Mixed-Mode Multicore Reliability

Philip M. Wells
Google, Inc.
pwells@google.com

Koushik Chakraborty
Electrical and Computer Engineering
Utah State University
kchak@engineering.usu.edu

Gurindar S. Sohi
Computer Sciences Department
University of Wisconsin, Madison
sohi@cs.wisc.edu

Abstract

Future processors are expected to observe increasing rates of hardware faults. Using Dual-Modular Redundancy (DMR), two cores of a multicore can be loosely coupled to redundantly execute a single software thread, providing very high coverage from many difference sources of faults. This reliability, however, comes at a high price in terms of per-thread IPC and overall system throughput.

We make the observation that a user may want to run both applications requiring high reliability, such as financial software, and more fault tolerant applications requiring high performance, such as media or web software, on the same machine at the same time. Yet a traditional DMR system must fully operate in redundant mode whenever any application requires high reliability.

This paper proposes a Mixed-Mode Multicore (MMM), which enables most applications, including the system software, to run with high reliability in DMR mode, while applications that need high performance can avoid the penalty of DMR. Though conceptually simple, two key challenges arise: 1) care must be taken to protect reliable applications from any faults occurring to applications running in high performance mode, and 2) the desire to execute additional independent software threads for a performance application complicates the scheduling of computation to cores. After solving these issues, an MMM is shown to improve overall system performance, compared to a traditional DMR system, by approximately 2X when one reliable and one performance application are concurrently executing.

Categories and Subject Descriptors B.8.2 [*Performance and Reliability*]: Performance Analysis and Design Aids

General Terms Reliability, Design, Performance

Keywords Multicore, Dual-Modular Redundancy

ASPLOS'09, March 7–11, 2009, Washington, DC, USA.

1. Introduction

As technology scales, the components of future multicore processors become less reliable because smaller transistors and wires are more susceptible to hardware faults. These faults are caused by a variety of factors including high-energy particle strikes, manufacturing process variation, device wear-out, and are affected by temperature and voltage fluctuations [5, 6, 8, 10, 22, 24]. Hardware faults can cause transient, intermittent, or permanent computational errors, which then manifest within software in a multitude of ways [15, 35].

Many circuit and microarchitectural techniques can tolerate hardware faults to a degree, while preserving the view of continuous, reliable hardware operation that system and application software have come to expect. Unfortunately these techniques tend to either have low overhead, but only modest fault coverage (e.g., [7, 25, 30]), or excellent fault coverage, but high overhead (e.g., [12, 14, 19, 29]). One technique in particular, Dual-Modular Redundancy (DMR), falls into the latter category: It can provide very high coverage from many different sources of faults, at nearly a 4X reduction in throughput in some cases. Nonetheless, certain applications and users already desire high reliability and the peace of mind that comes with the use of DMR — and are willing to pay extra in terms of performance and machine cost [1, 4, 18, 26].

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. But 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 nearly *all* software needs to run with DMR. In the meantime, however, we expect many applications (or portions of many applications) to remain sufficiently reliable while using only low-overhead techniques, leading a user to run multiple applications with differing reliability requirements at the same time.

For example, a desktop user may wish to run both a media application and a personal finance application at the same time. Media applications tend to be insensitive to moderate

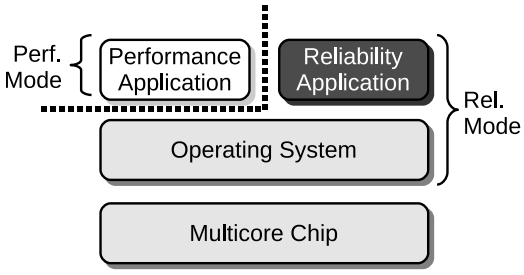


Figure 1. Mixed-Mode with a Single-OS

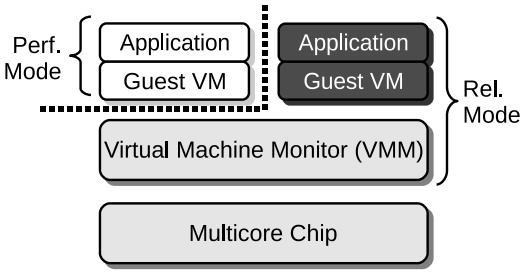


Figure 2. Mixed-Mode Consolidated Server

levels of hardware faults [23], but a user may be willing to sacrifice a certain degree of performance to ensure the integrity of their financial data. Another example is a consolidated server hosting multiple guest virtual machines (VMs) for multiple customers with different service-level agreements. Some customers may require very high reliability (at a premium price). Other customers may demand more performance at an economy price, but are willing to tolerate occasional data corruption and down-time due to crashes.

Such scenarios result in a system where one set of applications, the *reliable* applications, need the protection of DMR, while another set, the *performance* applications, need the high performance available through independent utilization of all of the computing resources. To enable such a system, this paper proposes the design of a *Mixed-Mode Multicore* (MMM) that can execute both types of applications simultaneously on the same machine. The basics of an MMM seem simple: Use DMR for reliable applications, and turn off DMR for performance applications.

Although several architectural DMR proposals suggest that DMR can easily be turned on and off (e.g., [19, 32]), a key observation of this paper is that dynamically switching between DMR and non-DMR within a single system is not as straightforward as it might appear. In particular, we observe that 1) care must be taken to preserve the integrity of reliable applications' memory and register state, and 2) the desire to execute additional independent software threads for a performance application complicates the scheduling of software threads to cores. As part of the solution to the first problem, we propose to maintain a small amount of redundancy for non-DMR applications by re-validating permission for any stores that miss in the L1 cache. To address the second problem, we propose to leverage hardware virtualization techniques to flexibly and quickly assign threads to cores. The resulting MMM system is able to protect the integrity of reliable applications needing DMR, significantly improve overall performance of applications that do not, and preserve a simple interface to the system software (i.e., operating system or virtual machine monitor).

2. Mixed-Mode Overview

The primary objective of an MMM is simple: provide reliability for software that requires it, and improve the performance of software that does not. Figure 1 depicts this basic objective for an MMM running two applications on a single-OS system. One application always requires high performance, and the other requires reliability. Both the reliable application and the operating system must be executed in reliable mode for reasons described in Section 3.4.2. An MMM can also offer differentiated service to different VMs within a consolidated server under the control of a Virtual Machine Monitor (VMM). An example of such a system is depicted in Figure 2. In this case, one guest Virtual Machine (VM) (including the OS and applications) requires reliability, while the other guest VM requires performance. Again, the highest privileged software (in this case the VMM) must always run in reliable mode, while less privileged software has the option of running with higher performance.

