01100	11111111111100000
10100	00000000000000000

Figure 7.3 The Structure of the Protection Assistance Buffer (PAB)

or software operating in reliable mode. The PAB also contains an *active/inactive* control bit, and some simple control logic.

Figure 7.3 shows a diagram of PAB placement and structure for one core in the system. It is organized much like a cache, with a tag array and data array containing protection information.

The PAB operates on physical addresses, since the virtual address is assumed to no longer be known for a store accessing the L2 cache. Because the tagging and indexing of the PAB thus differs from that of the TLB, the PAB and TLB can experience misses at different times. This fact has an important consequence: there is no benefit to a PAB organization and structure similar to that of the TLB. In particular, the need for a $\sim$20-bit tag per 1-bit entry disappears. Instead, each entry of the proposed PAB contains a 64-byte, cache-line sized array of bits, representing protection information for a total of 4MBytes worth of physical memory. A valid bit is also needed for each line. For a 43-bit address space with 8KByte (2^{13}) pages, like the UltraSPARC IIICu, a 128-entry (2^7), direct-mapped PAB, with each 64-byte entry mapping 512 (2^9) pages, thus requires 14-bit tags plus 1-bit for state. This results in a tag overhead of less than 3%. A 128-entry PAB requires 8.2KBytes, can map 512MBytes of physical memory, and represents a storage overhead of 1.6% compared to the private L2 cache.

Assuming an acceptably low probability of faults occurring to both the PAB and TLB, the PAB does not need any special hardening or ECC protection. This is because the PAB's operation is checked by the TLB, and vice-versa. However, using ECC, or some other circuit reliability mechanism in one or both structures can reduce the number of false positive fault detections.

Basic PAB Operation When in DMR mode, the PAB in deactivated, since all address translation and permission verification is already redundantly performed. All accesses bypass the structure and access the L2 cache as normal. When not in DMR mode, the PAB is activated, and stores (i.e., write-throughs from the L1) examine the PAB in parallel with their access to the L2 cache. Should the PAB indicate an invalid store, the L2 access is aborted.

A parallel lookup and transaction abort adds complexity to the L2 cache controller. Waiting to access the L2 cache until the store is validated by the PAB is also possible. This serial lookup incurs additional latency for stores, but due to the small size of the PAB, the latency is expected to be small. Experiments in Section 7.3 evaluate both parallel and serial lookups.

Loads and fetches bypass the PAB.

PAB Misses The PAB is updated through a memory structure called the *Protection Assistance Table* (PAT). The PAT is similar to an inverse page table: for each physical page currently mapped by the system, a "1" entry indicates that page can only be accessed by DMR applications, and a "0" entry indicates that page can potentially be accessed by any software. At one bit per 8KByte page, the PAT thus requires $\frac{1}{2^{16}}$ the space of physical memory, or 16MBytes for one TBytes of physical memory.

The PAT resides in cacheable memory. On a PAB miss, the PAB simply sends a request to local L2 cache requesting the appropriate PAT line. Since stores are prefetched before retirement, the PAB miss and possible L2 cache miss can be prefetched in parallel before the actual write requested is made.

The PAB is kept coherent during a TLB demap operation, even when the PAB is deactivated. On a demap, the TLB sends the physical page address of the demapped page to the PAB, which invalidates the corresponding entry.

System software is responsible for maintaining the PAT. It must set aside physical memory for the PAT, and update the entries when it updates its page table.

7.2.3 Protecting Registers

Unprivileged software is not allowed write most privileged registers. However, a fault can cause unprivileged software to corrupt a privileged register, or erroneously allow buggy of malicious software to write one of these registers.

To protect against such faults the mixed-mode framework simply copies their contents to reserved physical memory before beginning non-redundant execution. When redundant execution is re-entered, the current contents of those register are compared with the saved copy to detect a fault. If differences are detected, one or more faults must have occurred. A (non-precise) exception can be then be triggered to notify system software.

7.2.4 Handling Mode Transitions

Each core contains a small hardware state machine to handle mode transitions. Each core communicates with the centralized Virtualization Controller, described in Chapter 4, which is also implemented as a hardware state machine. On a transition between performance and reliable mode, and when context switching VCPUs, these hardware state machines cooperate to perform the steps outlined below, and detailed in Table 7.1.

At a high level, all reliability and VCPU transitions involve two steps: switch *out* the currently executing VCPU (if any), and then switch *in* the next scheduled VCPU (if any). For the basic multicore virtualization proposed in Chapter 4, cores go through two states when switching out one VCPU, and only one state when switching in another. These states are shown in the top two boxes of Table 7.1.

Switching to and from reliable mode is similar in concept, especially for the mute core: when entering reliable mode, it needs to acquire the necessary VCPU state in order to switch in and start executing a VCPU, and when leaving reliable mode, it needs to save that state to be loaded later.

The details, however, are more involved. State transitions are also different depending on whether the cores are switching in new VCPUs that have a different reliability requirement, or whether they are switching to/from reliable mode while executing the same VCPU (e.g., due to a system call).

If a core is switching to a new VCPU, but *not* switching reliability modes, then it performs the same basic switch out as in Chapter 4. In reliability mode, both cores perform the same actions, though they use different addresses to duplicate VCPU state.

If a core is switching to a new VCPU, and switching modes, since the reliability needs of the new VCPU differ from the current one, then the state transitions in the middle boxes of Table 7.1 are used. Compared to the basic switch transition, the primary difference is additional synchronization between vocal and mute cores, and the need for the mute core to flush non-coherent data from its L2 cache and write back any coherent data (i.e., VCPU state).

If a pair of cores is leaving DMR modes because of the program returning from a system call, for example, the transitions in the lower boxes of Table 7.1 are used. Leaving reliability mode is similar to the previous case, except that the cores need only store their privileged VCPU state to the cache hierarchy. Entering DMR, however, is substantially different. The vocal core (previously running in performance mode) already has all of the necessary state, but the mute core does not. In addition to simply loading the state, however, the mute core also needs to verify that the contents of privileged registers are the same as they were when last in reliable mode. This check is necessary to prevent faults from corrupting privileged state during performance mode.

Frequent synchronization is necessary during these processes, especially when entering DMR, because these steps can be initiated at different times for the vocal and mute cores depending on what each core was previously doing. If the core was idle, it can take action immediately. If the core was previously executing some other VCPU, then the process of switching out the previous VCPU must conclude before switching in another.

These mode transition protocols, as described, are believed to be sufficient to further the goals of an MMM. No claim of optimality is made regarding these protocols, however.

7.2.5 Software Interface

This work proposes to implement the reliability mechanisms in a thin virtual machine layer beneath the ISA. The chip exposes three things to software via the ISA. First, the interface exposes that the chip has multiple operating modes with different levels of reliability. Ideally, a chip might expose maximum FIT rates for these different levels of reliability as well, though in practice, companies may be reluctant to do so. Second, the interface exposes that choosing to operate in the most reliable mode comes at a price in terms of performance and power. Third, the interface exposes the fact that software is responsible for determining the desired mode, and can do so dynamically on a VCPU by VCPU basis.

This chapter assumes that the reliable mode is implemented via Dual-Modular Redundancy (DMR), while the less reliable mode uses only a single core, but may implement numerous circuit or microarchitectural techniques within that core to maintain reasonable levels of reliability.

Similar to the virtualization mechanisms in Chapter 4, the chip exposes a certain number of virtual processors (VCPUs) to the operating system (OS), and is responsible for mapping the VCPUs to the physical cores. Software is simply responsible for determining when reliability is necessary depending upon what software is running on each VCPU. The number of cores required to execute each VCPU changes dynamically depending on the whims of the OS scheduler, creating dynamic heterogeneity in terms of the requirements of the software.

The basis of the mixed-mode hardware/software interface is a single register per VCPU specifying whether reliability is needed or not. This 2-bit register specifies one of three modes: 1) operate with high reliability, 2) operate with lower reliability and higher performance, or 3) operate with lower reliability only when executing non-privileged (user) software.

When the privileged software is about to context switch to a application or VM which requires less reliability, it can write this register to indicate that the software running on that virtual processor need not be reliable. The hardware implementation then has the opportunity to run with less reliability.

This register is only writable by privileged software. It is expected that most commonly, an individual application will run from start to finish with either lower or higher reliability.

An MMM could leave the management of reliability to the system software, however, the choice of pushing the responsibility down to the chip has several desirable properties similar to Section 5.3.4. Most importantly, as will be shown in Section 7.3.3, the need to switch modes occurs much more frequently than an OS would be able to maintain. The chip, on the other hand, can implement certain functionality in hardware to reduce the overhead. Second, changes to the interface are minimal, allowing multiple OSs from multiple vendors to easily take advantage of the new functionality, while unmodified software can just default to one mode or the other. Finally, by abstracting the details of hardware reliability from the software, those details more easily change with every generation of hardware without requiring interface or software modifications.

Requiring all privileged software to run with reliability greatly reduces the burden of correctness on OS software. However, this technique does require that the OS (or hypervisor) be able to tolerate invalid input across its defined interfaces (such as system calls). This is because a application that sustains faults can modify data (such as the length of an array) that is passed as an argument to these calls. Similarly, if a DMR and non-DMR application memory-map any of the same pages, the DMR application must be able to handle bogus data in that page.

System software is also required to maintain the PAT in a memory location agreed upon by the software and hardware.

7.3 Evaluation

To demonstrate the effectiveness of mixed-mode reliability, several experiments were performed to demonstrate and examine the overheads of DMR, the overheads and frequency of mode transitions in an MMM, and the overall performance improvement resulting from mixed-mode operation.

7.3.1 Methodology

Details of the workloads, target multicore configurations, basic methodology are provided in Chapter 3. In this chapter, some workloads use a single guest VM with 16-VCPUs. In order to avoid the problems comparing separate 16-processor and 8-processor checkpoints (as described in Section 5.4.1.1), these single guest VMs are emulated by combining two 8-processor checkpoints. All experiments are run for 90 million cycles.

Timing First Simulation Unfortunately, using MAI in the traditional manner, as described in Section 3.1, imparts the functional simulator's limitations onto the timing simulator (e.g., restrictions on which instructions are allowed to execute when). For this reason, a second simulator, *ms2sim-tf*, was developed in order to perform experiments with Dual-Modular Redundancy (DMR).

The timing component of *ms2sim-tf* is nearly identical to *ms2sim*, however, the use of Simics MAI is greatly changed. Similar to TFSim [Mauer et al., 2002], *ms2sim-tf* executes each instruction, with the appropriate timings, and then at commit stage, calls Simics MAI to execute the entire instruction all at once, as a *shadow* processor, and compares the outputs. Unlike TFSim, however, *ms2sim-tf* still uses Simics to functionally execute most instructions. But instead of stepping each instruction through the pipeline as dictated by the timing simulator as in normal MAI usage, *ms2sim-tf* steps each instruction up to the appropriate stage, and then tells Simics to squash the instruction before it can change visible program state. In order to do so, *ms2sim-tf* must properly handle dependencies and the register file, supply input values to each instruction, and copy output values before it is squashed. Instructions which cannot be squashed after execute are not executed by Simics until commit. Instead they are either executed within *ms2sim-tf*, or have their output copied from Simics at commit time. Although Simics executes each instruction multiple times, *ms2sim-tf* incurs little additional overhead over *ms2sim*, likely due to the fact that Simics only ever observes one instruction in the window at a time.

Implementing Reunion Dual Modular Redundancy (DMR) is implemented in *ms2sim-tf* by allowing the timing-first components of two cores to execute instructions on, and verify their committed instructions with, one Simics-visible "shadow" processor. The timing components of the two cores compare fingerprints before the vocal core checks the instruction with the Simics shadow.

A dedicated fingerprint network with a 10-cycle latency is assumed, as was done in the original Reunion proposal [Smolens et al., 2006]. Sync requests are not implemented through L2 directory protocol modifications, but rather through direct messages sent from the vocal to the mute core.

The first set of experiments in Section 7.3.4 assumes a parallel PAB and L2 access. A PAB latency of two cycles is used for experiments in Section 7.3.4.1

Limitations of the Simulation Model Timing differences between the vocal and mute cores can create *input incoherence*, as described in Section 7.2.1.1, when they direct the functional simulator to read memory at different times. But since the simulated memory hierarchy is not a functional model, the mute core is not guaranteed to observe the proper data value. This limitation is not expected to affect results significantly, however, since Smolens et al. [2006] report less than 1.5 input incoherence events per million instructions for all but one workload when using best-effort ("Global") coherence for mute requests, as done for this work.

