Cost-effective Designs for Supporting Correct Execution and Scalable Performance in Many-core Processors

by

Bogdan Florin Romanescu

Department of Electrical and Computer Engineering
Duke University

Date: ______________________

Approved:

____
Daniel J. Sorin, Advisor

____
Alvin R. Lebeck

____
Christopher Dwyer

____
Romit Roy Choudhury

____
Landon Cox

Dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Electrical and Computer Engineering in the Graduate School of Duke University
2010

Abstract

(Computer engineering)

Cost-effective Designs for Supporting Correct Execution and Scalable Performance in Many-core Processors

by

Bogdan Florin Romanescu

Department of Electrical and Computer Engineering
Duke University

Date: ______________________

Approved:

____
Daniel J. Sorin, Advisor

____
Alvin R. Lebeck

____
Christopher Dwyer

____
Romit Roy Choudhury

____
Landon Cox

An abstract of a dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Electrical and Computer Engineering
in the Graduate School of Duke University
2010

Abstract

Many-core processors offer new levels of on-chip performance by capitalizing on the increasing rate of device integration. Harnessing the full performance potential of these processors requires that hardware designers not only exploit the advantages, but also consider the problems introduced by the new architectures. Such challenges arise from both the processor's increased structural complexity and the reliability issues of the silicon substrate. In this thesis, we address these challenges in a framework that targets correct execution and performance on three coordinates: 1) tolerating permanent faults, 2) facilitating static and dynamic verification through precise specifications, and 3) designing scalable coherence protocols.

First, we propose CCA, a new design paradigm for increasing the processor's lifetime performance in the presence of permanent faults in cores. CCA chips rely on a reconfiguration mechanism that allows cores to replace faulty components with fault-free structures borrowed from neighboring cores. In contrast with existing solutions for handling hard faults that simply shut down cores, CCA aims to maximize the utilization of defect-free resources and increase the availability of on-chip cores. We implement three-core and four-core CCA chips and demonstrate that they offer a cumulative lifetime performance improvement of up to 65% for industry-representative utilization periods. In addition, we show that CCA benefits systems that employ modular redundancy to guarantee correct execution by increasing their availability.

Second, we target the correctness of the address translation system. Current

processors often exhibit design bugs in their translation systems, and we believe one cause for these faults is a lack of precise specifications describing the interactions between address translation and the rest of the memory system, especially memory consistency. We address this aspect by introducing a framework for specifying translation-aware consistency models. As part of this framework, we identify the critical role played by address translation in supporting correct memory consistency implementations. Consequently, we propose a set of invariants that characterizes address translation. Based on these invariants, we develop DVAT, a dynamic verification mechanism for address translation. We demonstrate that DVAT is efficient in detecting translation-related faults, including several that mimic design bugs reported in processor errata. By checking the correctness of the address translation system, DVAT supports dynamic verification of translation-aware memory consistency.

Finally, we address the scalability of translation coherence protocols. Current software-based solutions for maintaining translation coherence adversely impact performance and do not scale. We propose UNITD, a hardware coherence protocol that supports scalable performance and architectural decoupling. UNITD integrates translation coherence within the regular cache coherence protocol, such that TLBs participate in the cache coherence protocol similar to instruction or data caches. We evaluate snooping and directory UNITD coherence protocols on processors with up to 16 cores and demonstrate that UNITD reduces the performance penalty of translation coherence to almost zero.

To my grandparents

Bunicilor mei

Contents

List of Tables

List of Figures

List of Abbreviations

AT	Address translation
CC	Cannibalizable core
CS	Core shutdown
DMR	Dual modular redundancy
MRF	Map/remap function
NC	Normal core
PTE	Page table entry
TLB	Translation lookaside buffer
TMR	Triple modular redundancy
SC	Sequential consistency

Acknowledgements

First and foremost, I want to thank my parents for their support throughout my graduate studies.

My advisor, Prof. Daniel Sorin, has been a continuous source of motivation and mentoring. I learned from Dan the art of abstracting concepts, analyzing problems rigorously, and meaningful communication. I thank Dan for his patience and guidance in my development as a researcher. I am grateful to Prof. Alvy Lebeck for the decision to join our research, as his vast experience on architecture and systems proved invaluable.