Additional objectives of an MMM system are 1) to isolate reliable applications from any hardware faults affecting software executing in performance mode, and 2) to perform mixed-mode operation with only minor changes to the system and application software.

We propose two different ways of handling high-performance mode in an MMM system, *MMM-IPC* and *MMM-TP*. The simpler method, MMM-IPC idles the redundant cores, eliminating verification and synchronization delays, and improving the Instructions per Cycle (IPC) of each thread by 34-48%.[1] When in high-performance mode, MMM-TP uses all available cores to independently run additional software threads of high-performance applications, improving throughput by 2.5–4 times. In either system, different cores can be in different modes at different times.

2.1 Mixed-Mode Challenges

Although the objectives are straightforward, the implications of running different applications in different modes on the same machine are less so. Two key challenges in particular make building an MMM more complicated than simply turning off DMR when high performance is required. First, the memory and register state of reliable applications must

[1] By placing idle redundant cores in a low-power sleep state, *MMM-IPC* would also likely either reduce the chip's power consumption, or allow the remaining cores to use the excess power budget by increasing their frequency, for example. We do not evaluate these power options in this paper, however.

be isolated from any hardware faults that may occur when running performance applications. Second, system software itself must operate in reliable mode, even when performing operations on behalf of a performance application — a feat which is greatly complicated by the desire to run additional software threads of a performance application. A brief discussion of these two challenges and an overview of the proposed solutions are provided below.

Memory and Register Protection In an MMM, the first key challenge is to prevent high-performance applications from corrupting the state of reliable applications. In a fault-free system this protection is achieved through a combination of now-standard software mechanisms and policies (e.g., page-based memory protection) and hardware support (e.g., TLB permission checks). Further protection can be achieved using additional software mechanisms such as *Overshadow* [9]. However, these existing mechanisms make the assumption that hardware itself is reliable.

If hardware faults do occur, these protection mechanisms remain sufficient if the processor is always operating in DMR mode, since the redundant execution ensures that a hardware fault on one core is detected and corrected before the application's state is updated.

The problem arises when we allow the processor to avoid DMR mode for certain software. In this case, simple faults in certain hardware structures will go undetected, allowing malicious, buggy, or even correct software to write to physical addresses that are not owned by the application. For example, a bit flip in the *privileged mode* bit, checking logic, or TLB array can result in the successful translation of an invalid virtual address. If the resulting physical address contains state used by other software components, such as ostensibly reliable applications or the system software, these other components will become corrupted.

The primary line of defense we adopt for an MMM is the use of a small hardware structure called the Protection Assistance Buffer (PAB). When a core is operating in high-performance mode, the PAB redundantly verifies the permission of stores emanating from the core. If a physical address is presented to the memory system that is not owned by the high-performance application, an exception can be generated to notify the system software before corruption occurs.

System Software Protection The second key challenge to mixed-mode operation is that *all* privileged software must execute in reliable mode, even when called from a high-performance application. The reason is that when performing a system call, or other service such as paging, the system software updates its own internal state, which is used when performing services for both reliable and high-performance applications. Executing privileged software in reliable mode prevents the system from crashing due to faults, as well as protects the integrity of system services performed on behalf of reliable applications.

This observation has an important implication for the scheduling of software threads on the physical cores: Every time a thread of a high-performance application encounters a system call, page fault, or interrupt, a mode transition is triggered. Reliable mode must be entered when these events occur by appropriating another core to use as a redundant pair. As a result, transitions from performance mode to reliable mode must be performed with low overhead, since some applications enter the operating system every 200k cycles (Section 5.3). This rapid transition becomes even more challenging when the redundant cores needed for reliable mode are currently being used to independently execute additional software threads of a high-performance application.

To enable a high performance application to utilize all on-chip cores, while still switching to reliable mode whenever the code running on a core enters the system software, we propose to leverage our multicore virtualization techniques [34]. These techniques allow the chip to decouple the *physical cores* from the *virtual processors* onto which the system software schedules threads, providing flexibility in the mapping of computation onto one or more cores.

3. Mixed-Mode Implementation

The previous section outlined the objectives and challenges of mixed-mode operation. This section presents the implementation details of a mixed-mode multicore (MMM). First, we mention basic assumptions about the target multicore and provide an overview of the Reunion DMR proposal we leverage for this work [27, 29]. We then focus on the other aspects mixed-mode implementation: the proposed hardware/software interface, the mechanisms for protecting memory and system state, and the use of virtualization.

3.1 Target Multicore Assumptions

For this work, we assume a 16-core processor, with out-of-order cores and a 3-level cache hierarchy. Cores are each provided with private, write-through L1 caches and a private L2 cache. Similar to the IBM Power5 [13] and AMD quad-core Opteron [11] processors, we use a shared L3 cache (on-chip, unlike Power5) that maintains *exclusion* with the private L2s. Further details of the target multicore are provided in the methodology (Section 4).

We assume that reliable mode is implemented via Dual-Modular Redundancy (DMR), while performance mode uses only a single core to run each thread of an application. In either mode, however, designers may still choose to implement numerous circuit or microarchitectural techniques within each core.

As with other architecture-level reliability proposals (e.g., [7,12,14,19,25,27,29,33,38]), we assume that most memory hierarchy components maintain reliability by implementing other techniques such as Error Correcting Codes (ECC). The exception is the private L1 caches, which are not assumed to be reliable in this work. We believe this assumption is rea-

sonable because 1) techniques such as ECC are much more effective for these regular, repeated structures than they are for combinational logic within the cores, 2) the additional delay introduced by techniques like ECC is more easily tolerated in the caches (especially L2 and beyond), and 3) although caches have traditionally been more susceptible to certain hardware faults than logic circuits, Shivakumar, et al., project per-chip fault rates in logic to increase much faster, catching and surpassing the per-chip rates for caches and other SRAM components by 2011 [24].

3.2 Reunion Overview

Reunion [27, 29] is a form of "loose lock-stepping," which defines a logical processing pair as two cores that 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*, an instruction computes a *fingerprint*, or hash of its results, and sends this fingerprint to the other core. Each instruction waits in the *Check* stage until it receives the other core's fingerprint for the same instruction. The instruction is then committed to the architected state of each core. A single fingerprint can capture all outputs, branch targets, and store addresses and values for multiple instructions.

A mute core is not required to maintain coherence with the rest of the system. Instead, all requests emanating from the private cache hierarchy of a mute core do not change the state of the line in the directory or any other caches. The cache hierarchy makes a best-effort attempt to provide the correct value. Should that attempt fail, a fingerprint mismatch will occur, which will be detected and corrected similar to a transient fault.