Timing first simulation has another related limitation: even if the vocal and mute cores load the same, sequentially consistent, value from memory, the functional "shadow" simulator can receive a different sequentially consistent value when it executes the instruction as the vocal cores commits. Such an error causes both cores to synchronize, flush their pipelines, copy all VCPU state from the shadow CPU, and return to normal operation. These errors occur up to once per million instructions for the baseline non-DMR system, and twice per million instructions for the DMR configurations.

Misses in the PAB are not modeled, since a 128-entry PAB maps one quarter of the 4GByte physical memory of the simulated machine, which is substantially more memory than can be accessed during a simulation. In addition, the PAB is only accessed by non-DMR applications, which can consume only part of the available memory. Finally, accesses and misses to the PAB can only occur for L1 write-throughs. Since stores are prefetched before retirement, PAB misses

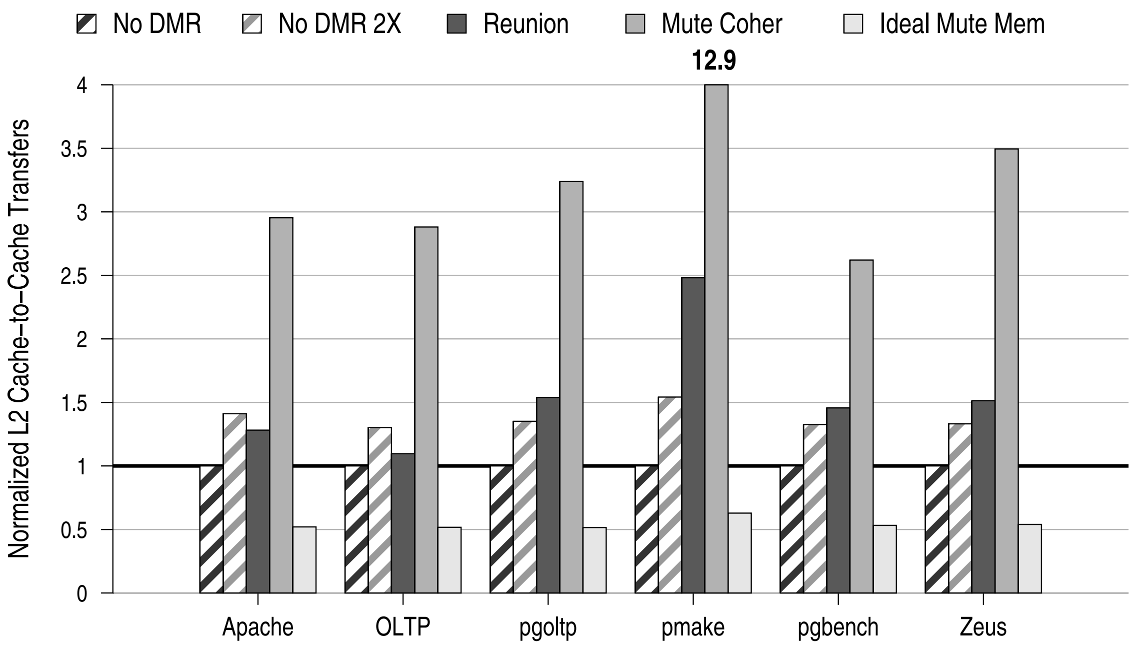


Figure 7.4 DMR L2 Cache-to-Cache Transfers

and any L2 cache misses can operate in parallel before the actual store request is made. Thus, little if any overhead from misses to this structure is expected for the experimental setup in this chapter.

7.3.2 Overhead of Dual Redundancy

This section compares the overheads of a Reunion-like, always-DMR system which performs no mode switching, to a baseline system without DMR. Overheads arise from L2 cache-to-cache transfers and mute coherence, serializing instructions, and instruction window pressure. These three sources of overhead are each examined in detail, followed by a look at the overall DMR performance impact.

Cache-to-Cache Transfers Figure 7.4 shows the number of L2 cache-to-cache (C2C) transfers *per core*. Data are normalized to the baseline non-DMR configuration and account for differences in the amount of work performed. The first bar, *No DMR* represents a non-DMR system running the eight VCPUs on only eight cores. The other eight cores are idle. The second bar, *No DMR 2X* represents a non-DMR using all 16 cores for running independent VCPUs. With these two non-DMR systems, three DMR systems are compared, each of which is running the same eight VCPUs as *No DMR*, but running them redundantly across all 16 cores. The third bar, *Reunion* is for the best-effort re-implementation of Reunion. The fourth, *Mute Coher*, is the same as *Reunion*, except that mute cores maintain coherence as a potential optimization for mixed-mode. Finally, the rightmost bar, *Ideal Mute Mem*, is again similar to *Reunion*, except that all memory accesses from mute cores complete in zero cycles.

For the idealized DMR configuration, we might expect approximately half the number of C2C transfers as the baseline, since only half of the active cores access the cache hierarchy (mute core accesses are idealized). This is indeed the case for every benchmark.

For the *No DMR 2X*, as well as the Reunion, configurations, we would expect the number of C2C transfers per core to stay the same. The reason is these schemes both have twice as many active cores, yet cores are either independently accessing approximately twice as much total data as the baseline, or redundantly accessing approximately the same total data as the baseline. Because mute cores do not maintain coherence, the vocal cores should observe almost exactly the same L2 behavior as the baseline. The reason is that even if a mute core acquires an incoherent "exclusive" copy of a line before the vocal core does, the directory is not updated to reflect this information. Thus, the vocal core will still receive its copy of the line from another L2, the shared L3, or main memory just like in the baseline case. Since both redundant cores are accessing the same data at roughly the same time, doubling the number of C2C transfers should result in a similar number of off-chip misses and similar performance as the baseline (in the absence of other overheads).

Experiments show, however, that the number of C2C misses for both *No DMR 2X* and Reunion increases by 7–250%. The reason *No DMR 2x* observes an increase is because twice as many cores have active data in their respective L2 caches, which means that data is not held in the exclusive shared L3.

The reason DMR observes an increase is that when the vocal core acquires the line first from any source, the mute core's later request is likely to receive it via a C2C transfer from the vocal core.

These additional C2C transfers are one source of overhead for Reunion (since in the target multicore, a 3-hop C2C transfer incurs additional latency compared to a 2-hop L3 hit). These additional transfers occur primarily due to the use an exclusive shared cache. An inclusive shared cache would still see these C2C transfers for data that the vocal core loads in *exclusive* state, but not for *shared* data with a copy still in an inclusive L3 cache.

The effect of prefetching main-memory misses with Reunion is negligible, since neither core runs significantly ahead of the other due to frequent serialization (see below).

When maintaining mute coherence (the third bar) L2 C2C transfers increase by 2.6–3.5 time for most benchmarks, and by a whopping 12.9 times for `pmake`. As described in Section 7.2.1.1, the reason for these additional transfers is that on every load-modify-store sequence, the vocal core must invalidate the shared (coherent) line just loaded by the mute core. These transfers create additional runtime overhead for maintaining mute coherence, but may be offset by eliminating the need to flush the cache when performing a mode switch during mixed-mode execution.

The reason the relative increase in cache-to-cache transfers is so dramatic for `pmake` is that it has lower baseline L2 miss rates than the other benchmarks. The absolute increase in similar to the other benchmarks.

Serializing Instructions OS-intensive workloads, like most of the commercial workloads studied in this dissertation, typically encounter frequent *Serializing Instructions* (SIs), such as those that write control registers [Wells and Sohi, 2008]. Due to their complex dependencies, executing SIs out-of-order (OoO) can be difficult. Instead, these instruction are often implemented in real processors by draining the pipeline and executing the SI as the only instruction in the window [Compaq Computer Corp., 2000; Intel Corporation, 2007; Sun Microsystems, Inc., 2003].

The instructions can cause a significant (3–17%) performance bottleneck with OoO processors [Wells and Sohi, 2008]. With Reunion the impact is even worse for two reasons. First, instructions younger than an SI must be committed before the SI executes, but the *Check* stage incurs additional latency due to the delay of communicating fingerprints between cores (the *comparison latency*). Then, the SI itself must be validated before younger instructions can enter the pipeline, incurring an extra fingerprint transmission and comparison latency delay.

Smolens et al. [2006] report a performance loss of up to 28% with a 40-cycle comparison latency, or approximately 10% with the 10-cycle latency they use as a default. The Flexus simulator user for their study [Wenisch and Wunderlich, 2005], does serialize writes to register-mapped ASIs [Smolens et al., 2006], but does not serialize of all of the register writes considered by Wells and Sohi [2008] and this dissertation to be serializing [Wenisch, 2008]. Thus, the performance impact of SIs reported by Smolens et al. [2006] is optimistic compared to the infrastructure used for this dissertation. It is not clear which more closely approximates a real microprocessor.

Table 7.2 examines the frequency and impact of serializing instructions (SIs) on the non-DMR baseline and the Reunion configuration. The second column show the number of SIs encountered per thousand committed (user and OS) instructions. This data closely matches serializing register writes presented by Wells and Sohi [2008] for similar workloads running on a uniprocessor.

SIs are implemented by flushing younger instructions from the window and blocking fetch until the SI executes as the only instruction in the window. The right two columns of Table 7.2

present the fraction of cycles that fetch is stalled due to an SI in the window. While SIs stall fetch for a significant fraction (10-42%) of cycles in the baseline, this number grows to an even larger value of 16–49% when using Reunion. Certainly, fetch stall does not necessarily imply that something useful could be fetched, but the stall does prevent the opportunity to try.

The other two DMR configurations observe nearly identical SI behavior compared to Reunion, and are not shown.

Due to limitation of Simics MAI, and of the methodology for performing timing-first simulated used in this chapter, it is not possible to evaluate the performance of a hypothetical *SI free* configuration.

Instruction Window Utilization The third overhead affecting DMR execution is capacity pressure on the instruction window and load/store queue (LSQ). This pressure arises primarily from two sources: 1) the requirement that instructions wait in the *Check* stage before releasing their instruction window resources, and 2) the use of sequential consistency (SC).

Table 7.3 shows the fraction of cycles either the instruction window or LSQ are full for the baseline non-DMR system, and each of the three DMR configuration. Reunion observes full structures for approximately 4–15% more cycles than the baseline. The other DMR configurations are similar. `pmake`, with the highest IPC and fewest SIs, sees the largest impact, where these structures are full more than 50% of the time for all DMR configurations.

The original reunion evaluation does not report the impact of this additional pressure, but this impact is likely to be larger for the evaluation in this chapter because a 128-entry instruction window is used instead of a 256-entry window.

The impact in this dissertation's evaluation is also likely larger because the target multicore maintains sequential consistency (SC), meaning that stores are only retired after they are written to the cache in-order. The original Reunion evaluation assumes a *Total Store Order* (TSO) memory

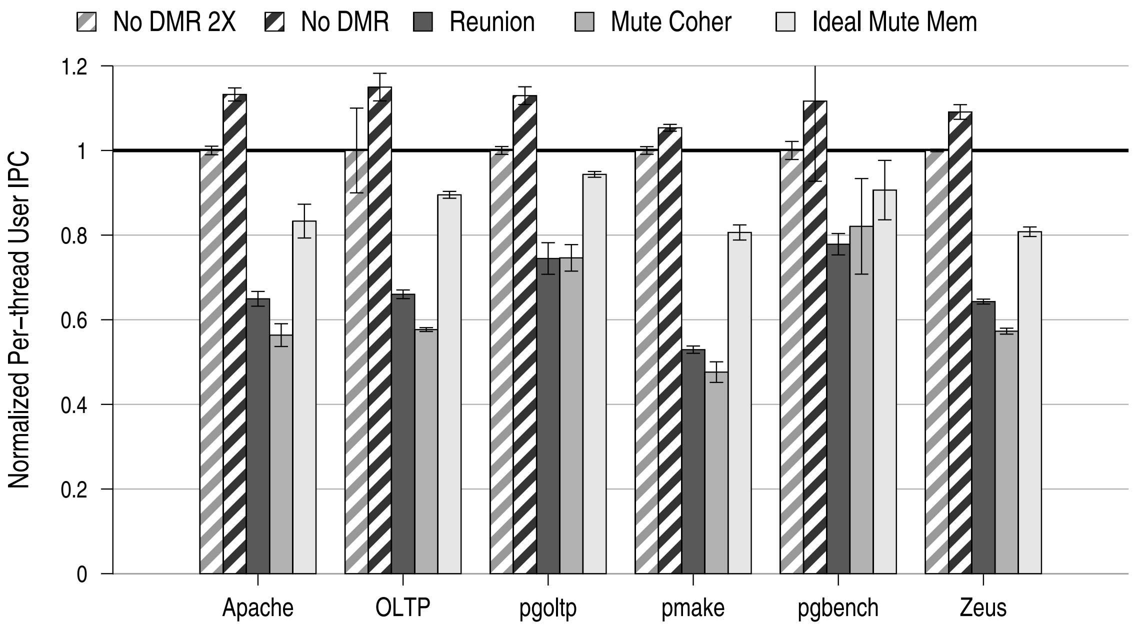


Figure 7.5 DMR Performance Comparison: Single Thread Latency

consistency model. Compared to TSO, SC prevents instructions younger than a store from retiring before that store, keeping them in the window and LSQ longer. Like serializing instructions, when these structures are full, fetch is stalled, hurting performance. TSO would allow those stores to be moved into a store buffer, after verification, but before waiting on any cache misses.