I benefited from being part of a great computer architecture group at Duke. The reading group discussions helped me become a better critic and a sharper thinker. I was also fortunate to have two fantastic mentors during my summer internships, Jaidev Patwardhan and Anne Bracy. Both Jaidev and Anne showed me the importance of being a good mananger in addition to being a skillful engineer.

My student life would have certainly been duller if it weren't for my colleagues and friends. In particular, Vincent Mao has been a great office mate and I thank him for all the time spent discussing not just research. I am also grateful to Ionut Constandache for sharing memories and thoughts.

Finally, I am forever in debt to Prof. Călin Caşcaval from TU Iaşi for introducing me to research and supporting me in pursuing my graduate studies.

1

Introduction

Architects look ahead to many-core designs as the next standard of cost-effective performance [53]. Leveraging the still increasing rate of on-die transistor integration, many-core processors are expected to feature hundreds to thousands of cores [24]. This order of magnitude increase in core count over existing processors offers tremendous performance opportunities, but also introduces new challenges for hardware designers [15]. Consequently, architects must address issues such as scalability, power-efficiency, and unreliability of the device substrate.

This thesis proposes architectural solutions for some of these problems that affect a processor's correct execution and performance. In particular, we focus on dependability and scalability issues. Dependability encompasses a vast area of topics, including reliability, maintanability, and security. We restrict our dependability approach to two aspects, availability and error detection. Thus, we address the challenges of many-core processors on three directions: 1) availability in the presence of permanent faults, 2) supporting error detection through precise specifications, and 3) designing scalable coherence protocols.

Availability characterizes a system's capacity to function properly at a specific time, and is a function of the resources the system can provide to support correct execution. Availability is a primary concern for many-core processors given the increased impact of permanent hardware faults (*i.e.*, hard faults) and manufacturing defects for deep-submicron technologies [25]. Considering the increased density of on-chip transistor integration, these types of faults are expected to impact multiple processor resources. Designers must assume that such faults will occur during the processor's lifetime, and propose architectural solutions to maximize the available on-chip resources. In Section 1.1, we describe a case for increasing processor availability by tolerating hard faults in cores. We propose handling such faults through a reconfiguration mechanism that aggregates functional units from neighboring faulty cores. Our solution provides sustained availability and increases the processor's expected lifetime performance.

A fundamental prerequisite for our availability solution is the system's ability to detect incorrect execution in any of the processor's components. Incorrect execution can be caused by either hardware faults, or design faults which are introduced during the design process. Several efficient solutions exist for detecting faults in cores and parts of the memory system [16, 86, 89]. However, in Section 1.2, we identify address translation as one system for which no error detection solutions are currently available. One possible cause for this lack of error detection mechanisms is the absence of precise specifications of how the address translation system interacts with the rest of the memory system, and especially memory consistency. We address this lack of specifications by proposing a framework for specifying translation-aware consistency models. The critical role played by address translation in supporting memory consistency motivates us to propose a set of invariants that characterizes the address translation system. Based on these invariants, we develop a dynamic verification solution for address translation which facilitates the runtime verification of memory consistency.

The last part of the thesis addresses the issue of scalable performance, arguably one of the most critical aspects of many-core processors design. Integrating hundreds of cores on the same die requires scalable interconnects and inter-core communication mechanisms such as coherence protocols [15]. Although architects have proposed scalable solutions with respect to these components [96, 50, 8, 84], we identify translation coherence as one area that has been generally neglected. Software-based solutions for maintaining translation coherence are performance costly and non-scalable, and no alternatives are currently available. Section 1.3 argues that the time has come to move translation coherence into hardware. We propose one such solution by integrating translation coherence into the regular cache coherence protocol. We implement our solution on systems with both snooping and directory cache coherence protocols, and demonstrate that it reduces the performance penalty associated with translation coherence to almost zero.

Next, we discuss in detail the motivation for the three research directions of this thesis.

1.1 Processor Availability in the Presence of Hard Faults

Deep-submicron technologies are characterized by an increased likelihood of hard faults [42, 120]. Smaller transistors and wires are more susceptible to permanent faults. For pre-90nm technologies, the degradation caused by such faults was small enough to be accounted for in the component's testing margin such that it would not affect the device functionality [25]. However, Srinivasan *et al.* [120] demonstrated that there is a sharp decrease in reliability beyond 90nm due to physical wearout induced by time-dependent dielectric breakdown, electromigration, and stress migration. Furthermore, as we continue to add more transistors and wires, there are more opportunities for hard faults to occur either during fabrication or in the field [25].