3.3 Hardware/Software Interface

This work proposes to implement the reliability mechanisms in a thin virtual machine layer beneath the ISA. The chip exposes two new pieces of information to software via the ISA. First, it exposes that the chip has multiple operating modes with different levels of reliability. Second, it exposes the fact that software is responsible for determining the desired mode, and can do so dynamically for each OS-visible *virtual processor* (VCPU).

The basis of the mixed-mode 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 high performance, or 3) operate with high performance only when executing non-privileged (user or guest VM) software. This paper addresses the issues when mixing the first and third modes.

When the privileged software is about to context switch to an application (or guest VM) which requires high performance, it writes this register to indicate the requirements of the software running on that VCPU. This register is only writable by privileged software.

We have intentionally not defined the OS/application interface for using this register. Our evaluation assumes that an individual application runs from start to finish with either high reliability or high performance, possibly specified by an administrator. However, some applications may desire a finer granularity of control. To support this usage, new system calls to change the reliability mode, and system and compiler support to specify which pages of memory can be accessed in performance mode, would likely be necessary. We leave a more detailed investigation for future work.

3.4 Protecting System Integrity

When performing mixed-mode reliability, a key challenge is to protect the integrity of the system while executing in performance (non-DMR) mode. As described in Section 2.1, hardware faults can potentially allow buggy, malicious, and even correct software operating in performance mode to corrupt the state of other applications or the system software. This section describes the mechanisms to prevent such hardware faults from corrupting the integrity of the system or applications requiring reliability.

3.4.1 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.

To prevent arbitrary software from accessing memory for which it does not have permission, we propose to duplicate the TLB protection check *outside of the core* before allowing a core running in high-performance mode to store data to its L2 cache.[2] This check requires system software to identify pages that are only allowed to be written by reliable applications through the proposed Protection Assistance Table (PAT). Hardware uses this information to perform a redundant check through the proposed Protection Assistance Buffer (PAB). The PAT and PAB are described below.

An alternate option may be to replicate these structures, or harden them by making the transistors larger, slower, and/or liberally applying other circuit-level reliability techniques. The problems with replicating or hardening are 1) all of these structures are on the critical load path, and applying these techniques could impact the latency of every memory access, and 2) every hardware structure that carries or stores an address from the TLBs until the L2 cache must be fully hardened or replicated, impacting area, perfor-

[2] 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 left for future work.

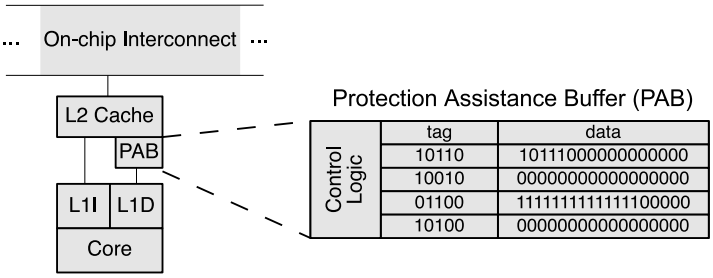


Figure 3. The Structure of the Protection Assistance Buffer (PAB)

mance, and power. Despite these drawbacks, hardening may be a viable option for a particular design, making existing memory protection mechanisms sufficient.

Protection Assistance Table The *Protection Assistance Table* (PAT) is similar to an inverse page table: for each physical page in the system, a "1" entry indicates that page can only be accessed by applications executing in reliable mode, and a "0" entry indicates that page can potentially be accessed by any software, including high-performance applications. At one bit per 8KByte page, the PAT thus requires 16MBytes for one TByte of physical memory. The PAT resides in cacheable memory. 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 (e.g, on a page fault). We leave a detailed study of the interactions with large page sizes for future work.

Protection Assistance Buffer A hardware structure, called the *Protection Assistance Buffer* (PAB), acts as a cache of PAT entries. It is used to redundantly verify a store's permission after the TLB, but before accessing the rest of the system. For a core executing in performance mode, the PAB is accessed either before, or in parallel with, the L2 cache. When in reliable mode, the PAB is not used. A match in the PAB (and TLB) means an access has the proper permissions; a permission failure in either the PAB or TLB triggers an exception, which the system software can interpret as it wishes. Thus, the PAB and TLB provide redundancy for each other.

Figure 3 shows a diagram of PAB placement and structure for one core in the system. This structure is organized much like a cache, with a physically tagged and indexed array containing 64 Bytes (one cache-line worth) of PAT entries. The PAB operates on physical addresses, since the virtual address is assumed to no longer be known for a store accessing the L2 cache. A 128-entry PAB requires 8.2KBytes, can map 512MBytes of physical memory (for SPARC addressing), and represents a storage overhead of 1.6% compared to the private L2 cache.

L1 write-throughs can either examine the PAB in parallel with their access to the L2 cache, aborting the L2 access should the PAB indicate an invalid store, or wait to access the L2 cache until the store is validated by the PAB. This serial lookup incurs additional latency for stores, but can simplify the L2 controller logic. Experiments in Section 5 evaluate both parallel and serial lookups.

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

3.4.2 Protecting System Software

As described in Section 2.1, software at the highest privilege level is always executed in reliable mode. Privileged software may be the OS in a single-OS system, or the software VMM or hypervisor in a consolidated server system (see Figures 1 and 2). As a consequence, a core operating in performance mode cannot execute any privileged instructions without causing a transition to reliable mode.

For a consolidated server, we treat each guest VM, including OS and applications, as a single entity. We thus protect the VMM and other, reliable guest VMs from faults occurring to a high-performance VM, by ensuring that all traps to the VMM (running at the highest privileged level) execute in reliable mode. We assume that we do not need to protect the OSs running inside individual guest VMs (though our proposal could be modified to do so), since a fault in a performance guest VM will not affect the reliable VMs.

3.4.3 Protecting Registers During Mode Transitions

Unprivileged software is not allowed to write most privileged registers. However, a fault can cause unprivileged software to corrupt a privileged register, or erroneously allow buggy or malicious software to write one of these registers. State for reliable applications is always replicated, and faults are detected before they are committed to architected state. To protect against such faults that might occur during performance mode, care must be taken during mode transitions to replicate and verify privileged state. Each core contains a small hardware state machine to handle the required steps of these mode transitions.