Overall DMR Performance Impact Given these sources of overhead examined above, Figures 7.5 and 7.6 examine the performance overheads of DMR. Figure 7.5 shows the per-thread latency impact, and Figure 7.6 shows the overall throughput impact. The two non-DMR configurations and three Reunion-like configurations are again represented. Note that, in contrast to Figure 7.4, performance data are normalized to the *No DMR 2X* configuration, which is utilizing all 16 cores like the DMR configurations.

In Figure 7.5, per-thread latency is measured as the average of each active VCPU's *User IPC*, as described in Section 3.4. The *No DMR* configuration, running only 8 VCPUs, observes 6–15% higher IPC than the *No DMR 2X* configuration, since it has approximately half of the bandwidth

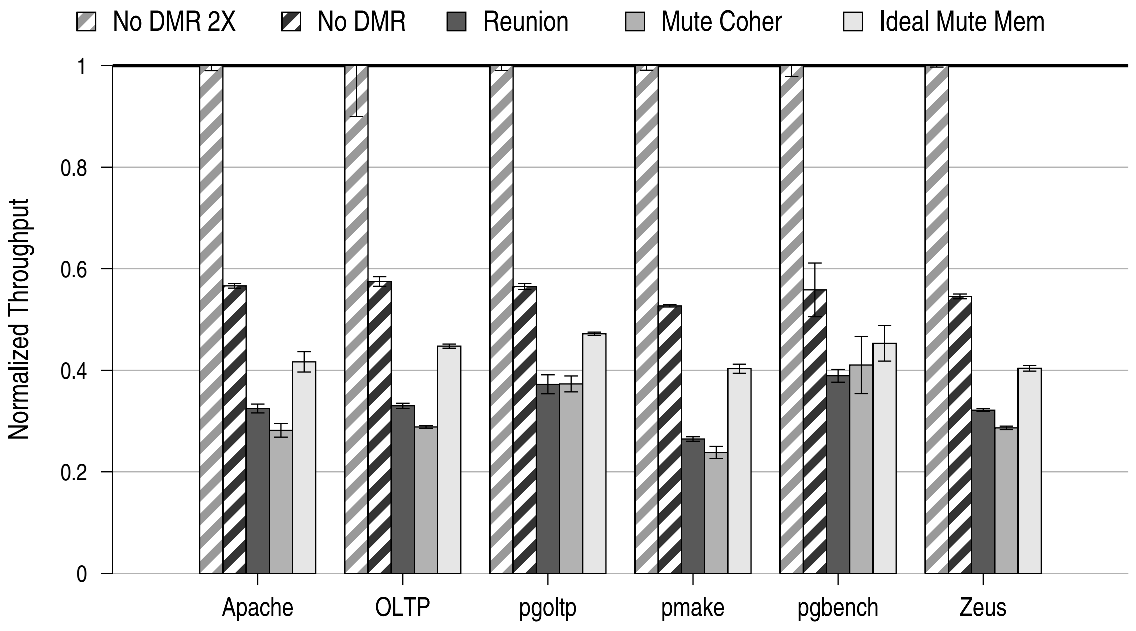


Figure 7.6 DMR Performance Comparison: Throughput

and capacity pressure on the shared cache and network resources. The *Reunion* configuration, however, sees a 22–47% decrease in the IPC of each VCPU. The performance penalty of using Reunion is 35–53% compared to the 8 VCPU *No DMR* configuration.

This result is in contrast to the published Reunion work [Smolens, 2008; Smolens et al., 2006], which reports 5–10% overheads for the 10-cycle comparison latency used here. The reasons for this discrepancy are directly related to the data presented above: increased latency from the 3-level cache hierarchy, increased serializing instructions, and increased instruction window pressure from maintaining sequential consistency. Although the benchmarks (SPEC CPU95) and target microarchitecture (SMT) are very different, Mukherjee et al. [2002] report a similar 32–40% per-threads performance loss for their *Simultaneously and Redundant Threading* (SRT) processor. Walcott et al. [2007] also report a 43% loss for one benchmark (`twolf` from SPEC CPU2000) when using DMR.

While this increase in per-thread latency is larger than some previous work, and similar to others, none of this prior work examined the impact on *throughput* created by the need to use twice as many core (or thread contexts) to run the same number of threads.

Figure 7.6 shows this overall throughput impact, and the results are dramatic. As expected, throughput lost by *No DMR*, when not running VCPUs on all cores, is nearly half that of *No DMR 2X*. The loss isn't quite half (43–45%) due to fewer cache-to-cache transfers, as shown in Figure 7.4, and due to less pressure on the capacity of the shared L3 and main memory bandwidth. The throughput for the *Reunion* configurations is approximately one third to one quarter that of *No DMR 2X*, due not only to half as many VCPUs running, but to the fact that those VCPUs each slow down by nearly a factor of two.

As discussed in Section 7.2.1.1, the original baseline Reunion configuration does not maintain coherence for the private caches of the mute cores, instead, making a best-effort approximation of coherence to deliver the correct value. But maintaining coherence can avoid the need to flush the caches on a mode switch in mixed-mode operation. The second DMR configuration (fourth bar, labeled *Mute Coher*) shows the overhead of maintaining coherence for mute caches when not switching modes. This throughput overhead ranges from 0–5% beyond that of Reunion due to the additional C2C transfers.

Though not as bad, *Ideal Mute Mem* still observes a 50–60% drop in throughput, even after idealizing all memory accesses from the mute cores. In the absence of serializing events, the mute core will nearly always run ahead of the vocal core, providing fingerprints by the time the vocal core needs them, and insulating it from the fingerprint comparison latency. This configuration does not observe memory overheads, but still observes overheads from SIs and instruction window and LSQ pressure.

Overall, the runtime overheads of Reunion execution are quite large, and arise due to increased L2 cache-to-cache transfers, increased latency for processing serializing instructions, and increase pressure on the instruction window and LSQ. Clearly, certain customers and applications are willing to pay the penalty of for the peace of mind of the high reliability it provides. However, the fact that the overheads of DMR are significantly higher than reported by Smolens et al. [2006] only reinforces the notion that DMR should only be used for applications which actually need that high level of reliability.

7.3.3 Overhead of Switching to and from DMR

Mixed-mode operation can help save some of the overhead of DMR, but incurs additional overheads of its own. In particular, this section first examines the overhead of entering and leaving dual-redundancy, by using two mixed-mode consolidated server configurations, and a redundant consolidated server baseline, and then examines the frequency that mode switching is necessary in a single-OS system.

Given the SPARC infrastructure, there is no way of evaluating switches to and from the system VMM. Thus consolidated server workloads only switch to or from DMR mode during at the end of each VMs timeslice. Two guest VMs, one which requires DMR and one which does not, are gang scheduled at a 3ms timeslice. Dynamically (or statically) partitioned consolidated workloads are not investigated in this chapter, since VCPUs would not switch modes frequently enough to be of interest.

The baseline full DMR configuration runs both guest VMs in redundant mode. The first mixed-mode configuration, the *static MMM* scheme depicted in Figure 7.1(b), allows unused redundant cores to idle when the non-redundant VM is scheduled. The second mixed-mode configuration represents a dynamic MMM system, similar to 7.2(a), which can take advantage of the idle cores to execute additional VCPUs.

Switching Overhead Base DMR does not perform mode switching, it just redundantly performs the normal multicore virtualization VCPU context switch. But this VCPU switching takes, on average, 800-1400 cycles. The overheads for switching *in* are higher than in Chapter 4 due to both the cost of synchronizing the redundant cores, and the fact that less frequent switches means more VCPU state will be found in the L3 cache or main memory. The Base DMR system synchronizes the cores at the beginning of switching one VCPU *out*, and synchronizes again and the end of switching the next one *in*. This second synchronization can be costly due to different cache latencies observed by all of the operations. The additional overheads for switching *out* arise due to the more limited bandwidth of the write-through cache.

Mixed-mode performs a basic switch *in* of a VCPU after performing the *exit* DMR mode switch transition, and performs a DMR *enter* transition after a basic switch *out*. Mute cores in the static MMM system do not perform basic switching of non-DMR VCPUs, since they go idle instead. Unlike the full DMR configuration, the cores in the mixed mode do not have to wait to synchronize after they switch in a non-DMR thread. This reduces the overheads of switching in by 20–40%.

The cost of entering DMR is somewhat larger than the cost of the baseline DMR system switching in a new thread, due to additional synchronization. The overheads for performing DMR mode exit transitions are, as expected, much higher than that of basic virtualization. In particular the cost of flushing the L2 cache lies near 8k cycles. This operation proceeds at the bandwidth of the L2 cache, which with its 15 cycle latency, 4 banks, and 4-stage pipeline, can process one line on average per cycle. It is this cost that maintaining mute coherence completely removes. The reason the cache flush is not strictly greater than 8192 cycles is that the data averages both the time the mute spends flushing and the time the vocal spends waiting for the mute. The vocal cores tend to finish their state manipulation and begin waiting few cycles after the flush has begun.

Switching Frequency The cost of the mode transitions in Table 7.4 is relatively small if these transitions occur infrequently, as in the mixed-mode consolidated servers these data are from. However, when performing mixed-mode operation on a single-OS system, transitions become necessary whenever the user applications enters the kernel, e.g., for an interrupt or system call.

To examine the impact the switching latencies have in a single-OS system, Table 7.5 presents the average number of cycles before switching from a user application to the OS, and from the OS back to the user application. These data are for the baseline, non-DMR system. All benchmarks except `Apache` and `Zeus` spend at least 200k cycles in user mode before entering the OS. All benchmarks except `Apache` make a set of transitions into and out of the OS only every 245k cycles or more (for `Apache` it is approximately 160k cycles).

The cost of switching into and out of DMR mode, from Table 7.4 is approximately 10k cycles for all benchmarks. The implication of this data is that flushing the L2 cache to allow Reunion's mute incoherence results in at least a 6.5% overhead for `Apache`, and at least a 4% overhead for the other benchmarks. Given that the overhead of maintaining mute coherence is 0–5%, from Figure 7.6, both are viable techniques for single-OS mixed-mode operation. Reunion's original incoherence proposal is better for consolidated servers that use long timeslices.

7.3.4 Latency and Throughput of a Mixed-Mode Multicore

A mixed-mode consolidated server can provide differentiated service to different VMs. Figure 7.7 demonstrates the per-thread performance gains of mixed-mode operation. The striped bars at the bottom (labeled *VM 2*) represent the per-thread performance of the guest VM which requires redundancy. The solid, top bars (labeled *VM 1*) represent the guest VM which does not require redundancy.

In a traditional consolidated server, if guest VM 2 require DMR, the all guests would need to run with DMR to protect the integrity of VM 2. The left set of bars (labeled *DMR Base* for

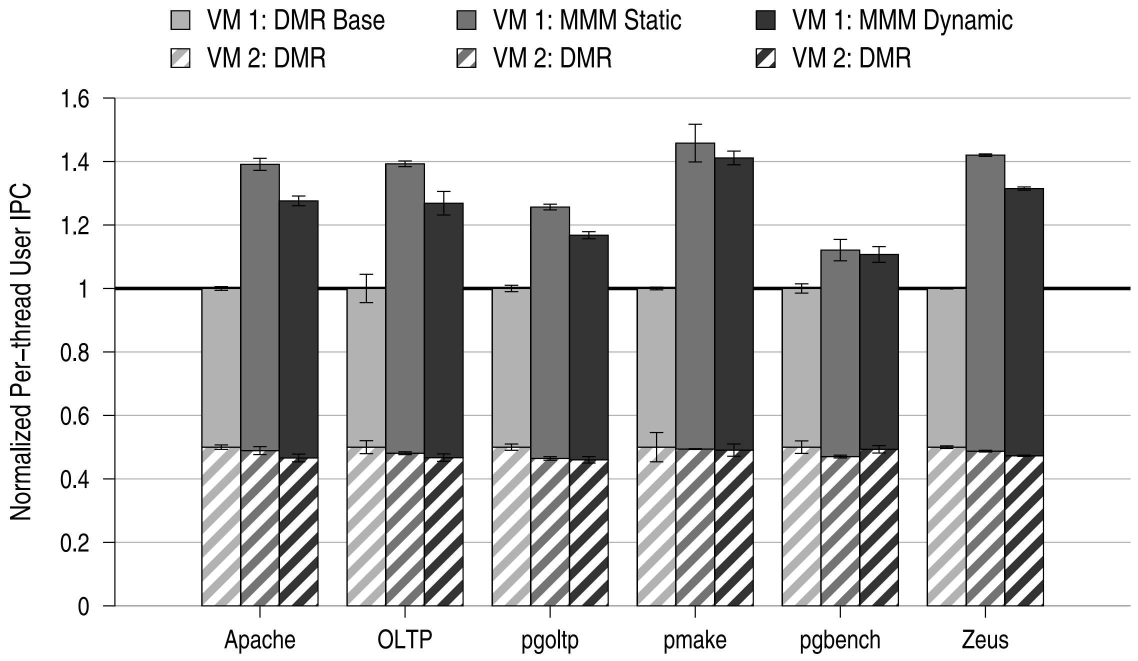


Figure 7.7 Mixed-Mode Performance Comparison: Per-thread Latency

VM 1) thus represents the Reunion DMR baseline, where DMR is used all of the time. The second set of bars, labeled *MMM Static*, represents the static MMM scheme depicted in Figure 7.1(b), where unused redundant cores are allowed to idle. Due to the latency overhead of DMR execution, VM 1 observes 36–90% speedup over the full DMR configuration. The performance of VM 2 is virtually unchanged, though `pgoltp` observes a 6.5% slowdown. A system VMM could potentially adjust the timeslice of VM 2 to make up for the difference.