Although current chips already incorporate mechanisms for addressing hard faults, most of them target SRAM structures. This is a consequence of the memory cells being more prone to faults than regular logic for pre-90nm technologies [52]. Such solutions for tolerating hard faults in memory structures include error correcting codes and provisioning spare rows/columns [77, 26]. The spare components can be used to replace or remap few memory blocks transparently to the software such that processor's performance is virtually unaffected.

In contrast, processors have few, if any, solutions for tolerating hard faults in cores. The most common method of handling such faults is to disable either the affected component or the entire core. The former requires however that the faulty component can be precisely identified, and that the core contains replicas of the unit. The latter condition is difficult to satisfy even by superscalar cores as few structures are replicated within the core [97]. Consequently, chip designers prefer disabling the entire core, a technique that is prevalently used by industry to increase the chip's manufacturing yield. For example, IBM markets Cell processors for Sony Playstations with just 7 out of 8 functional SPEs [80].

The main drawback of disabling cores is that it reduces the availability of on-chip resources, leading to decreased overall processor performance. Thus, highly-available systems rely instead on spare cores for delivering performance in the presence of hard faults [17]. Unfortunately, spare components (either cold or hot) [10, 117] consume hardware resources that provide no performance benefit during fault-free operation. If we provision spares for all components, then we achieve approximately half the fault-free performance of an equal-area chip without spares. The sparing cost increases for systems that must tolerate multiple hard faults such as triple modular redundant (TMR) systems [68].

In this thesis, we address the inefficiencies of current solutions in providing cost-effective availability in the presence of hard faults in cores by proposing the Core Cannibalization Architecture (CCA). The CCA concept builds on the observation that despite multiple hard faults in cores, a chip provides enough fault-free resources that can be aggregated to yield functional cores. In Chapter 2, we propose and evaluate various CCA designs that reuse components at the granularity of pipeline stages. We demonstrate that CCA significantly improves lifetime chip performance compared to processors that rely on disabling cores. In addition, CCA can be combined with solutions using redundant cores for increased processor availability.

1.2 Checking Correctness of Address Translation and Translation-Aware Memory Consistency

In addition to permanent faults, many-core processors face dependability concerns due to transient faults and design faults [42, 25]. Similar to permanent faults, transients are a consequence of the smaller transistor sizes which render chips more susceptible to faults caused by neutrons and alpha particles [42]. In contrast, design faults represent human errors, and are "facilitated" by increased design complexities, reduced testing time and imperfect coverage of random testing [66]. Despite different causes, both types of faults have the same effect on a circuit, resulting in incorrect behavior.

One of the systems that is currently vulnerable to these faults is address translation (AT). Representative of AT's vulnerability is the disproportionate fraction of published bugs in shipped processors [2, 3, 4, 59, 61, 62, 63] that involve AT hardware, including the infamous TLB coherence bug in AMD's quad-core Barcelona processor [131]. Table 1.1 lists a few examples of these bugs.

We believe that one of the underlying causes for AT's reliability problems is the designers' tendency to over-simplify memory consistency and to neglect AT's impact on consistency models. Current specifications do not provide a precise description of the interactions between AT and the rest of the memory system. Such clear specifications of correctness are a fundamental prerequisite for detecting incorrect

Table 1.1: Examples of Published Address Translation Design Bugs.

Processor	Design Bug	Effect
AMD Athlon64/ Opteron [2]	TLB flush filter may cause coherency problem in multicore systems	Unpredictable system failure (possible use of stale translations)
AMD Athlon64/ Opteron [2]	INVLPG instruction with address prefix does not correctly invalidate the translation requested	Unpredictable system behavior (use of stale translation)
Intel Core Duo [62]	One core is updating a page table entry while the other core is using the same translation entry may lead to unexpected behavior	Unexpected processor behavior
Intel Core Duo [62]	Updating a PTE by changing R/W, U/S or P bits without TLB shootdown may cause unexpected processor behavior	Unexpected processor behavior

behavior.

In Chapter 3, we propose a framework for precise, implementation-independent specification of AT-aware memory consistency. We discuss in depth the memory consistency levels that closely interact with the AT system. We identify one particular level that requires AT support and analyze the AT aspects that affect the consistency specifications at this level.