In MMM-IPC, two types of mode transitions can occur: 1) a pair of cores leaves DMR mode, because of a high-performance application returning from a system call, for example, or 2) a pair of cores enters DMR mode when beginning a system call or other privileged operation, for example. When leaving DMR, the cores need only store their

privileged state to the cache hierarchy for later use, using a reserved portion of the physical address space (i.e., "scratchpad space"). Entering DMR, however, is more involved. The vocal core (previously running in performance mode) has all of the necessary state, but the mute core does not, since execution has progressed on the vocal core alone. The vocal core stores all of its state to the cache hierarchy. The mute loads, from the scratchpad space, its own previously saved (redundant) copy of the privileged registers, the vocal's copy of the user registers, and finally the privileged registers of the vocal core, verifying them with its own copy. This final check prevents faults from corrupting the vocal core's privileged state when operating in performance mode. Certain registers, such as exception conditions, can change during unprivileged execution, and should be sanity checked instead.

For MMM-TP, the steps can be slightly different because the hardware scheduler (described in Section 3.5) might have scheduled an independent software thread from a different VCPU onto the mute core while the vocal core was operating in performance mode. In particular, when leaving DMR mode, both cores must store all (not just privileged) state to the scratchpad space, and the mute core must flush its caches of any incoherent data resulting from Reunion's mute incoherence policy.

An interesting issue arises, however, when performing loads and stores of a VCPU's state during a mode switch. These requests, even from a mute core, must be processed as normal, even though a mute typically does not perform coherent memory requests. This means that the cache at a mute core can simultaneously consist of both incoherent lines brought into the cache via normal incoherent Reunion operation, 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.[3] As a result of mixing coherent and non-coherent lines, flushing lines when MMM-TP leaves DMR mode may not be as simple as gang-invalidating the cache. Instead, cache lines must be inspected one by one to see if they are dirty and need to be written back, a potentially costly operation (see Section 5.3).[4]

3.5 Scheduling and Virtualization

In any MMM, or architecture-level DMR system, the chip exposes a certain number of virtual processors (VCPUs) to the operating system (OS), and is then responsible for mapping those VCPUs onto the physical cores. In the case of a standard DMR system, the chip statically (e.g., [12]) or dynamically (e.g., [14]) maps one VCPU onto a pair of cores. The two MMM systems we propose in this paper, *MMM-IPC* and *MMM-TP*, perform this mapping in two different ways. Like a traditional DMR system, MMM-IPC statically maps one VCPU to a pair of cores, and then simply idles the redundant core when the software running on that core enters performance mode. By eliminating the verification and synchronization latency of DMR, MMM-IPC can improve the IPC of each VCPU running in performance mode.

MMM-TP, on the other hand, aims to also improve throughput by scheduling an independent VCPU to run on the otherwise idle core. To enable this flexibility, while preserving a simple interface to the system software, MMM-TP employs our previously proposed multicore virtualization techniques [34]. The reason such multicore virtualization is useful for MMM-TP is that the number of cores required to execute each VCPU changes dynamically depending on the whims of the system software's scheduler: It can schedule software requiring reliability on all VCPUs, software requiring performance on all VCPUs, or any combination, which can rapidly change.

In MMM-TP, a hardware/firmware layer manages the mapping of VCPUs to cores similar to, but much simpler and at a lower level than, a traditional software Virtual Machine Monitor (VMM). Hardware supported multicore virtualization provides two important services. First, hardware support for maintaining VCPU state enables low-overhead migration of a VCPU from one core (or pair of cores) to another, unbeknownst to the system software. Second, virtualization allows the cores to be *overcommitted*, such that more VCPUs are exposed to system software than there are available *pairs* of physical cores. When many VCPUs wish to execute in DMR mode, some of them must be paused. But when many VCPUs do not require DMR, all of the cores can be used for independent VCPUs to increase throughput. An overcommitted mixed-mode system is depicted in Figure 4. Here, one VCPU (V2) is executing a software threads that requires reliability, and is executing redundantly on cores C2 and C3. V3 is paused since there are no cores available to execute it. The other VCPUs are all executing threads that require performance. This technique operates in the same manner whether the system software is a single OS, or a software VMM in its own right, performing another layer of virtualization among its guest VMs. A major advantage of choosing Reunion as the DMR system for an MMM is that it allows any core to operate as a vocal or mute for any other core, significantly easing the scheduling challenges that arise from MMM-TP.

4. Experimental Methodology

4.1 Simulation

For the experiments in Section 5, we use Virtutech Simics [16], an execution driven, full-system simulator which functionally models a *SunFire 6800* server in sufficient detail to boot unmodified operating systems. We use Simics as a functional simulator only, and model timing using Simics MAI with our own cycle-accurate processor and memory hierarchy module.

[3] A *Coherent* bit must also be added to each request sent to the L3.

[4] The number of dirty lines will never exceed the size of the VCPU state, about 2.3KBytes for SPARC.

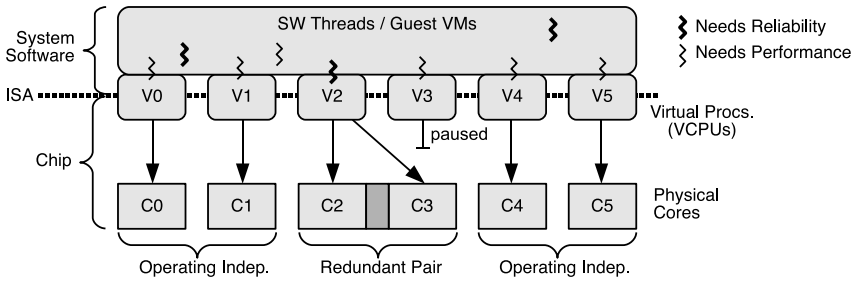


Figure 4. Improving Throughput in a Mixed-Mode Multicore by Overcommitting Cores

We model each core as having an 8-stage pipeline, out-of-order, 2-wide issue, an 128-entry instruction window, and operating at 3 GHz. The load/store queue contains entries for 32 loads and 32 stores. The pipeline is 9 stages when using Reunion. The chip consists of 16 cores. Located with each core is a split 16k, 2-way, write-through I&D caches, and a unified 512k, 4-way private L2. We also model an 8MB, 16-way, shared L3 that is exclusive with the L2s, and has a 55-cycle load to use latency. Cores maintain coherence via a MOSI directory protocol over a point-to-point interconnect with an average 10 cycle latency. The L2 directory uses shadow tags, which are co-located with each L3 bank. Main memory is 350 cycles load-to-use, with 40 GB/sec of off-chip bandwidth. In order to not overstate the penalty of DMR, we model a hardware-filled TLB, like [29].

A dedicated fingerprint network with a 10-cycle latency is assumed, as was done in the original Reunion proposal [29]. Reunion's "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 5.2 assumes a parallel PAB and L2 access. A PAB latency of two cycles is used for serial PAB access in the second set. Although the full software overhead of PAT manipulation is not modeled, it is expected to be very minimal, since it should only be necessary only when page mappings change.