The third set of bars, labeled *MMM Dynamic*, represents a dynamic MMM system, which can take advantage of the idle cores to execute additional VCPUs. In this case, the per-thread latency of those VCPUs still increases, though since more VCPUs are executing and consuming cache resources, the speedup of VM 2 is 33–80%.

Per-thread latency is only part of the picture, however, since the dynamic MMM configuration is using those otherwise-idle cores to execute more VCPUs. Figure 7.8 shows the overall system throughput, again normalized to the always-DMR baseline, and again broken into throughput

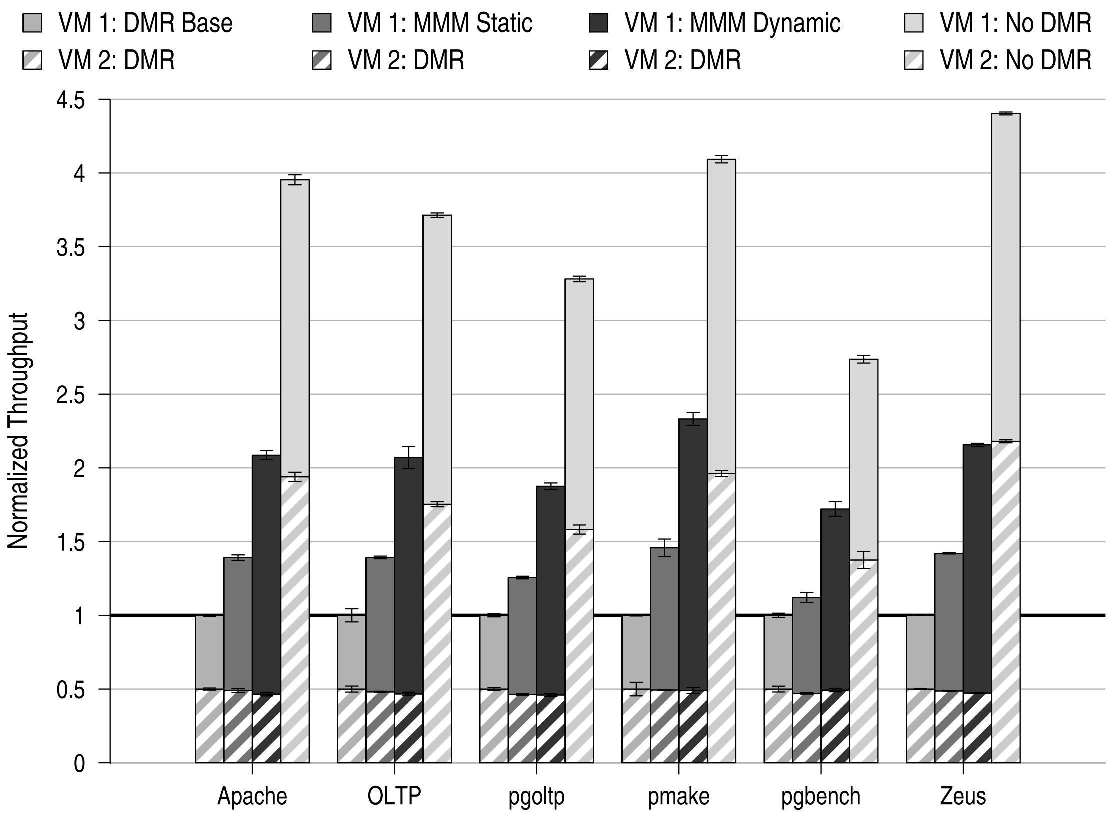


Figure 7.8 Mixed-Mode Performance Comparison: Overall Throughput

from each guest VM. The throughput of the static MMM is the same as Figure 7.7. However, for scalable applications, such as these commercial workloads, improvements in throughput can be significant using a dynamic MMM, where VM 1 now independently executes 16 VCPUs. VM 1 observes speedups of 2.5–3.7 due to the combined effect of per-VCPU latency reduction, and additional throughput from more VCPUs. Speedup of VM 1 over the static MMM configuration are 1.8–1.9. The throughput of the machine overall increases by 1.7–2.3X.

For comparison, the fourth set of bars represents the performance of a system which does not perform DMR at all. Instead of gang scheduling the guest VMs (at a 3ms timeslice), both 8-VCPU VMs are concurrently run on separate cores. This configuration achieves speedups of 2.7–4.4 for both VMs, due to the improved latency, improved throughput, and no overhead from

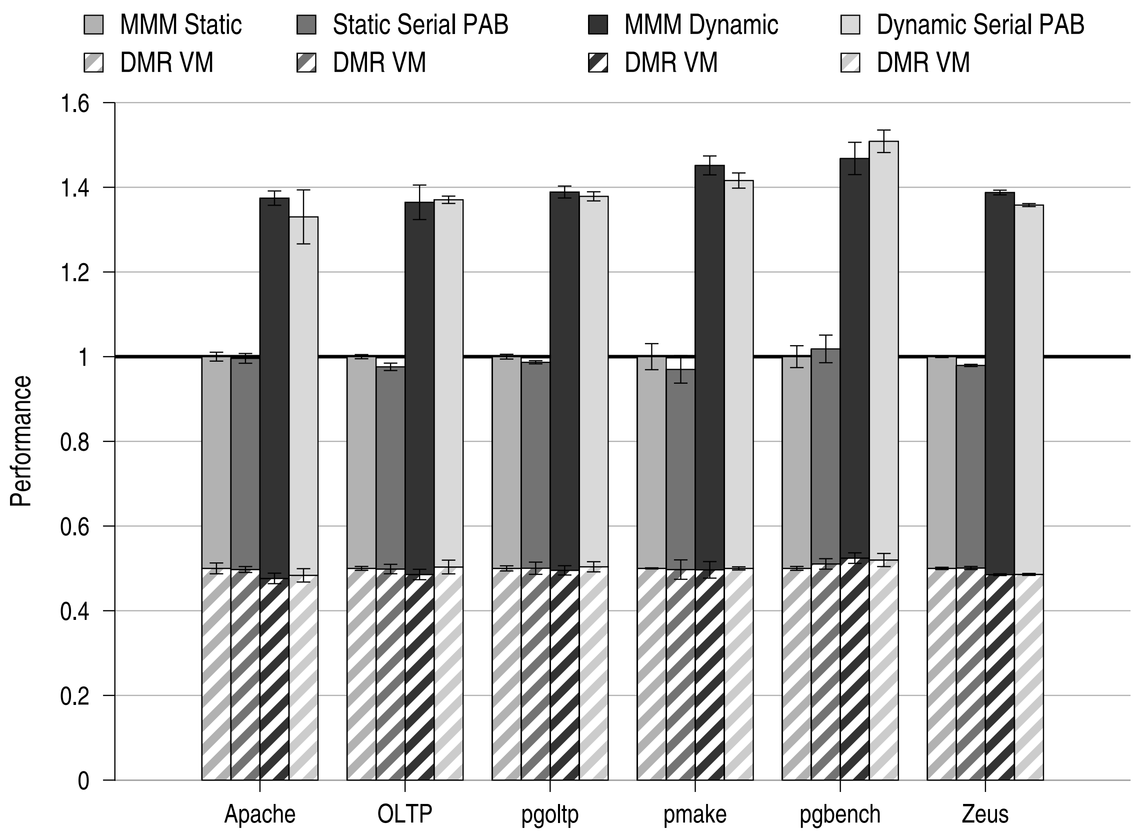


Figure 7.9 Performance Impact of PAB Latency

gang scheduling. However, this configuration does not offer any protection and peace of mind for the customers of VM 2.

7.3.4.1 Effect of PAB Latency

Figure 7.9 shows the impact of a 2-cycle PAB lookup in serial before accessing the L2 cache. The figure shows impact of serial PAB lookups for both static and dynamic MMM configurations. Performance is normalized to the static mixed-mode configuration show in Figure 7.8.

Since only store write-throughs are stalled by this serial lookup, the performance impact arises primarily through increase pressure on the instruction window and other structures. Indeed, serial lookups decrease performance of the non-DMR VM by 0–3%. The DMR VM does not incur the penalty of the PAB. Its performance does not change.

Note that because performance is calculated independently for each VM, and Figure 7.9, uses *Static MMM* as the baseline, both VMs of *Static MMM* are shown to have normalized performance of 0.5. In Figure 7.7, the performance of the VMs in *Static MMM* is unequal because performance is normalized to a different baseline.

7.4 Related Work

Many circuit- and microarchitectural-level techniques for tolerating various hardware faults have been discussed in Sections 2.2 and 5.6. Of primary interest to this chapter is a number of recent microarchitectural DMR proposals, which join together two cores to reliably execute one VCPU [Gomaa et al., 2003; LaFrieda et al., 2007; Mukherjee et al., 2002; Reinhardt and Mukherjee, 2000; Rotenberg, 1999; Smolens, 2008; Smolens et al., 2006; Weaver and Austin, 2001; Zhou, 2006]. Two of these proposals suggest (without investigate those claims further) that DMR can easily be turned on and off in these systems [Mukherjee et al., 2002; Walcott et al., 2007]. This chapter demonstrates, however, that running some applications of one system in DMR mode and some in non-DMR mode is not as straightforward as it might first appear.

That said, there is nothing inherent in any of these DMR proposals that is incompatible with the modifications for mixed-mode execution proposed in this chapter. Yet each of these systems (including Reunion, which was chosen as the basis for mixed-mode) has drawbacks with respect to the goals of this dissertation.

In particular, Reinhardt and Mukherjee [2000]; Rotenberg [1999] focus on SMT processors, not multicores. Gomaa et al. [2003]; Mukherjee et al. [2002]; Weaver and Austin [2001]; Zhou [2006] statically couple redundant cores with a queue to communicate identical load values from the master thread to the slave thread. Weaver and Austin [2001] also communicate the input register values of each instruction. A queue for load values prevents one of the issues that arises for Reunion, that of input incoherence. It also eliminates the need for both cores to access the cache, reducing duplication. However, no duplication means that all private caches, LSQ bypassing logic, and the load value queue itself must be hardened against faults. Static coupling also reduces the ability of an overcommitted MMM from flexibly scheduling VCPUs to cores.

The proposal of LaFrieda et al. [2007] allows dynamic coupling of redundant cores, but does not work with a directory coherence protocol, and requires significant coherence protocol changes to work with multithreaded applications.

In addition, none of the above proposals (except Reunion) have been previously evaluated with multithreaded commercial applications.

Walcott et al. [2007] observe that the continuous use of redundant multithreading (RMT), within a single SMT core, can lead to significant IPC overheads. They report overheads of 43% for one benchmark. To combat this overhead, they propose to toggle RMT one and off for a given application to achieve the desired level of vulnerability from faults. They address how to decide when RMT is and is not necessary, given the *Architectural Vulnerability Factor* (AVF) of the processor and application, but do not address the other issues relating to mixed-mode execution.

Aggarwal [2008] also addresses the need to offer differentiated service to certain applications. This proposal starts with the assumption of replicated execution through a software VMM rather than through microarchitectural mechanisms. To reduce software overheads, the VMM compares the outputs of redundant copies only before I/O, and thus cannot easily support multithreaded applications which can experience non-determinism within each redundant execution. The microarchitectural support used in this chapter performs replication and comparison on each thread before it communicates with any other threads, preventing the non-determinism problem.