Our framework benefits both hardware designers and programmers. Precisely specifying the interactions between AT and the memory system reduces the probability of designers introducing design faults at this interface. Second of all, our specifications help system programmers in writing software that involves AT by clearly stating the requirements for correct execution. Finally, the proposed framework facilitates static verification and allows architects to develop checkers for runtime verification of address translation.

The important role that AT plays in supporting some levels of memory consistency implies that a correct AT system is required for correct memory consistency implementations. To facilitate checking AT correctness, we propose a framework for AT specifications (Chapter 4). Based on this framework, we create DVAT, an efficient dynamic verification scheme for AT coherence that can detect errors due to design bugs and runtime faults. We demonstrate that DVAT detects design bugs similar to the ones reported in processor errata, and supports comprehensive dynamic verification of memory consistency.

1.3 Scalable Translation Coherence Protocol Design

Our analysis of the AT system reveals that maintaining translation coherence has a significant performance cost even for systems with few cores. Translation caches are just one of multiple types of caches that shared memory processors or multiprocessor systems must maintain coherent, including instruction and data caches. While instruction and data cache coherence has been the focus of extensive research on scalable coherence protocols [96, 50, 8, 1, 84, 9], few solutions have been proposed for scalable translation coherence [125]. Designing a scalable protocol for translation coherence requires us to first understand what essentially differentiates translation coherence from instruction/data coherence.

For caches that hold instructions or data, coherence is almost generally maintained with an all-hardware cache coherence protocol. Hardware controllers at the caches coordinate amongst themselves using snooping or directories to ensure that instructions and data are kept coherent, and this coherence is not software-visible. However, for caches that hold address translations (*i.e.*, TLBs), coherence is almost always maintained by an OS-managed software coherence protocol. Even for architectures with hardware control of TLB fills and evictions, when an event occurs that affects the coherence of TLB entries (*e.g.*, eviction of a page of virtual memory), the OS ensures translation coherence through a software routine called TLB shootdown [19].

Performing cache coherence in hardware provides two major advantages: performance and microarchitectural decoupling. Performance-wise, hardware is much faster than software. For coherence, this performance advantage grows as a function of the number of caches. Although using software for local activities (*e.g.*, TLB fills and replacements) might have acceptable performance, even some architectures that have traditionally relied on software for such operations (*e.g.*, SPARC) are now transitioning to hardware support for increased performance [95]. In contrast, activities with global coordination are painfully slow when performed in software. For example, Laudon [75] mentions that for a page migration on the SGI Origin multiprocessor, the software routine for TLB shootdown is three times more time-consuming than the actual page move. The second reason for performing cache coherence in hardware is to create a high-level architecture that can support a variety of microarchitectures. A less hardware-constrained OS can easily accommodate heterogeneous cores as it does not have to be aware of each core's particularities [71]. Furthermore, hardware coherence enables migrating execution state between cores for performance, thermal, or reliability purposes [34, 51] without software knowledge.

Given that hardware seems to be an appropriate choice for cache coherence, why has TLB coherence remained architecturally visible and under the control of software? We believe that one reason architects have not explored hardware TLB coherence is that they already have a well-established mechanism that is not too costly for systems with a small number of processors. For previous multiprocessor systems, Black [19] explains that the low overhead of maintaining TLB coherence in software on current machines may not justify a complete hardware implementation. As we show in the Section 5.1.2, this conclusion is likely to change for future many-core chips.

This motivates us to consider a hardware approach for translation coherence. A hardware TLB coherence protocol provides three primary benefits. First, it drastically reduces the performance impact of TLB coherence. While this performance benefit is worthwhile on its own, it also lowers the threshold for adopting features that incur a significant amount of TLB coherence activity, including: hardware transactional memory (*e.g.*, XTM [40]), user-level memory management for debugging [43], and concurrent garbage collection [39]. Second, hardware TLB coherence provides a cleaner interface between the architecture and the OS, which could help to reduce the likelihood of bugs at this interface, such as the recent TLB coherence bug in the AMD Barcelona chip [131]. Third, by decoupling translation coherence from the OS, hardware TLB coherence can be used to support designs that use TLBs in non-processor components such as network cards or processing elements [82, 102]. This might facilitate a globally-shared address space among all components of a computing system.

Considering these advantages, in Chapter 5 we propose UNITD, a hardware coherence protocol that integrates translation coherence within the regular cache coherence protocol. UNITD "snoops" TLBs on regular coherence requests, such that any change to the page tables automatically triggers TLB coherence. Relying on small additional hardware, UNITD successfully eliminates the performance cost associated with the TLB shootdowns routines. In addition, UNITD does not affect the complexity or performance of the regular cache coherence protocol.