To implement virtualization, we evaluate a thin virtual-machine layer implemented primarily in hardware, similar to [34]. We do model the overhead of maintaining VCPU state. This task is performed by storing the running VCPU's state in a portion of cacheable physical memory and loading it later from the same or a different core. The state can be transparently migrated to other cores using the on-chip coherence protocol.

Applications We use several workloads for these experiments, all of which are running on Solaris 9. `Apache` and `Zeus` are static web servers driven by the Surge [2] client. We do not use any think time in the Surge client. `OLTP` is a TPC-C-like workload using IBM's DB2 database. The database is scaled down from TPC-C specification to about 800MB and runs 192 concurrent user threads with no think time. `pgoltp` uses the PostgreSQL database version 8.1.3 [21] to run TPC-C-like queries from the OSDL dbt2 test suite [20]. The database is scaled similarly to `OLTP`. `pgbench` runs TPC-B like queries on the PostgreSQL database [21]. `pmake` is a parallel compile of PostgreSQL using GNU make and the Sun Forte Developer 7 C compiler. We do not include serial phases.

Each simulation runs for 100 million cycles. Due to workload variability, we simulate multiple runs and report average results with 95% confidence intervals. We use *committed user instructions* as our metric for 'work' in all experiments. User commits has been shown to correlate well with other 'work' metrics, such as workload transactions [37].

Consolidated Server Experiments Experiments in Sections 5.2 and 5.3 evaluate a mixed-mode consolidated server where one guest VM is running an application that requires reliability, and a second guest VM is running an application that required performance (similar to Figure 2).

Each consolidated workload combines two guest VMs running the applications described above. In each workloads, the same application is running in both guests. Each guest VM is configured with its own I/O devices and physical memory space, but VMs dynamically share the processors and caches. We are assuming the use of a software VMM, similar to VMWare ESX Server, which virtualizes I/O, memory, and privileged instructions. Since we do not have access to a software VMM which supports our simulated SPARC platform, we are unable to model the overhead of virtualizing memory or I/O. The two guest OSs are allocated enough physical memory so that the VMM does not need to swap *real* memory. This methodology for simulating consolidated servers is similar to that used by in prior research [17, 34].

For evaluating MMM-IPC, two guest VMs are running, each of which exposes 8 VCPUs to the VMM. The first VM runs redundantly on all 16 cores, and the second runs in performance mode using only 8 cores. Guests are gang scheduled using a 1ms (3 million cycle) timeslice. Using a longer timeslice with this methodology can create performance inconsistencies due to OS timers and interrupts.

For evaluating MMM-TP, we again model two guest VMs. The reliability VM runs 8 VCPUs on 16 cores, and the performance VM runs 16 VCPUs on all 16 cores. To avoid

comparing results using different workload checkpoints, we implement the 16 VCPU guest as two co-scheduled 8 VCPU guests running the same application. This methodology pessimistically inflates the memory requirements of the high-performance guest, but optimistically assumes linear scaling of applications.

Given our SPARC infrastructure, there is also no way of evaluating switches to and from the system VMM. Thus consolidated server workloads only switch to or from reliable mode during at the end of each VMs timeslice, however, we do investigate the overheads of more frequent switching in Section 5.3.

5. Evaluation

In this section, we first analyze the throughput and IPC overheads of DMR. We then demonstrate the effectiveness, and examine several design trade-offs, of mixed-mode reliability. We use a mixed-mode consolidated server for this analysis, but also present results to gauge the benefits and overheads of mixed-mode operation on a single-OS system.

5.1 Overhead of Dual Redundancy

To determine the performance overheads of DMR, this section compares three systems. The first, *No DMR 2X* represents a non-DMR system using all 16 cores for running independent OS-visible VCPUs. The second *No DMR*, represents a non-DMR system running eight VCPUs on only eight cores. The other eight cores are idle. The third is our re-implementation of Reunion [29], which is running the same eight VCPUs as *No DMR*, but running them redundantly across all 16 cores.

Figure 5 examines the performance of these three systems. Data are normalized to the *No DMR 2X* configuration. Figure 5(a) shows the per-thread IPC impact, and Figure 5(b) shows the overall throughput impact. In Figure 5(a), per-thread IPC is measured as the average of each active VCPU's *User IPC*, or the number of *User* instructions committed divided by the *total* number of cycles. The *No DMR* configuration, running only 8 VCPUs, observes 8–15% higher IPC than the *No DMR 2X* configuration, since it has approximately half of the bandwidth and capacity pressure on the shared cache and network resources. *Reunion*, however, sees a 22–48% decrease in the IPC of each VCPU compared to *No DMR 2X*. The performance penalty of using Reunion is 34–53% compared to the 8 VCPU *No DMR* configuration. The reason for this overhead arises primarily from three sources: additional instruction window pressure, L2 cache-to-cache transfers, and serializing instructions. Each of these three is discussed in more detail below.

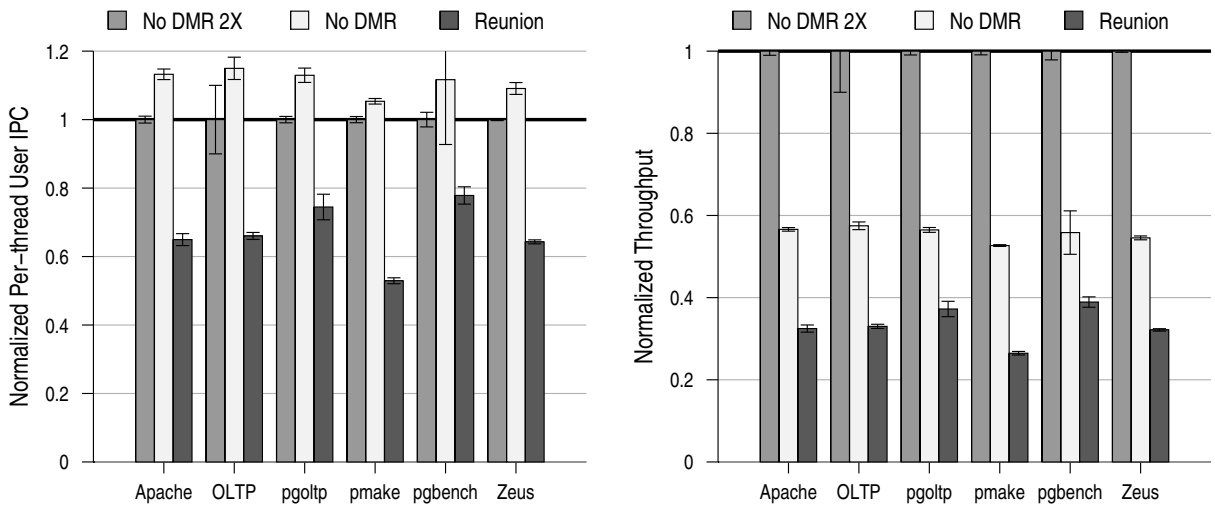
While this reduction in per-thread IPC is part of the picture, the impact on *throughput* created by the need to use twice as many core to run the same number of threads is even larger. Figure 5 shows this overall throughput impact, and the results are dramatic. As expected, throughput lost by *No DMR*, when not running VCPUs on all cores, is nearly half that of *No DMR 2X*. The throughput for *Reunion* is approximately one third to one quarter that of *No DMR 2X*, due not only to half as many VCPUs running, but also to the fact that each of those VCPUs slows down significantly.