Overshadow is a software VMM-based memory encryption technique that can protect application data from a security-compromised OS Chen et al. [2008]. Similar techniques could be used to provide additional levels of protection among different applications, or different guest virtual machines. *Overshadow* may be able to detect certain cases when an application's data is modified due to hardware faults that may occur when another application is executing through data integrity checks. However, it cannot prevent the corruption from occurring in the first place.

Configurable Isolation Aggarwal et al. [2007] is a technique to reconfigure around permanent hardware faults while losing the use of only a small fraction of the available core, cache, and network resources. In addition, they partition physical memory between different *color domains*, and use redundant hardware to maintain isolation between partitions.

7.5 Chapter Summary

As the underlying hardware becomes less reliable, system designers will seek to include higher-level redundancy techniques such as Dual-Modular Redundancy (DMR) in their multicore designs [Gomaa et al., 2003; LaFrieda et al., 2007; Mukherjee et al., 2002; Smolens et al., 2006; Zhou, 2006]. DMR can provide excellent coverage from a variety of transient, intermittent, and permanent sources, yet it comes with latency overheads of 35–50% for each thread, and throughput overheads of 3–4 times.

The work in this chapter builds on the observation that some applications even today require DMR, and more will in the future. Yet not all software requires DMR all the time. To address this diversity in needs, even among application running on the same machine, this chapter proposes and designs a *Mixed-Mode Multicore* (MMM). An MMM enables applications that need extra reliability to run in DMR mode, while applications that do not need DMR observe significant improvements in performance.

Though conceptually simple, two key challenges arise in designing an MMM. First, care must be taken both during execution, and during a mode switch in order to protect the rest of the system from faults occurring to non-DMR applications. As part of the solution to this problem, this chapter proposes a hardware structure called the *Protection Assistance Buffer* (PAB) to re-validate the permissions of stores before they write into the L2 cache. The PAB has no impact on performance if it can be accessed in parallel with the L2 cache, and imposes a 1–5% performance loss if accessed serially. But once continued system integrity is provided, an MMM which statically maps VCPUs to a pair of cores can idle one of the cores when scheduling a non-DMR application. For a system where one application requires DMR and one doesn't, this static MMM can improve overall system performance by 25–45% compared to a system that executes both application with DMR.

The second key challenge is that providing a simple interface to software complicates the scheduling of virtual processors (VCPUs) to cores: As software dictates which VCPUs need to execute redundantly, the number of cores required to execute all of the VCPUs changes, creating rapidly changing dynamic heterogeneity with respect to the software's requirements. To address this second problem, this chapter proposes to use the multicore virtualization techniques from Chapter 4 to enable an overcommitted system and flexibility assign VCPUs to cores. In doing so, the cores left idle by a non-DMR application can be used to improve throughput. This dynamically scheduled MMM can improve the overall performance of a two-application system by 1.9-2.1 times.

If reliability trends continue for the next decade or longer, multicore processors without DMR will become less and less reliable, and therefore useful for a smaller fraction of applications. Eventually, manufacturing experts may choose to push technology to a point where essentially *all* software needs to run with DMR. In the meantime, however, Mixed-Mode Multicore processors can help ease this transition by letting the user run more applications in DMR mode with every processor generation, instead of having to switching all at once from running with no applications in DMR to incurring a 3–4X throughput loss for all applications.

1.	Drain pipeline
2.	Store all state to caches

Basic Switch Out

Leave Perf. Mode (New VCPU) or

Stay in Current Mode (New VCPU)

1.	Load previously saved state

Basic Switch In

Enter Perf. Mode (New VCPU) or

Stay in Current Mode (New VCPU)

	Vocal	**Mute**
1.	Wait for outstanding sync requests	
2.	Flush pipelines & sync	
3.	Store ***all*** state & sync	
4.	Wait	Flush & write-back L2 cache
5.	Sync & continue	

Leave Reliable Mode (New VCPU)

	Vocal & Mute
1.	Load previously saved state
2.	Sync & resume execution

Enter Reliable Mode (New VCPU)

	Vocal	**Mute**
1.	Wait for outstanding sync requests	
2.	Flush pipelines & sync	
3.	Store ***privileged*** state & sync	
4.	Wait	Flush & write-back L2 cache
5.	Sync & continue	

Leave Reliable Mode (Same VCPU)

	Vocal	**Mute**
1.	Store all state	Load previous priv state
2.	Synchronize	
3.	Wait	Ld & compare vocal's priv state
4.	Wait	Load vocal's user state
5.	Sync & resume execution	

Enter Reliable Mode (Same VCPU)

Table 7.1 State Transitions for Entering and Leaving Reliable Mode

	SIs per 1k	Fetch Stalled by SI	
Workload	No DMR	No DMR	Reunion
Apache	2	33%	46%
OLTP	1.1	20%	30%
pgoltp	0.7	13%	20%
pmake	0.6	10%	16%
pgbench	0.5	15%	17%
Zeus	2.4	42%	49%

Table 7.2 Impact of Serializing Instructions (SIs). Table shows SIs per 1k instructions, percent of cycles where fetch is stalled due to an SI in the window, and an estimate for the additional SI performance impact to Reunion over the baseline.

Workload	No DMR	Reunion	Mute Coher.	Ideal Mute Mem
Apache	17%	24%	22%	26%
OLTP	15%	27%	30%	27%
pgoltp	20%	35%	32%	35%
pmake	43%	57%	59%	53%
pgbench	24%	38%	33%	40%
Zeus	21%	25%	24%	26%

Table 7.3 Fraction of Cycles Instruction Window or LSQ are Full

		Basic Virt		DMR	DMR Exit	
	Workload	**In**	**Out**	**Enter**	**State & Sync**	**Cache Flush**
Base DMR	Apache	918	1337	n/a		
	OLTP	898	1152	n/a		
	pgoltp	830	1167	n/a		
	pmake	798	882	n/a		
	pgbench	868	1148	n/a		
	Zeus	851	1295	n/a		
Static MMM	Apache	686	1055	1892	1221	8202
	OLTP	666	1065	1861	1026	8195
	pgoltp	656	1062	1856	1054	8194
	pmake	622	991	1852	744	8172
	pgbench	663	1051	1887	1046	8195
	Zeus	676	1058	1945	1029	8177
Dynamic MMM	Apache	744	1141	1650	1080	8185
	OLTP	736	1124	1648	934	8198
	pgoltp	689	1118	1588	856	8182
	pmake	637	1055	1611	716	8164
	pgbench	701	1097	1628	900	8176
	Zeus	717	1128	1682	1006	8190

Table 7.4 Mixed-Mode Switching Overheads

Workload	User Cycles	OS Cycles
Apache	59k	98k
OLTP	218k	52k
pgoltp	210k	35k
pmake	312k	47k
pgbench	554k	126k
Zeus	65k	220k

Table 7.5 Average Cycles Before Switching Modes for Single-OS

Chapter 8

Conclusions

The continued exponential growth in the number of transistors on a chip presents several challenges to computer architects, who's job it is to determine how to use these transistors effectively. In order to simplify the problem for hardware designers, most computer manufacturers have switched to building multicore processors, where one processing *core* is designed and then replicated several times across the chip. Multicore chips, however, greatly complicate the task of software running on the chip.

One complicating issue, identified by others, is the desire to design different cores with different engineering trade-offs, resulting in *static heterogeneity* (or *asymmetry*) [Gschwind et al., 2005; Kumar et al., 2003, 2004; Li et al., 2007]. However, this dissertation identified another issue increasing the complexity of multicore processors: that of *dynamic heterogeneity*. Dynamic heterogeneity can occur, even among physically homogeneous cores, from reliability, power, or thermal conditions, or even different cache and TLB contents, for example. This heterogeneity can change very rapidly, creating *uncertainty* regarding which cores are available or most appropriate for running a given computation at a particular time. Current multicore processors, and most designs for future multicore systems, simply pass this uncertainty up to the software. Yet software's need for extracting concurrency in the first place is a big enough challenge in the multicore era. Continuing to require software to explicitly manage the use of all cores in order to express that concurrency to the hardware is an additional burden which is both undesirable and unattainable.

This dissertation began a push toward hardware taking a more active role in the management of its own resources. It did so by proposing hardware techniques to *virtualize* the cores of a multicore processor. This multicore virtualization allows hardware to transparently remap the *virtual processors* (VCPUs) exposed even to a single operating system (OS) to any subset of physical cores. This dissertation demonstrated that by using these techniques, a processor can manage its dynamically changing reliability conditions, the varying scheduling requirements of a consolidated server, and the rapidly changing reliability requirements of the software. Moreover, by using the proposed virtualization techniques, only the last of these three objectives requires any changes to the system software.

8.1 Contributions and Key Results

This dissertation made several contributions for both researchers and designers of future multicore processors:

- Dynamic heterogeneity was identified as an emerging challenge, as well as an opportunity, for future multicore processors. Several examples of dynamic heterogeneity were described at a high level in Chapter 1, while Chapters 5-7 each presented and analyzed an example of dynamic heterogeneity in detail.

- Multicore virtualization techniques were proposed to allow hardware to manage the use of its resources without requiring any modifications to existing application or system software (Chapter 4). An *overcommitted system* was identified as one where the number of cores available to perform work is less than the number of VCPUs exposed the operating system (OS). The challenges of executing in such an environment were examined, and solved in part with a proposal for hardware spin detection. The proposed *Spin Detection Buffer* (SDB) was show to be capable of detecting all known spins in all workloads examined in this dissertation, yet operate with a runtime overhead of less than 1.5%.

- *Intermittent faults*, a class of hardware faults which occur in bursts from several cycles to several seconds or more, were identified as a source of dynamic heterogeneity and an emerging challenge for future multicores (Chapter 5). An analysis of three existing techniques to adapt to the effects of these faults found the techniques deficient in several areas. To remedy these drawbacks, a fourth technique using multicore virtualization was proposed. The proposed technique was the only one to achieve high marks on all of the performance metrics, gracefully handle multiple concurrent failures, and involve only moderate complexity. By using this proposed technique, the chip can present a view of continuous, fully-functional, reliable operation to the system software, insulating software from the details of rapidly changing hardware reliability conditions.

- Scheduling consolidated server workloads on a multicore processor was investigated in Chapter 6. This chapter found that the current status-quo of *gang scheduling*, or *co-scheduling*, all VCPUs from a single guest virtual machine (VM) leads to conflicting objectives with respect to the system's goals. The proposed multicore virtualization techniques, on the other hand, allow gang scheduling to be avoided. Instead, cores are dynamically partitioned among guest VMs, specializing the predictive structures of each core. This dynamic heterogeneity, intentionally created from physically homogeneous cores, allows the guest VM within each partition to operate with higher throughput, lower transaction latency, and more isolation compared to gang scheduling. Furthermore, these partitions can be easily and quickly adjusted to handle not only bursts in demand, but also any changing capabilities of the underlying hardware.

- The need for simultaneously executing both applications which require high reliability and those that require high performance was addressed in Chapter 7. The performance costs of Dual Modular Redundancy (DMR), one technique for maintaining very high reliability, were examined, and shown to be a 35-48% reduction in per-thread IPC, and a 2.5–4X redunction in overall system throughput. While a significant, and growing, fraction of applications are willing to incur such overheads for high reliability and the peace of mind that comes with it, that price is too high for many other applications. A *Mixed-Mode Multicore* (MMM) design was proposed to allow both classes of applications to simultaneously run on the same machine. Though conceptually simple, two issues were discovered when designing an MMM. First, to protect the rest of the system from faults occurring to non-DMR applications, an MMM must prevent non-reliable applications from improperly writing memory, as well as properly handle transitions between DMR and non-DMR modes. Second, to allow high-performance applications to fully utilize all available cores, techniques such as the multicore virtualization proposed in this dissertation must be used. The resulting MMM was shown to improve overall system throughput by approximately 2X, compared to a traditional DMR system, when one high-reliability and one high-performance application are concurrently executing.

8.2 Additional Uses for Multicore Virtualization

The multicore virtualization techniques proposed in this work have been used in two other projects performed in collaboration with the work in this dissertation. In addition, these multicore virtualization techniques appear directly applicable to several other recent research proposals.

8.2.1 Computation Spreading and Over-provisioned Multicore Systems

Future processors will have a growing transistor budget, but are already approaching the limits of cost-effective cooling. To tolerate this divergence of trends, an *Over-provisioned Multicore System* (OPMS) was proposed, where the number of physical cores exceeds the number that can be actively computing due to power and thermal limitations [Chakraborty, 2008; Chakraborty et al., 2007]. Though such a system may appear counter-intuitive, the observation is made that inactive cores can remain useful by retaining predictive state in close proximity to the pipeline, by allowing thermal hot-spots to dissipate, or simply existing as spares. The challenge of an OPMS is to efficiently use the growing number of cores, despite not being able to simultaneously activate them.