1.4 Thesis Statement and Contributions

The imminent adoption of many-core processors as the next computing standard will make these designs ubiquitous in our daily lives. Such processors will have to support a wide variety of applications, ranging from systems that require correct execution above all, to applications that demand performance. This observation motivates the following thesis statement:

The characteristics of many-core processors enable the design of cost-effective solutions for supporting correct execution and performance, given the reliability and *scalability challenges of these processors.*

To support this statement, this thesis makes the following contributions in the context of many-core processors:

- **Proposes a solution to improve processor's lifetime performance in the presence of hard faults.** The dissertation introduces a low-cost and efficient self-repair mechanism for many-core processors with simple cores by enabling sharing of resources. The reconfiguration solution provides sustained performance and availability, that outweigh the slight performance overhead in fault-free scenarios over the processor's lifetime.

- **Develops a framework for specifying address translation-aware memory consistency models.** The framework analyzes the consistency levels that closely interact with the address translation system, and identifies the translation-related aspects that impact consistency models. Providing a thorough, multi-level specification of consistency enables programmers, designers, and design verifiers to more easily reason about the memory system's correctness.

- **Proposes a dynamic verification scheme for address translation.** We support the dynamic verification solution with an implementation-independent framework for specifying address translation. In addition to checking the correctness of the address translation system, the proposed mechanism facilitates comprehensive verification of memory consistency.

- **Introduces a hardware coherence protocol for translation coherence.** The proposed protocol integrates translation coherence into the existing cache coherence protocol, with TLBs participating in the protocol like instruction or data caches. Our hardware coherence protocol provides scalable performance compared to existing software-based solutions for maintaining translation coherence.

1.5 Thesis Structure

Chapter 2 describes CCA, our solution for improving the lifetime performance of many-core processors in the presence of hard faults. Chapter 3 introduces the framework for specifying translation-aware consistency models, and analyzes the impact of address translation on virtual address memory consistency. Chapter 4 proposes a framework for specifying address translation and details DVAT, a dynamic verification mechanism for checking the correctness of the address translation system. Chapter 5 describes UNITD coherence, a unified hardware coherence framework that integrates instruction, data and translation coherence in the same coherence protocol. Finally, Chapter 6 summarizes the thesis' contributions.

2

Improving Lifetime Performance of Many-core Processors in the Presence of Hard Faults

Technology trends are leading to an increasing likelihood of hard (permanent) faults in processors [120]. Traditional approaches to this problem include provisioning spare components or simply disabling cores. Unfortunately, spare components (either cold or hot) consume hardware resources that provide no performance benefit during fault-free operation. If we provision spares for all components, then we achieve approximately half the fault-free performance of an equal-area chip without spares. In turn, core shutdown (CS) disables an entire core if any of its components has a hard fault and thus wastes much fault-free circuitry.

Motivated by the deficiencies of existing solutions, our goal is to tolerate hard faults in many-core processors without sacrificing hardware for dedicated spare components. There are two aspects to many-core processors that distinguish the issue of self-repair from the case for single-core processors. First, power and thermal constraints motivate the use of simple, in-order cores, perhaps in conjunction with one or two superscalar cores. Examples of chips with simple, narrow cores include the UltraSPARC T1 [70] and T2 [112], Cray MTA [31], empowerTel MXP processor [54], Renesas SH-2A-Dual [122], and Cisco Silicon Packet Processor [41], and we expect this trend to continue for many-core processors. Unfortunately, simple cores have little intra-core redundancy of the kind that has been leveraged by superscalar cores to provide self-repair [27, 113, 119]. Just one hard fault in the lone ALU or instruction decoder renders a simple core useless, even if the entire rest of the core is fault-free. The second aspect of self-repair that is distinct to many-core processors is the opportunity to use resources from fault-free cores.

We propose the Core Cannibalization Architecture (CCA), the first design of a low-cost and efficient self-repair mechanism for many-core processors with simple cores. The key idea is that one or more cores can be cannibalized for spare parts, where parts are considered to be pipeline stages. The ability to use stages from other cores introduces some slight performance overhead, but this overhead is outweighed by the improvement in lifetime chip performance in the presence of multiple hard faults. Furthermore, CCA provides an even larger benefit for many-core chips that use cores in a triple modular redundancy (TMR) or dual modular redundancy (DMR) configuration, such as Aggarwal *et al.*'s approach [10]. CCA enables more cores to be operational, which is crucial for supporting TMR or DMR.