Instruction Window Utilization The first overhead affecting DMR execution is capacity pressure on the instruction window and load/store queue (LSQ). In our experiments, Reunion observes full structures for approximately twice as many cycles as does the baseline. 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), which causes stores to wait in the instruction window until they are committed to the cache. The original Reunion proposal used TSO memory consistency [29], which allows the use of a store buffer, relieving some of this pressure.

Cache-to-Cache Transfers A second part of the overhead of Reunion results from increases in L2 cache-to-cache (C2C) transfers. C2C transfers increase by 20–50% for all benchmarks except `pmake` (`pmake` has very few C2C transfers in the baseline, and thus observes a 220% increase). Due to our use of an exclusive L3 cache (like the IBM Power5 [13] and AMD quad-core Opteron [11] processors), 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 3-hop transfers incur additional latency compared to a 2-hop L3 hit.

Serializing Instructions Finally, OS-intensive workloads typically encounter frequent *Serializing Instructions* (SIs) that cannot execute out of order [29, 36]. With Reunion the impact of these instructions is significant because 1) younger instructions must be committed before an SI executes, but the *Check* stage incurs additional delay before commit, and 2) the SI itself must be validated before younger instructions can enter the pipeline, incurring an additional comparison delay. When using Reunion, SIs stall fetch in our experiments for 15–46% of cycles. We serialize most of the SIs considered by Wells and Sohi [36], similar to the kinds of instructions serialized on a Pentium M (though for a very different ISA). Smolens, et al., consider most of the same instructions to be serializing [28], and also report a significant performance impact of SIs [29]. Also like Smolens, et al. [29], and the *Ideal SPARC* from Wells and Sohi [36], we simulate a hardware-filled TLB to avoid over-inflating the number of SIs.

Comparison to Prior Work The IPC impact we identify is in contrast to the published Reunion work [27, 29], which reports a single thread's IPC loss of 5–10%. The reason for this discrepancy arises from increased impact in each of the three sources identified above. The original Reunion proposal uses a 2-level (inclusive) cache hierarchy, which is not likely to incur additional C2C misses. They also use a larger



(a) Single Thread Performance

(b) Overall Throughput

Figure 5. DMR Performance Comparison

(256-entry) instruction window and a TSO memory consistency model, reducing the resource pressure of store latency, both on the instruction window and in the presence of SIs. In fact, Smolens reports that SC reduces the performance of Reunion by 30% on average [28], likely making this the largest contributor to the discrepancy. While both target configurations can potentially represent realistic systems, the reader should keep in mind that, even with a different configuration, per-thread IPC is only a small part of the motivation for mixed-mode operation. The original Reunion work did not examine throughput overheads.

5.2 Performance of a Mixed-Mode Multicore

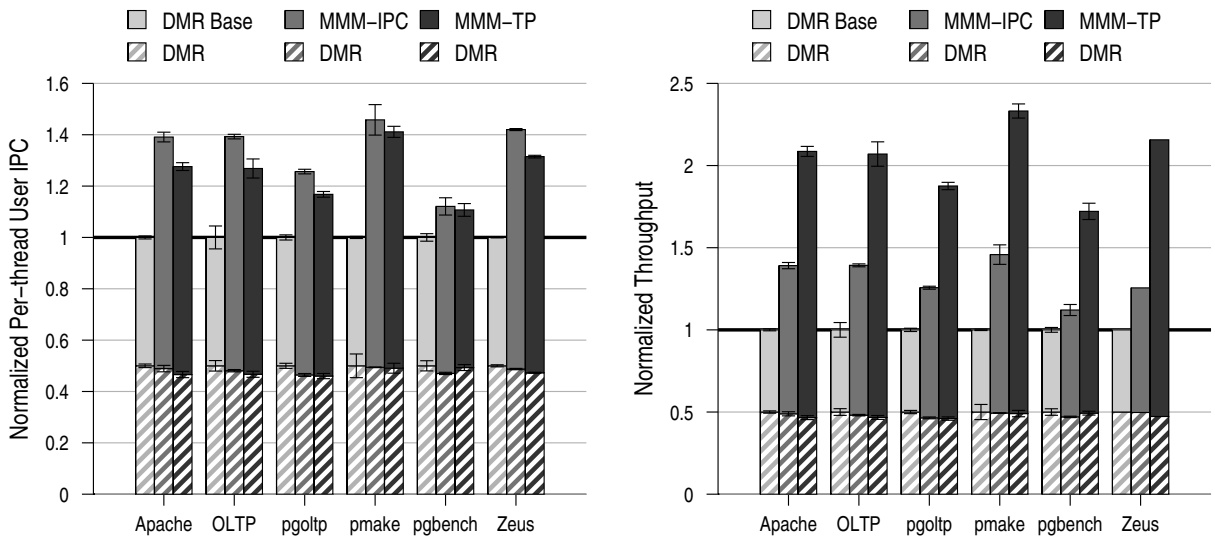
A mixed-mode consolidated server can provide differentiated service to different VMs, as described in Section 2. Figure 6(a) demonstrates the per-thread performance of mixed-mode operation. The striped bars at the bottom represent the normalized per-thread IPC of the guest VM that requires the high reliability of DMR. The solid, top bars represent the guest VM that does not require such high reliability.

In a traditional consolidated server, if one guest VM required reliability, then all guests would need to run with DMR to protect the integrity of the reliable VM. The left set of bars (labeled *DMR Base*) thus represents the baseline, where reliable, DMR mode is used for both VMs. The second set of bars, labeled *MMM-IPC*, represents the first MMM scheme where unused redundant cores are allowed to idle. Due to the IPC overhead of DMR execution, the high-performance guest VM observes 25–85% speedup over the full DMR configuration. The runtime of the reliable VM is virtually unchanged, though `pgoltp` observes a 6.5% slowdown due to the performance application more quickly displacing the reliable application's data in the shared L3 cache. Although we do not capture the effect on application's user-request latency, this per-thread IPC provides an indication of expected improvements.