One proposed solution is to perform *Computation Spreading* (CSP) across the full set of cores [Chakraborty et al., 2006]. CSP identifies similar *computation fragments* from different threads, and executes them on a single core, while identifying dissimilar fragments from within each thread, and spreads their execution across multiple cores. CSP actively creates dynamic heterogeneity by tailoring each core's predictive structures to a certain type of computation.

The mechanism enabling CSP to be performed with unmodified software is the multicore virtualization proposed in this dissertation. CSP first requires the basic multicore virtualization techniques in order to move VCPUs around the chip in an effort to improve their locality. But CSP, even on an OPMS, also requires support for *overcommitting*. The reason for this disparate nomenclature is the following: An OPMS may have more cores than it exposes VCPUs to the OS. But power and thermal limitations restrict the number of cores that are available at any given time, and CSP further restricts which cores are *currently appropriate* for a given set of VCPUs. Whenever the number of appropriate cores is less than the number of VCPUs wishing to execute on those cores, that subset of cores becomes overcommitted, regardless of the remaining number of cores in the system.

8.2.2 Support for Other Research

In addition to CSP and OPMS, multicore virtualization, with support for overcommitting, appears directly applicable to other research as well. Speculative multithreading and other techniques (e.g., [Ípek et al., 2007; Sohi et al., 1995; Zilles and Sohi, 2002]) dynamically couple multiple cores together to run a single VCPU more quickly. These techniques can allow the same chip to be effective for single or multithreaded applications, and can help mitigate the effects of Amdahl's Law for semi-parallel applications [Hill and Marty, 2008].

By increasing the number of cores used to execute a single VCPU, the number of VCPUs that can execute concurrently becomes limited, i.e., an overcommitted system is created. This scenario is exactly like the one examined in Chapter 7 when using DMR on a Mixed-Mode Multicore.

What makes using multicore virtualization especially appealing for semi-parallel applications, however, is that the proposed Spin Detection Buffer (SDB) can *automatically* infer parallel and serial regions of the code when VCPUs are either spinning on a lock or running the OS idle loop. When a VCPU stops performing useful work, the SDB notifies the Virtualization Controller, which could then join idle cores together to more quickly execute the remaining VCPUs which *are* performing useful work. If the latency of joining and separating cores is low enough, this action could be initiated even, for example, when the threads on several VCPUs have already reached a barrier, but are waiting on one or more slower threads.

An overcommitted system can also potentially arise with the Internet Suspend/Resume (ISR) project [Satyanarayanan et al., 2007]. The idea with ISR is to separate the *state* of a personal computer from the hardware, allowing a user to physically move from one computer to another, and start up a system VM containing the state of their session exactly as it was on the previous computer, including an already booted OS and running applications. This feat is accomplished by virtualizing the hardware, and using the network for storage. Of interest is that the number of cores on the computers can be different, yet for most OSs, there is no way to change the number of cores it is using without a reboot. If moving from a machine with fewer core to one with more, the additional cores will be unused. If moving from a machine with more to one with fewer, the VM will not load, and must force the OS to reboot.

Allowing an overcommitted system solves this problem with ISR. The OS can be configured with the maximum number of cores that the user expects to see, yet can run with little overhead on a machine with any number of fewer cores.

8.3 Cost and Benefits of Hardware Virtualization

The multicore virtualization proposed in this dissertation is *hardware-centric*. It holds as one of its major design goals the ability to virtualize completely unmodified operating systems. Yet with the proliferation of para-virtualized system VMs, such as Xen [Barham et al., 2003], or IBM and Sun's hypervisors [Armstrong et al., 2005; Charlesworth, 1998], operating system developers are becoming accustomed to the idea of the OS being modified to better support virtualization. One useful modification to the traditional interface (already provided by IBM [Armstrong et al., 2005]) would be to notify the virtualization layer when a VCPU is spinning, idle, or otherwise not performing useful work. If such an enhancement is possible for a particular scenario, it is likely to be simpler, and potentially more accurate, than using the heuristic-based Spin Detection Buffer (SDB).

The other components of multicore virtualization, however, appear more useful in a broader range of scenarios — useful enough, in fact, that AMD and Intel have both adopted mechanisms similar to those proposed for manipulating VCPU state [Advanced Micro Devices, 2005; Uhlig et al., 2005]. These companies have no public plans, however, to place control over VCPU state manipulation in the hands of a hardware-based Virtualization Controller. And it is this controller that allows the virtualization to become useful while remaining transparent to the system software.

The controller is also the only piece that needs to change to support both adapting to intermittent faults, and dynamic partitioning of consolidated servers. Should a future multicore already contain the basic virtualization mechanisms, either as proposed or with some software support, adding control logic to enable these additional applications seems like a winning proposition. On the other hand, the benefits of implementing the entire multicore virtualization proposal for either intermittent faults or consolidated servers individually may not outweigh the cost, given other lower performance, but simpler alternatives.

Adding mixed-mode operation to a system that already implements microarchitectural DMR is straightforward if the primary goal is to improve single-thread latency, and possibly save power. Yet simpler alternatives may arise, which will not require implementing the full multicore virtualization techniques for the sole purpose of improving throughput for high-performance applications. If hardware reliability concerns actually do rise to the level where a significant number of users want DMR for their desktop applications, then providing a simplified mixed-mode operation would likely prove beneficial for a company.

8.4 Revisiting the Thesis Statement

Chapter 1 presents the following thesis statement:

> *Future multicore processors should support software-transparent virtualization of on-chip cores due to the emerging challenges and opportunities of dynamic heterogeneity.*

A basic assumption of this thesis is that dynamic heterogeneity will exist in future multicore processors, and present opportunities as well as challenges. This assumption is supported by

Chapters 5–7, as well as Chakraborty et al. [2006] and Chakraborty et al. [2007], which each present an example of an opportunities or challenge (or both) of dynamic heterogeneity.

Given this assumption, the above thesis makes two claims: not only is virtualization of on-chip cores needed, but this virtualization needs to be software transparent.

The need for software transparency arises for practical, but also somewhat ideological, reasons. Many of the issues have been discussed at various places in this dissertation (e.g., Sections 1.1.1, 5.3.4, and 7.2.5), and come back to the same problem: the desire to maintain a clean hardware/software interface that does not change significantly, or create new obligations for the system software, with every new generation of hardware. Without alternatives, such as those proposed in this dissertation, the task of acquiring and then using detailed information about the current configuration and capabilities of each core will place a significant burden on system software, at the very least, threatening to break backward compatibility.

The need for hardware virtualization is more quantitative, and is supported by data in this dissertation. In particular, Chapters 5 and 7 each demonstrated an example of rapidly changing dynamic heterogeneity. Given that the capabilities and configurations of cores may change within thousands or millions of cycles, the need for hardware support is clear: current system software is not up to the task (see Section 5.4), and improved system software is still hampered by the cost of trapping into the operating system to perform and implement a decision about how to efficiently use each core [Nellans et al., 2005]. The result is that the latency of adapting to dynamic heterogeneity can easily surpass any opportunity to do so.

8.5 Future Directions

The thesis of this dissertation can thus be justified given its engineering and ideological context. That being said, compromises must often be made to a proposal in order to realize its practical benefits. The challenge in this case is to strike the proper balance between hardware cost and complexity, software burden, and perceived improvement.

For example, this dissertation has shown that certain kinds of hardware resource management can be undertaken with unmodified system and application software, yet this approach cannot be expected to scale to hundreds or more cores. The reason is because modern OSs treat multicore chips the same as they did uniprocessors: by time sharing each VCPU with multiple application threads, OS threads, resource management tasks, and other functions. Consequently, two major problems arise. First, system software hides concurrency from the hardware, such as independence between the system and user code that typically executes on a single core, or the existence of other runnable user threads that are not currently scheduled. Second, it hinders opportunities for efficiency by scheduling multiple, unrelated, tasks onto the same VCPU.

As shown in this dissertations, some resource management tasks are best suited to hardware, such as fine-grain mapping of computation to cores. Yet some tasks remain best suited for system software, such as setting policies and helping to expose all available concurrency. In defining the interface between hardware and software for future multicores, researchers should work to define appropriate levels of abstraction, creating an interface that can exchange information in both directions for efficient resource management, and can remain stable across several generations of diverse chips.

What the proper balance should be for a particular multicore systems of the future is dependent on a number of factors, and will be the subject of much future research. Two claims, however, can be made with near certainty: 1) future multicore processors will be much more complex than they are today, due in part to dynamic heterogeneity, and 2) both hardware *and* software will need to take on more responsibility for managing this complexity.

Bibliography

Advanced Micro Devices. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, Dec 2005.

Aggarwal, Nidhi. PhD thesis, University of Wisconsin-Madison, 2008.

Aggarwal, Nidhi, Parthasarathy Ranganathan, Norman P. Jouppi, and James E. Smith. Configurable isolation: building high availability systems with commodity multi-core processors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA)*, pages 470–481, 2007.

Ailamaki, Anastassia, David J. DeWitt, Mark D. Hill, and David A. Wood. DBMSs on a modern processor: Where does time go? In *Proceedings of the 25th Annual Very Large Databases (VLDB)*, pages 266–277, 1999.

Alameldeen, Alaa R. and David A. Wood. Variability in architectural simulations of multi-threaded workloads. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA)*, 2003.

Anderson, Thomas E., Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, 1992.

Armstrong, W, R Arndt, D Boutcher, R Kovacs, D Larson, K Lucke, N Nayar, and R Swanberg. Advanced virtualization capabilities of POWER5 systems. *IBM Journal and Research and Development*, 49(4/5), 2005.

Arpaci-Dusseau, Remzi H. and Andrea C. Arpaci-Dusseau. Fail-stutter fault tolerance. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS)*, 2001.

Austin, Todd, David Blaauw, Trevor Mudge, and Krisztián Flautner. Making typical silicon matter with razor. *IEEE Computer*, 37(3):57–65, 2004.

Austin, Todd M. DIVA: A reliable substrate for deep submicron microarchitecture design. In *Proceedings of the 32th Annual International Symposium on Microarchitecture (MICRO)*, 1999.

Balakrishnan, Saisanthosh, Ravi Rajwar, Mike Upton, and Konrad Lai. The impact of performance asymmetry in emerging multicore architectures. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA)*, 2005.

Barford, Paul and Mark Crovella. Generating representative web workloads for network and server performance evaluation. In *Proceedings of the 1998 International Conference on Measurement and Modeling of Computer Systems*, 1998.

Barham, Paul, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the 19th Annual Symposium on Operating Systems Principles (SOSP)*, 2003.

Bernick, David, Bill Bruckert, Paul Del Vigna, David Garcia, Robert Jardine, Jim Klecka, and Jim Smullen. Nonstop advanced architecture. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN)*, 2005.

Blough, Douglas M., Gregory F. Sullivan, and Gerald M. Masson. Intermittent fault diagnosis in multiprocessor systems. *IEEE Transactions on Computers*, 41(11):1430–1441, 1992.

Blough, Douglas M., Fadi J. Kurdahi, and Seong Yong Ohm. High-level synthesis of recoverable VLSI microarchitectures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 7(4):401–410, 1999.

Borden, Terry L., John P. Hennessy, and James W. Rymarczyk. Multiple operating systems on one processor complex. *IBM Systems Journal*, 28(1):104–123, 1989.

Borkar, Shekhar. Microarchitecture and design challenges for gigascale integration: Keynote. In *Proceedings of the 37th Annual International Symposium on Microarchitecture (MICRO)*, 2004.

Borkar, Shekhar, Tanay Karnik, Jim Tschanz, Ali Keshavarzi, and Vivek De. Parameter variations and impact on circuits and microarchitecture. In *Proceedings of the 40th Annual Conference on Design Automation*, 2003.

Bossen, Douglas. CMOS soft errors and server design. In *Workshop on Radiation Induced Soft Errors, Proceedings of the IEEE International Reliability Physics Symposium*, 2002. As quoted by Mukherjee et al. [2005].

Bower, Fred A., Daniel J. Sorin, and Sule Ozev. A mechanism for online diagnosis of hard faults in microprocessors. In *Proceedings of the 38th Annual International Symposium on Microarchitecture (MICRO)*, 2005.

Bowman, K.A., S.G. Duvall, and J.D. Meindl. Impact of die-to-die and within-die parameter fluctuations on the maximum clock frequency distribution for gigascale integration. *IEEE Journal of Solid-State Circuits*, 37(2):183–190, Feb 2002.

Brooks, David and Margaret Martonosi. Dynamic thermal management for high-performance microprocessors. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture (HPCA)*, 2001.