We develop several concrete implementations of CCA in the context of processors that consist of up to four simple OpenRISC 1200 cores [74]. We also present a straightforward extension of these designs to many-core processors. We show that CCA achieves better performance than CS over the chip's lifetime. After only 2 years, CCA chips outperform CS chips. Over a lifetime of 12 years, CCA achieves a 63% improvement in cumulative performance for 3-core chips and a 64% improvement for 4-core chips. Furthermore, if cores are used redundantly (*e.g.*, TMR or DMR), then CCA's improvement is 70% for 3-core chips and 63% for 4-core chips.

In this chapter, after describing the baseline system model (Section 2.1), we detail the CCA concept (Section 2.2) and discuss design-related aspects (Section 2.3). We describe our CCA implementations in Section 2.4. We then evaluate CCA (Section 2.5) and compare it to prior research (Section 2.6). Finally, we draw conclusions in Section 2.7.

2.1 Baseline System Model

In this section, we present our core model and discuss core shutdown, the natural design point against which we compare.

2.1.1 Core Model

In our analysis we focus on simple, in-order cores with little redundancy. We present CCA in the context of 1-wide (scalar) cores, but CCA also applies to many cores that are wider but still have numerous single points of failure. There are many *k*-wide cores that cannot tolerate a fault by treating the core as being *k-1*-wide. For example, the Renesas SH-2A [122] is dual-issue, but it has only one shifter and one load/store unit. Any fault in either of those units renders the entire core unusable. Other simple cores are susceptible to numerous single faults (*e.g.*, in the PC update logic) that affect all lanes of the processor. Many commercial cores fit our core model [70, 112, 31, 41]. In addition, Powell *et al.* [97] show that non-redundant structures represent the vast majority of core area even for superscalar cores.

Our model assumes that the core has mechanisms for detecting errors and diagnosing hard faults (*i.e.*, identifying the locations of hard faults). Detection and diagnosis are orthogonal issues to self-repair, and acceptable schemes already exist, such as the built-in self-test (BIST) used by the BulletProof pipeline [114]. CCA may require additional BIST test vectors than a baseline system to distinguish faults that are in different pipeline stages and that would otherwise be exercised by the same test vector. CCA can also rely on software-based diagnosis solutions such as the ones proposed by Hari *et al.* [110], which eliminate the need for additional test

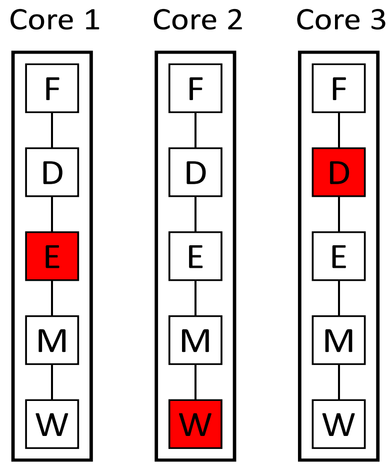


Figure 2.1: 3-core CS Chip. Generic cores have five pipe stages: Fetch, Decode, Execute, Memory, and Writeback. Each core has one fault (Core 1 in the Execute stage, Core 2 in Writeback and Core 3 in Decode), rendering the chip useless.

vectors.

2.1.2 Core Shutdown Design

As mentioned in the chapter's introduction, a multicore processor with C simple cores can tolerate hard faults in F (F<C) distinct cores by simply not using the faulty cores. A single fault in a core renders the entire core useless. Additional faults in the same core (*e.g.*, multiple faults can occur during the manufacturing process) do not matter, since the core has already been shut off. The performance of a chip with CS is proportional to the number of fault-free cores, C-F. Figure 2.1 illustrates a 3-core processor with core shutdown. In the presence of three hard faults, one in each core, the processor achieves zero performance because none of its cores are operable.