The third set of bars, labeled *MMM-TP*, represents the second MMM system, which can better utilize all available cores to execute additional VCPUs. In this case, the per-thread IPC of those VCPUs still increases, though since more VCPUs are executing and consuming cache resources, the speedup of the high-performance VM is 24–67%, somewhat less than that of MMM-IPC.

Per-thread IPC is only part of the picture, however, since MMM-TP is using those otherwise-idle cores to execute more VCPUs. Figure 6(b) shows the overall system throughput, similarly normalized to the always-DMR baseline, and broken into throughput from each guest VM. The throughput of MMM-IPC is the same as the per-thread IPC speedup from Figure 6(a), since the same 8 VCPUs are executing in either mode. However, for scalable applications, such as these commercial workloads, improvements in throughput can be significant using MMM-TP, where the first VM now independently executes 16 VCPUs. This high-performance VM observes speedups of 2.4–3.6 due to the combined effect of per-VCPU IPC increase, and additional throughput from more VCPUs. Speedup of this VM over the static MMM configuration are 1.8–1.9. The throughput of the machine overall increases by 1.7-2.3X.

Effect of PAB Latency In previous results, we have assumed that the PAB was accessed in parallel with the L2 tags, causing no additional latency for any memory operations. We have also examined the impact of a 2-cycle PAB lookup in serial before accessing the L2 cache. Serialized accesses can possibly reduce the complexity of the cache controller. Since only store write-throughs are stalled by this serial lookup, the performance impact arises primarily through increased pressure on the instruction window



(a) Single Thread Performance

(b) Overall Throughput

Figure 6. Mixed-Mode Performance Comparison

and other structures. Detailed results (not shown for brevity) demonstrate that serial PAB lookups reduce the IPC of the application in performance mode by 3–10%. Such a small performance penalty is easily justified by the approximately 3X throughput gained by the ability to run in performance mode. The reliable application does not use the PAB, and therefore its performance does not change.

5.3 Overhead of Mode Switching

In addition to the PAB, mixed-mode operation can incur overhead during mode transitions. For the consolidated servers in the previous section, these transitions are infrequent, and their cost is easily amortized. However, with a trap-and-emulate software VMM, or if performing mixed-mode in a single-OS system, mode transitions need to occur much more frequently: Every time a guest VM attempts to perform a privileged operation, or a performance application enters or exits the operating system (e.g., for a system call), a mode switch occurs. To understand the expected overheads of mixed-mode operation in such systems, this section first examines the cost of entering and leaving dual-redundancy, and then examines the frequency that mode switching would be necessary. We show that the overheads are low enough that even frequent mode switches can be easily outweighed by the benefits of mixed-mode operation.

Switching Overhead Table 1 presents the average overhead (in cycles) for each VCPU to perform a mode switch. This data is taken from MMM-TP, which has higher average overhead than MMM-IPC because it must flush the L2 cache. As shown in the table, the overhead of the *Enter DMR* mode switch is approximately 2.2k cycles for all benchmarks. The overhead includes the cost of context switching *out* the state of the performance VCPU, switching *in* the state of the newly scheduled reliable VCPU, and synchronizing the vocal and mute cores. The overhead of *Leave DMR* includes the cost of synchronizing, context switching *out* the reliable VCPU, flushing the L2 cache, and context switching *in* the newly scheduled VCPU running a high-performance application. This overhead is much larger due to the cost of flushing the L2 cache, which takes approximately 8k cycles since we pessimistically assume that only one cache line can be flushed or written back to the shared L3 per cycle.

	Enter DMR	Leave DMR
Apache	2.4k	10.4k
OLTP	2.4k	10.3k
pgoltp	2.3k	10.2k
pmake	2.2k	9.9k
pgbench	2.3k	10.2k
Zeus	2.4k	10.3k

Table 1. Mixed-Mode Switching Overheads (cycles)

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

Table 2. Cycles Before Switching Modes for Single-OS

Switching Frequency The cost of the mode transitions in Table 1 is relatively small if these transitions occur infrequently, as is the expected case for a mixed-mode consolidated server using some hardware virtualization support (e.g., [31]), or para-virtualized guests (e.g., [3]). 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. Transitions may also be frequent when using a trap-and-emulate software VMM.

To examine the impact the switching latencies would have in a single-OS system, Table 2 presents the average number of cycles before switching from a user application to the OS, and from the OS back to the user application. This data is for each thread of the baseline, non-DMR system. All benchmarks except `Apache` and `Zeus` spend at least 200k cycles in user mode before entering the OS. Including the time spent in the OS itself (the sum of the two columns), 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 2 is approximately 13k cycles for all benchmarks. The implication of this data is that switching modes in a single-OS system would result in an 8% overhead for `Apache`, and less than a 5% overhead for the other benchmarks. For applications similar to SPEC CPU2000 that encounter a system call and subsequent mode switch less often, this overhead would be even less.

The bottom line, even in a single-OS mixed-mode system running applications with frequent OS activity, is that the IPC and throughput benefits of high-performance mode are expected to far outweigh the overhead of mode transitions.

6. Related Work

Many circuit- and microarchitectural-level techniques for tolerating various hardware faults have been proposed. Of primary interest to this work is a multitude of recent microarchitectural DMR proposals, which join together two cores to reliably execute one VCPU [12, 14, 19, 29, 33, 38]. Two of these in particular suggest that DMR can easily be turned on and off [19, 32], however, they do not investigate the issues involved in doing so. There is nothing inherent in any of these DMR proposals that is incompatible with the modifications for mixed-mode execution proposed in this paper. However, as this paper demonstrates, running some applications in DMR mode and some in high-performance mode is not as straightforward as it might first appear.

Walcott, et al., observe that the continuous use of redundant multithreading (RMT), within a single SMT core, can lead to significant IPC overheads [32]. They report overheads of 43% for one benchmark. To combat this overhead, they propose to toggle RMT on 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.

Overshadow is a software VMM-based memory encryption technique that can protect application data from a security-compromised OS [9]. 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 [1] 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. 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 [12, 14, 19, 29, 38]. DMR provides excellent coverage from a variety of sources, yet it comes with high performance overheads.