Bugnion, Edouard, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems*, 15(4):412–447, 1997.

Chakraborty, Koushik. *Over-provisioned Multicore Systems*. PhD thesis, University of Wisconsin-Madison, Aug 2008.

Chakraborty, Koushik, Philip M. Wells, and Gurindar S. Sohi. Computation spreading: Employing hardware migration to specialize CMP cores on-the-fly. In *Proceedings of the 12th International conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.

Chakraborty, Koushik, Philip M. Wells, and Gurindar S. Sohi. A case for an over-provisioned multicore system: Energy efficient processing of multithreaded programs. Technical Report CS-TR-2007-1607, University of Wisconsin-Madison, Aug 2007.

Chang, Jichuan. *Cooperative Caching for Chip Multiprocessors*. PhD thesis, University of Wisconsin-Madison, Aug 2007.

Chang, Jichuan and Gurindar S. Sohi. Cooperative cache partitioning for chip multiprocessors. In *Proceedings of the 21st Annual International Conference on Supercomputing (ICS)*, 2007.

Charlesworth, Alan. Starfire: Extending the smp envelope. *IEEE Micro*, 18(1):39–49, 1998.

Chen, Xiaoxin, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of the 13th International conference on Architectural Support for Programming Languages and Operating*

Systems (ASPLOS), pages 2–13, 2008.

Compaq Computer Corp. *Alpha 21264/EV6 Microprocessor Hardware Reference Manual*. Compaq Computer Corp., 2000.

Constantinescu, Cristian. Intermittent faults in VLSI circuits. In *Proceedings of the IEEE Workshop on Silicon Errors in Logic - System Effects (SELSE)*, 2007.

Constantinescu, Cristian. Trends and challenges in VLSI circuit reliability. *IEEE Micro*, 23(4): 14–19, 2003.

Contant, Olivier, Stéphane Lafortune, and Demosthenis Teneketzis. Diagnosis of intermittent faults. *Discrete Event Dynamic Systems*, 14(2):171–202, 2004.

Conway, Pat and Bill Hughes. The AMD Opteron Northbridge architecture. *IEEE Micro*, 27(2): 10–21, 2007.

Deen, G, M Hammer, J Bethencourt, I Eiron, J Thomas, and J Kaufman. Running Quake II on a grid. *IBM Journal and Research and Development*, 45(1), 2006.

Dorsey, J., S. Searles, M. Ciraula, S. Johnson, N. Bujanos, D. Wu, M. Braganza, S. Meyers, E. Fang, and R. Kumar. An integrated quad-core Opteron processor. In *Proceedings of the 2007 International Solid-State Circuits Conference*, pages 102–103, Feb. 2007.

Driesen, Karel and Urs Hölzle. The cascaded predictor: economical and adaptive branch target prediction. In *Proceedings of the 31st Annual International Symposium on Microarchitecture (MICRO)*, pages 249–258, 1998.

Eden, A. N. and T. Mudge. The YAGS branch prediction scheme. In *Proceedings of the 31st Annual International Symposium on Microarchitecture (MICRO)*, pages 69–77, 1998.

Eisele, M., J. Berthold, D. Schmitt-Landsiedel, and R. Mahnkopf. The impact of intra-die device parameter variations on path delays and on the design for yield of low voltage digital circuits. In *Proceedings of the 1996 International Symposium on Low Power Electronics and Design*, pages 237–242, 1996.

Ernst, Dan, Nam Sung Kim, Shidhartha Das, Sanjay Pant, Rajeev Rao, Toan Pham, Conrad Ziesler, David Blaauw, Todd Austin, Krisztian Flautner, and Trevor Mudge. Razor: A low-power pipeline based on circuit-level timing speculation. In *Proceedings of the 36th Annual International Symposium on Microarchitecture (MICRO)*, 2003.

Ernst, Dan, Shidhartha Das, Seokwoo Lee, David Blaauw, Todd Austin, Trevor Mudge, Nam Sung Kim, and Krisztian Flautner. Razor: Circuit-level correction of timing errors for low-power operation. *IEEE Micro*, 24(6):10–20, 2004.

Figueiredo, Reneto, Peter A. Dinda, and Jose Fortes. Resource virtualization renaissance. *IEEE Computer*, 38(5):28–31, 2005.

Forrester Research, Inc. The total economic impact of Sun Microsystems' enterprise consolidation solutions, Jul 2004. `http://www.sun.com/datacenter/consolidation/docs/tei_h.pdf`. Viewed 7/23/2008.

Gomaa, Mohamed, Chad Scarbrough, T. N. Vijaykumar, and Irith Pomeranz. Transient-fault recovery for chip multiprocessors. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)*, 2003.

Govil, Kinshuk, Dan Teodosiu, Yongqiang Huang, and Mendel Rosenblum. Cellular Disco: Resource management using virtual clusters on shared-memory multiprocessors. In *Proceedings of the 16th Annual Symposium on Operating Systems Principles (SOSP)*, 1999.

Govil, Kinshuk, Dan Teodosiu, Yongqiang Huang, and Mendel Rosenblum. Cellular Disco: Resource management using virtual clusters on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 18(3):229–262, Aug 2000.

Gschwind, Michael, Peter Hofstee, Brian Flachs, Marty Hopkins, Yukio Watanabe, and Takeshi Yamazaki. A novel SIMD architecture for the Cell heterogeneous chip-multiprocessor. In *Proceedings of the 17th Hot Chips*, 2005.

Gunther, S H, F Binns, D M Carmean, and J C Hall. Managing the impact of increasing microprocessor power consumption. *Intel Technology Journal*, Q1, 2001.

Hamilton, Samuel Norman and Alex Orailoglu. Transient and intermittent fault recovery without rollback. In *Proceedings of the 13th International Symposium on Defect and Fault-Tolerance in VLSI Systems*, 1998.

Held, Jim, Jerry Bautista, and Sean Koehl. From a few cores to many: A tera-scale computing research overview, 2006. `ftp://download.intel.com/research/platform/terascale/terascale_overview_paper.pdf`. Viewed 6/30/2008.

Hill, Eric L., Mikko H. Lipasti, and Kewal K. Saluja. An accurate flip-flop selection technique for reducing logic ser. In *Proceedings of the 2007 International Conference on Dependable*

Systems and Networks (DSN), 2008.

Hill, Mark D. and Michael R. Marty. Amdahl's law in the multicore era. *IEEE Computer*, 41(7): 33–38, Jul 2008.

Hohmuth, Michael and Hermann Hartig. Pragmatic nonblocking synchronization for real-time systems. In *Proceedings of the General Track USENIX Annual Technical Conference*, 2001.

IBM. SPAR saves a "six-figure sum" using a virtualized environment for SAP ERP with IBM System p, Mar 2008. `http://www.ibm.com/common/ssi/fcgi-bin/ssialias?infotype=pm&subtype=ab&appname=SNDE_SP_SP_CHEN&htmlfid=SPC03020CHEN&attachment=SPC03020CHEN.PDF`. Viewed 7/23/2008.

Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation, May 2007.

Ípek, Engin, Meyrem Kirman, Nevin Kirman, and José F. Martínez. Core fusion: accommodating software diversity in chip multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA)*, 2007.

Ismaeel, Asad A. and Rakesh Bhatnagar. Test for detection & location of intermittent faults in combinational circuits. *IEEE Transactions on Reliability*, 46(2):269–274, Jun 1997.

Jann, Joefon, Luke M. Browning, and R. Sarma Burugula. Dynamic reconfiguration: Basic building blocks for autonomic computing on IBM pSeries servers. *IBM Systems Journal*, 42(1): 29–37, 2003.

Joseph, Russ. Exploring core salvage techniques for multi-core architectures. In *Proceedings of the Workshop on High Performance Computing Reliability Issues*, 2006.

Joseph, Russ, David Brooks, and Margaret Martonosi. Control techniques to eliminate voltage emergencies in high performance processors. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA)*, 2003.

Jourdan, Stephan J., John Alan Miller, and Namratha Jaisimha. Return address stack including speculative return address buffer with back pointers. U.S. Patent 6,898,699, May 2005.

Kalbarczyk, Zbigniew T., Ravishankar K. Iyer, Saurabh Bagchi, and Keith Whisnant. Chameleon: A software infrastructure for adaptive fault tolerance. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):560–579, 1999.

Kalla, Ron, Balaram Sinharoy, and Joel M. Tendler. IBM Power5 chip: a dual-core multithreaded processor. *IEEE Micro*, 24(2):40–47, 2004.

Kehl, Ted. Hardware self-tuning and circuit performance monitoring. In *Proceedings of the 1993 International Conference on Computer Design*, 1993.

Kim, Seongbeom, Dhruba Chandra, and Yan Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings of the 13th Annual International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2004.

Kumar, Rakesh, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of the 36th Annual International Symposium on Microarchitecture (MICRO)*, 2003.

Kumar, Rakesh, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, and Keith I. Farkas. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, 2004.

Kyo, Shorin, Takuya Koga, Lieske Hanno, Shouhei Nomoto, and Shin'ichiro Okazaki. A low-cost mixed-mode parallel processor architecture for embedded systems. In *Proceedings of the 21st Annual International Conference on Supercomputing (ICS)*, pages 253–262, 2007.

LaFrieda, Christopher, Engin Ípek, José F. Martínez, and Rajit Manohar. Utilizing dynamically coupled cores to form a resilient chip multiprocessor. In *Proceedings of the 2007 International Conference on Dependable Systems and Networks (DSN)*, 2007.

Lamport, Leslie, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

Laudon, James. Performance/watt: the new server focus. *ACM SIGARCH Computer Architecture News*, 33(4):5–13, 2005.

Lepak, Kevin M. and Mikko H. Lipasti. Temporally silent stores. In *Proceedings of the 10th International conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 30–41, October 2002.

Li, Man-Lap, Pradeep Ramachandran, Swarup Kumar Sahoo, Sarita V. Adve, Vikram S. Adve, and Yuanyuan Zhou. Understanding the propagation of hard errors to software and implications for resilient system design. In *Proceedings of the 13th International conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 265–276, 2008.

Li, Tao, Lizy Kurian John, Anand Sivasubramaniam, N. Vijaykrishnan, and Juan Rubio. Understanding and improving operating system effects in control flow prediction. In *Proceedings of the 10th International conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 68–80, 2002.

Li, Tong, Alvin R. Lebeck, and Daniel J. Sorin. Spin detection hardware for improved management of multithreaded systems. *IEEE Transactions on Parallel and Distributed Systems*, 17(6): 508–521, 2006.

Li, Tong, Dan Baumberger, David A. Koufaty, and Scott Hahn. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, pages 1–11, 2007.

Liang, Xiaoyao and David Brooks. Mitigating the impact of process variations on processor register files and execution units. In *Proceedings of the 39th Annual International Symposium on Microarchitecture (MICRO)*, 2006.

Litt, Timothe. Method and apparatus for CPU failure recovery in symmetric multi-processing systems. U.S. Patent 5,815,651, Sep 1998.

Magnusson, Peter, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, Feb 2002.

Marty, Michael R. and Mark D. Hill. Virtual hierarchies. *IEEE Micro*, 28(1):99–109, 2008.

Marty, Michael R. and Mark D. Hill. Virtual hierarchies to support server consolidation. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA)*, pages 46–56, 2007.

Mauer, Carl J., Mark D. Hill, and David A. Wood. Full-system timing-first simulation. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 108–116, 2002.

McEvoy, Dennis. The architecture of tandem's nonstop system. In *Proceedings of the ACM 1981 Conference*, 1981.

Mitra, S., M. Zhang, N. Seifert amd T. M. Mak, and K. Kim. Soft error resilient system design through error correction. In *Proceedings of the Very Large Scale Integration*, January 2006a.

Mitra, S., M. Zhang, N. Seifert, B. Gill, S. Waqas, and K. Kim. Combinational logic soft error correction. In *Proceedings of the 2006 International Test Conference (ITC)*, November 2006b.

Mitra, Subhasish, Norbert Seifert, Ming Zhang, Quan Shi, and Kee Sup Kim. Robust system design with built-in soft-error resilience. *IEEE Computer*, 38(2):43–52, 2005.

Mukherjee, Shubhendu S., Michael Kontz, and Steven K. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA)*, 2002.

Mukherjee, Shubhendu S., Joel Emer, and Steven K. Reinhardt. The soft error problem: An architectural perspective. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA)*, 2005.

Mullender, Sape J., Ian M. Leslie, and Derek McAuley. Operating-system support for distributed multimedia. In *Proceedings of the USENIX Summer 1994 Technical Conference*, pages 14–14, 1994.

Najm, Farid N. and Noel Menezes. Statistical timing analysis based on a timing yield model. In *Proceedings of the 41th Annual Conference on Design Automation*, pages 460–465, 2004.