2.2 CCA Concept

The CCA concept is based on the tight integration of the neighboring cores in a many-core processor. The key idea is that cores can be cannibalized for spare parts by on-die adjacent cores to replace their own defective components, and thus become

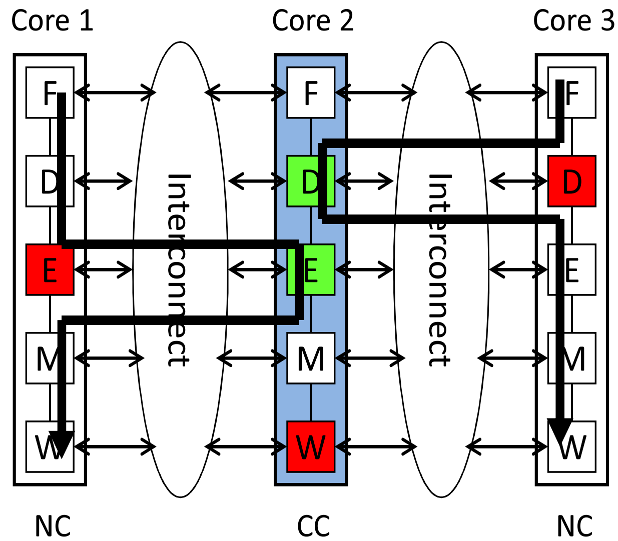


Figure 2.2: 3-core CCA Chip. Cores 1 and 3 are NC, Core 2 is a CC. The 2 NCs are functional leading to a non-zero chip performance.

fault-free. Thus, a CCA system consists of a number of *normal cores* (*NCs*) that cannot be cannibalized as well as some number of *cannibalizable cores* (*CCs*). We use the notation CCA$X(Y/Z)$ to refer to a CCA chip with a total of X cores, out of which Y are NCs and Z are CCs, where $X\!=\!Y\!+\!Z$. Similarly, we use the notation CSX to refer to a CS chip with X cores.

At a high level, a CCA processor resembles the system in Figure 2.2. The figure illustrates a CCA system with three cores, where Core 2 is a CC. CCA enables Core 1 to overcome a faulty Execute stage and Core 3 to overcome a faulty Decode stage, by cannibalizing these stages from Core 2. The cannibalization process is facilitated by a dedicated interconnect. The result is that, despite the presence of three hard faults (including the fault in Core 2's Writeback stage), Core 1 and Core 3 continue to function correctly.

The performance of both cores is somewhat degraded, though, because of the delay in routing to and from the cannibalized stages. However, comparing the chips in Figures 2.1 and 2.2, which both have three faults, we see that CS offers zero performance, yet CCA provides the performance of two slightly degraded cores.

In general, as the number of faults increases, CCA outperforms CS. For chips with zero or very few faults that do not allow CCA-type reconfigurations, a processor with CS outperforms CCA because CCA's reconfigurability logic introduces some performance overhead into the cores. This performance overhead is similar to that incurred by schemes that provide spare components. However, as the number of faults increases, CCA can tolerate more of them and provide a graceful performance degradation. We demonstrate in Section 2.5 that over the chip's lifetime, the expected performance of CCA chips exceeds the expected performance of CS chips.

2.3 CCA Design Decisions

There are three important issues involved in a CCA design: the granularity at which to cannibalize cores, the sharing policy between CCs and NCs, and the assignment of the chip's cores to be either an NC or a CC. After analyzing the first two issues, spare granularity and sharing policy, we make fixed decisions for both of them. For the third issue, chip layout, we explore several options.

Spare Granularity. We cannibalize cores at the granularity of pipeline stages. The coarsest possible granularity is spare cores (*i.e.*, CS), but coarse granularity implies that a single fault in a core renders the entire core useless. Finer granularity avoids wasting as much fault-free hardware, but it complicates the design, especially the routing to and from spare components. For example, one recent scheme for fine-grain redundancy [93] has an area overhead that is greater than 2x. We choose a granularity of pipeline stages because it offers a good balance between complexity and performance. Our choice is confirmed by Gupta *et al.* [48] that, in a concept similar to CCA, determined that providing spares at pipeline stages granularity offers the most cost-effective performance.

Sharing Policy. Another issue to resolve is whether to allow multiple cores to simultaneously share a given component (*i.e.*, pipeline stage for our implementation).

There are three options. First, at one extreme, a core with a faulty component of type Z "borrows" (time multiplexes) a component of type Z from a neighboring core that continues to function (*i.e.*, is not cannibalized). A second option is to allow multiple cores to time multiplex a single cannibalized component. Both of these first two options introduce resource contention, require arbitration logic, and complicate pipeline control logic. For these reasons, we choose a third option, in which any given component can only be used by a single core.