In this work, we build on the observation that some applications even today, and likely more in the future, require the high reliability of DMR. Yet for the foreseeable future, many applications will still not require such high hardware reliability, but will instead continue to require high performance. To address this diversity in needs, even among application simultaneously running on the same machine, we propose and design a *Mixed-Mode Multicore* (MMM). An MMM enables applications that need extra reliability to run in an extra-reliable mode, while applications that need high performance can avoid the high cost of that reliable mode.

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 reliable applications from any faults that may occur to a high-performance application. The second key challenge is that the need to protect the integrity of the system software, even when running a performance application, complicates the scheduling of software threads to cores.

After addressing these challenges, an MMM system is shown to improve the throughput of a high-performance application by 2.5–4 times compared to a system that always operates in reliable mode. An MMM can improve overall system throughput of a system with one reliable and one performance application 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 nearly *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, rather than switching all at once from running no applications in DMR mode to incurring significant performance loss for all applications.

Acknowledgments

We wish to thank Matthew Allen, Greg Wright, and Jared Smolens for helpful discussions and comments. This work is supported in part by National Science Foundation (NSF) grants CCF-0702313 and CNS-0551401, funds from the John P. Morgridge Chair in Computer Sciences and the University of Wisconsin Graduate School. Sohi has a significant financial interest in Sun Microsystems. The views expressed herein are not necessarily those of the NSF, Sun Microsystems or the University of Wisconsin.

References

[1] N. Aggarwal, P. Ranganathan, N. P. Jouppi, and J. E. Smith. Configurable isolation: building high availability systems with commodity multi-core processors. In *Proc. of 34th ISCA*, 2007.

[2] P. Barford and M. Crovella. Generating representative web workloads for network and server performance evaluation. In *1998 Conf. on Meas. & Model. of Comp. Sys.*, 1998.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proc. of 19th SOSP*, 2003.

[4] D. Bernick, B. Bruckert, P. D. Vigna, D. Garcia, R. Jardine, J. Klecka, and J. Smullen. Nonstop advanced architecture. In *Proc. of 2005 DSN*, 2005.

[5] S. Borkar. Microarchitecture and design challenges for gigascale integration: Keynote. In *Proc. of 37th MICRO*, 2004.

[6] S. Borkar, T. Karnik, J. Tschanz, A. Keshavarzi, and V. De. Parameter variations and impact on circuits and microarchitecture. In *Proc. of 40th DAC*, 2003.

[7] F. A. Bower, D. J. Sorin, and S. Ozev. A mechanism for online diagnosis of hard faults in microprocessors. In *Proc. of 38th MICRO*, 2005.

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

[9] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *Proc. of 13th ASPLOS*, 2008.

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

[11] P. Conway and B. Hughes. The AMD Opteron Northbridge architecture. *IEEE Micro*, 27(2):10–21, 2007.

[12] M. Gomaa, C. Scarbrough, T. N. Vijaykumar, and I. Pomeranz. Transient-fault recovery for chip multiprocessors. In *Proc. of 30th ISCA*, 2003.

[13] R. Kalla, B. Sinharoy, and J. M. Tendler. IBM Power5 chip: a dual-core multithreaded processor. *IEEE Micro*, 24(2):40–47, 2004.

[14] C. LaFrieda, E. Ípek, J. F. Martínez, and R. Manohar. Utilizing dynamically coupled cores to form a resilient chip multiprocessor. In *Proc. of 2007 DSN*, 2007.

[15] M.-L. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou. Understanding the propagation of hard errors to software and implications for resilient system design. In *Proc. of 13th ASPLOS*, 2008.

[16] P. Magnusson et al. Simics: A full system simulation platform. *IEEE Comp.*, 35(2):50–58, Feb 2002.

[17] M. R. Marty and M. D. Hill. Virtual hierarchies to support server consolidation. In *Proc. of 34th ISCA*, 2007.

[18] D. McEvoy. The architecture of tandem's nonstop system. In *Proc. of ACM 1981 Conf.*, 1981.

[19] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. In *Proc. of 29th ISCA*, 2002.

[20] Open Source Development Labs. Database test suite. Viewed 7/28/2008.

[21] PostgreSQL Global Development Group. PostgreSQL. Viewed 7/28/2008.

[22] Semiconductor Industry Association. International technology roadmap for semiconductors: Executive summary, 2005.

[23] J. W. Sheaffer, D. P. Luebke, and K. Skadron. The visual vulnerability spectrum: characterizing architectural vulnerability for graphics hardware. In *Proc. of 21st Eurographics symposium on Graphics hardware*, 2006.

[24] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Proc. of 2002 DSN*, 2002.

[25] S. Shyam, K. Constantinides, S. Phadke, V. Bertacco, and T. Austin. Ultra low-cost defect protection for microprocessor pipelines. In *Proc. of 12th ASPLOS*, 2006.

[26] T. J. Slegel et al. IBM's S/390 G5 microprocessor design. *IEEE Micro*, 19(2):12–23, 1999.

[27] J. C. Smolens. *Fingerprinting: Hash-Based Error Detection in Microprocessors*. PhD thesis, Carnegie Mellon University, 2008.

[28] J. C. Smolens. Personal communication, Dec 2008.

[29] J. C. Smolens, B. T. Gold, B. Falsafi, and J. C. Hoe. Reunion: Complexity-effective multicore redundancy. In *Proc. of 39th MICRO*, 2006.

[30] J. C. Smolens, B. T. Gold, J. C. Hoe, B. Falsafi, and K. Mai. Detecting emerging wearout faults. In *Proc. of Workshop on SELSE*, 2007.

[31] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. M. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith. Intel virtualization technology. *IEEE Comp.*, 38(5), 2005.

[32] K. R. Walcott, G. Humphreys, and S. Gurumurthi. Dynamic prediction of architectural vulnerability from microarchitectural state. In *Proc. of 34th ISCA*, 2007.

[33] C. Weaver and T. M. Austin. A fault tolerant approach to microprocessor design. In *Proc. of 2001 DSN*, 2001.

[34] P. M. Wells, K. Chakraborty, and G. S. Sohi. Hardware support for spin management in overcommitted virtual machines. In *Proc. of 15th PACT*, 2006.

[35] P. M. Wells, K. Chakraborty, and G. S. Sohi. Adapting to intermittent faults in multicore systems. In *Proc. of 13th ASPLOS*, 2008.

[36] P. M. Wells and G. S. Sohi. Serializing instructions in system-intensive workloads: Amdahl's law strikes again. In *Proc. of 14th HPCA*, 2008.

[37] T. F. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi. Temporal streaming of shared memory. In *Proc. of 32nd ISCA*, 2005.

[38] H. Zhou. A case for fault tolerance and performance enhancement using chip multi-processors. *Comp. Arch. Letters*, 5(1):6, 2006.