Nakano, J., P. Montesinos, K. Gharachorloo, and J. Torrellas. ReViveI/O: Efficient handling of I/O in highly-available rollback-recovery servers. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture (HPCA)*, 2006.

Nanya, Takashi and Hendrik A. Goosen. The byzantine hardware fault model. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 8(11):1226–1231, Nov 1989.

Nellans, David, Rajeev Balasubramonian, and Erik Brunvand. A case for increased operating system support in chip multi-processors. In *Proceedings of the 2nd IBM Watson Conference on Interaction between Architecture, Circuits, and Compilers (p=ac^2)*, 2005.

Normand, Eugene. Single event upset at ground level. *IEEE Transactions on Nuclear Science*, 43 (6):2742–2750, Dec 1996.

Open Source Development Labs. Database test suite. `http://osdldbt.sourceforge.net/`. Viewed 7/28/2008.

Ousterhout, John K. Scheduling techniques for concurrent systems. In *Distributed Computing Systems*, 1982.

Popek, Gerald J. and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, 1974.

PostgreSQL Global Development Group. PostgreSQL. `http://www.postgresql.org/`. Viewed 7/28/2008.

Powell, Michael D. and T. N. Vijaykumar. Exploiting resonant behavior to reduce inductive noise. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, 2004.

Powell, Michael D., Mohamed Gomaa, and T. N. Vijaykumar. Heat-and-run: leveraging SMT and CMP to manage power density through the operating system. In *Proceedings of the 11th International conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2004.

Rafique, Nauman, Won-Taek Lim, and Mithuna Thottethodi. Architectural support for operating system-driven cmp cache management. In *Proceedings of the 15th Annual International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 2–12, 2006.

Rajwar, Ravi and James R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th Annual International Symposium on Microarchitecture (MICRO)*, pages 294–305, December 2001.

Ramanathan, R. M. Intel multi-core processors: Making the move to quad-core and beyond. *Technology@Intel Magazine*, 4(9):2–4, Dec 2006.

Reinhardt, Steven K. and Shubhendu S. Mukherjee. Transient fault detection via simultaneous multithreading. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA)*, 2000.

Reis, George A., Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I. August. Swift: Software implemented fault tolerance. In *Proceedings of the International Symposium on Code generation and Optimization (CGO)*, 2005a.

Reis, George A., Jonathan Chang, Neil Vachharajani, Ram Rangan, David I. August, and Shubhendu S. Mukherjee. Design and evaluation of hybrid fault-detection systems. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA)*, 2005b.

Rosenburg, B. Low-synchronization translation lookaside buffer consistency in large-scale shared-memory multiprocessors. In *Proceedings of the 12th Annual Symposium on Operating Systems Principles (SOSP)*, 1989.

Rotenberg, Eric. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In *Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing*, 1999.

Sankaralingam, Karthikeyan, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Doug Burger, Stephen W. Keckler, and Charles R. Moore. Exploiting ilp, tlp, and dlp with the polymorphous trips architecture. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)*, pages 422–433, 2003.

Satyanarayanan, Mahadev, Benjamin Gilbert, Matt Toups, Niraj Tolia, Ajay Surie, David R. O'Hallaron, Adam Wolbach, Jan Harkes, Adrian Perrig, David J. Farber, Michael A. Kozuch, Casey J. Helfrich, Partho Nath, and H. Andres Lagar-Cavilla. Pervasive personal computing in an internet suspend/resume system. *IEEE Internet Computing*, 11(2):16–25, 2007.

Schlichting, Richard D. and Fred B. Schneider. Fail-stop processors: an approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems*, 1(3):222–238, 1983.

Semiconductor Industry Association. International technology roadmap for semiconductors: Executive summary, 2005.

Sherwood, Timothy, Suleyman Sair, and Brad Calder. Phase tracking and prediction. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)*, 2003.

Shivakumar, Premkishore, Michael Kistler, Stephen W. Keckler, Doug Burger, and Lorenzo Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks (DSN)*, 2002.

Shyam, Smitha, Kypros Constantinides, Sujay Phadke, Valeria Bertacco, and Todd Austin. Ultra low-cost defect protection for microprocessor pipelines. In *Proceedings of the 12th International conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.

Skadron, Kevin, Pritpal S. Ahuja, Margaret Martonosi, and Douglas W. Clark. Improving prediction for procedure returns with return-address-stack repair mechanisms. In *Proceedings of the*

31st Annual International Symposium on Microarchitecture (MICRO), pages 259–271, 1998.

Skadron, Kevin, Mircea R. Stan, Wei Huang, Sivakumar Velusamy, Karthik Sankaranarayanan, and David Tarjan. Temperature-aware microarchitecture. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)*, 2003.

Slegel, Timothy J., Robert M. Averill III, Mark A. Check, Bruce C. Giamei, Barry W. Krumm, Christopher A. Krygowski, Wen H. Li, John S. Liptay, John D. MacDougall, Thomas J. McPherson, Jennifer A. Navarro, Eric M. Schwarz, Kevin Shum, and Charles F. Webb. IBM's S/390 G5 microprocessor design. *IEEE Micro*, 19(2):12–23, 1999.

Smith, James and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann, 2005.

Smolens, Jared C. *Fingerprinting: Hash-Based Error Detection in Microprocessors*. PhD thesis, Carnegie Mellon University, 2008.

Smolens, Jared C., Brian T. Gold, Jangwoo Kim, Babak Falsafi, James C. Hoe, and Andreas G. Nowatzyk. Fingerprinting: bounding soft-error detection latency and bandwidth. In *Proceedings of the 11th International conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2004.

Smolens, Jared C., Brian T. Gold, Babak Falsafi, and James C. Hoe. Reunion: Complexity-effective multicore redundancy. In *Proceedings of the 39th Annual International Symposium on Microarchitecture (MICRO)*, 2006.

Smolens, Jared C., Brian T. Gold, James C. Hoe, Babak Falsafi, and Ken Mai. Detecting emerging wearout faults. In *Proceedings of the IEEE Workshop on Silicon Errors in Logic - System Effects (SELSE)*, 2007.

Sohi, Gurindar S., Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA)*, pages 414–425, 1995.

Sorin, Daniel J., Milo M. K. Martin, Mark D. Hill, and David A. Wood. Safetynet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA)*, 2002.

Suh, G. E., L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *Journal of Supercomputing*, 28(1):7–26, 2004.

Sun Microsystems. Sun enterprise 10000 server: Dynamic system domains, 1999. `http://www.sun.com/servers/white-papers/domains.html`. Viewed 6/09/2008.

Sun Microsystems, Inc. Open solaris cpu.c: cpu_offline(), 2008. `http://cvs.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/uts/common/os/cpu.c`. Viewed 6/09/2008.

Sun Microsystems, Inc. Sun fire high-end and midrange systems dynamic reconfiguration user's guide. `http://docs.sun.com/app/docs/doc/819-1501`. Viewed 12/19/2007.

Sun Microsystems, Inc. Open solaris cpu.c: sbdp_cpu_poweroff(), 2006. `http://cvs.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/uts/sun4u/serengeti/io/sbdp_cpu.c`. Viewed 6/09/2008.

Sun Microsystems, Inc. *UltraSPARC III Cu User's Manual*, 2003.

Torrellas, Josep, Andrew Tucker, and Anoop Gupta. Evaluating the performance of cache-affinity scheduling in shared-memory multiprocessors. *Journal of Parallel and Distrubuted Computing*, 24(2):139–151, 1995.

Tschanz, James W., Siva G. Narendra, Yibin Ye, Bradley A. Bloechel, Shekhar Borkar, and Vivek De. Dynamic sleep transistor and body bias for active leakage power control of microprocessors. *IEEE Journal of Solid-State Circuits*, 38(11), 2003.

Uhlig, Rich, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C. M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kagi, Felix H. Leung, and Larry Smith. Intel virtualization technology. *IEEE Computer*, 38(5), 2005.

Uhlig, Volkmar, Joshua LeVasseur, Espen Skoglund, and Uwe Dannowski. Towards scalable multiprocessor virtual machines. In *Proceedings of the 3rd Virtual Machine Research and Technology Symposium*, 2004.

U.S. Environmental Protection Agency. Report to congress on server and data center energy efficiency, Aug 2007. `http://www.energystar.gov/index.cfm?c=prod_development.server_efficiency_study`. Viewed 7/28/2008.

VMware. ESX Server - best practices using VMware virtual SMP. `www.vmware.com/pdf/vsm_best_practices.pdf`. Viewed 5/03/2006.

VMware. VMware ESX Server 3 ready time observations, 2006a. `www.vmware.com/pdf/esx3_ready_time.pdf`. Viewed 7/23/2008.

VMware. Reducing server total cost of ownershipwith VMware virtualization software, 2006b. `http://www.vmware.com/pdf/TCO.pdf`. Viewed 7/23/2008.

VMware. Statistics canada harvests $1.3 million in savings, 2006c. `http://www.vmware.com/pdf/statistics_canada.pdf`. Viewed 7/23/2008.

Walcott, Kristen R., Greg Humphreys, and Sudhanva Gurumurthi. Dynamic prediction of architectural vulnerability from microarchitectural state. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA)*, pages 516–527, 2007.

Waldspurger, Carl A. Memory resource management in VMware ESX Server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

Weaver, Chris and Todd M. Austin. A fault tolerant approach to microprocessor design. In *Proceedings of the 2001 International Conference on Dependable Systems and Networks (DSN)*, 2001.

Wells, Philip M. and Gurindar S. Sohi. Serializing instructions in system-intensive workloads: Amdahl's law strikes again. In *Proceedings of the 14th International Symposium on High-Performance Computer Architecture (HPCA)*, 2008.

Wells, Philip M., Koushik Chakraborty, and Gurindar S. Sohi. Hardware support for spin management in overcommitted virtual machines. In *Proceedings of the 15th Annual International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2006.

Wells, Philip M., Koushik Chakraborty, and Gurindar S. Sohi. Adapting to intermittent faults in future multicore systems (poster). In *Proceedings of the 16th Annual International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2007a.

Wells, Philip M., Koushik Chakraborty, and Gurindar S. Sohi. On hiding multicore complexity from system software (position paper). In *Proceedings of the Workshop on Operating System support for Heterogenous Multicore Architectures (OSHMA)*, 2007b.

Wells, Philip M., Koushik Chakraborty, and Gurindar S. Sohi. Adapting to intermittent faults in multicore systems. In *Proceedings of the 13th International conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 255–264, 2008.

Wenisch, Thomas F. Personal communication, Feb 2008.

Wenisch, Thomas F. and Roland E. Wunderlich. SimFlex: Fast, accurate and flexible simulation of computer systems (tutorial). In *Proceedings of the 38th Annual International Symposium on*

Microarchitecture (MICRO), 2005.

Wenisch, Thomas F., Stephen Somogyi, Nikolaos Hardavellas, Jangwoo Kim, Anastassia Ailamaki, and Babak Falsafi. Temporal streaming of shared memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA)*, 2005.

Whitaker, Andrew, Marianne Shaw, and Steven D. Gribble. Scale and performance in the Denali isolation kernel. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

Wisniewski, Robert W., Leonidas Kontothanassis, and Michael L. Scott. Scalable spin locks for multiprogrammed systems. Technical Report TR454, University of Rochester, Rochester, NY, USA, 1993.

Wonyoung, Kim, Gupta Meeta, Wei Gu-Yeon, and Brooks David. System level analysis of fast, per-core dvfs using on-chip switching regulators. In *Proceedings of the 14th International Symposium on High-Performance Computer Architecture (HPCA)*, February 2008.

Yeh, Y. C. Triple-triple redundant 777 primary flight computer. In *Proceedings of the 1996 IEEE Aerospace Applications Conference*, pages 293–307, Feb 1996.

Zahorjan, J., E. D. Lazowska, and D. L. Eager. The effect of scheduling discipline on spin overhead in shared memory parallel systems. *IEEE Transactions on Parallel and Distributed Systems*, 2(2):180–198, 1991.

Zhang, Ming, Subhasish Mitra, T. M. Mak, Norbert Seifert, Nicholas J. Wang, Quan Shi, Kee Sup Kim, Naresh R. Shanbhag, and Sanjay J. Patel. Sequential element design with built-in soft error resilience. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(12): 1368–1378, Dec 2006.

Zhou, Huiyang. A case for fault tolerance and performance enhancement using chip multiprocessors. *IEEE Computer Architecture Letters*, 5(1):6, 2006.

Zilles, Craig and Gurindar Sohi. Master/slave speculative parallelization. In *Proceedings of the 35th Annual International Symposium on Microarchitecture (MICRO)*, pages 85–96, 2002.