Chip Layout. Categorizing the chip's cores into CCs and NCs is crucial for the increased performance of the CCA chip. There are two aspects that influence CCA's performance given a fixed core count. The first is the number of cores that are CCs. Underprovisioning CCs leaves NCs without spare components, while overprovisioning CCs can lead to wasteful allocation of resources, as the interconnection required for providing access to CCs increases in complexity and size. The second aspect is the arrangement of NCs and CCs such that we minimize the distance between NC stages and potential CC spare stages. We must carefully balance the two aspects in order to provide the best area-performance tradeoff. Consequently, we implement several CCA designs based on different CCs-NCs configurations and compare them in terms of performance and cost.

2.4 CCA Implementations

In this section, we first describe the cores used in our CS and CCA chips (Section 2.4.1). We then describe two concrete CCA implementations, with three cores (Section 2.4.2) and four cores (Section 2.4.3), respectively. Based on these designs, we discuss how to extend CCA to chips with greater numbers of cores (Section 2.4.4).

A fundamental aspect in any CCA implementation is the latency of the interconnect required for cannibalizing components. The characteristics of this interconnect are a function of low-level issues such as chip layout and wire delay. Therefore, a proper evaluation of CCA requires us to implement the designs at a low level detail. We construct Verilog models for all designs we evaluate, including systems with and without CCA. To evaluate area and delays, we floorplan and layout chips using Synopsys Design Compiler [123] and Cadence Silicon Ensemble [28]. We use a proprietary TSMC 90nm standard cell library for the synthesis flow. Unfortunately, the library does not include memory cells, and using regular flip-flops in synthesis creates unrealistically large RAM structures and diminishes the impact of our changes. In order to provide a fair evaluation, we estimate the size of the memory structures using CACTI [92].

2.4.1 Baseline CS and CCA Cores

The core of the baseline CS processor is the OpenRISC 1200 (OR1200) [74]. The OR1200 core is a scalar, in-order, 32-bit core with 4 pipeline stages: Fetch, Decode, Execute, and Writeback. Each core has 32 registers and separate instruction and data L1 caches (I-cache and D-cache). Implemented in our 90nm technology, we can clock the core at a maximum frequency of roughly 400MHz.

The analysis of CCA cores is impacted by the implications of stage borrowing. An NC's use of a cannibalized CC's stage introduces issues that are specific to that particular stage, so we discuss next the cannibalization of each stage.

Fetch. The Fetch stage involves I-cache accesses. If an NC uses a CC's Fetch stage, it also uses the CC's I-cache instead of its own cache.

Decode. The Decode stage is responsible for instruction decoding, accessing the register file and determining the destination address for jump/branch instructions. A particularity of this stage is the branch destination (BD) block. The OR1200 core has a one-instruction delay slot for branches and jumps, and the BD block is responsible for computing the address during the delay slot and communicating the destination to the Fetch stage. This block is tightly coupled with the Fetch stage while operating independently from the rest of the decode logic. Therefore, due to this tight coupling, we consider the BD block as part of the Fetch stage. An NC that reuses the Fetch stage of a CC also reuses the CC's BD block. In addition to the BD block, the Decode stage includes the register file such that an NC that uses a CC's Decode stage also uses that CC's register file. In this case, the NC must route back to the CC's register file during Writeback.

Execute. The Execute stage is where computations occur and where loads and stores access the D-cache. An NC that uses a CC's Execute stage also uses that CC's D-cache; the NC no longer uses its own D-cache.

Writeback. CCA does not require modifications for the Writeback logic, but it motivates a small change for register writing. Because the register writing logic is extremely small, it is preferable, in terms of area and performance, to simply replicate it (as a cold spare) in the original Writeback stage. Intuitively, forcing an NC to go to a CC for a tiny piece of logic is not efficient. If replication is not possible due to possible area constraints, this logic can be considered to be a component of the Decode stage.

2.4.2 CCA3: 3-Core CCA Implementation

We first consider a 3-core chip that we refer to as CCA3(2/1): 2 cores are NCs and 1 is CC. Our CCA3(2/1) implementation arranges the cores as shown in Figure 2.3, and we designate only the middle core, Core 2, as a CC. By aligning the cores in the same orientation, we facilitate routing from an NC to a CC. By provisioning one CC, we obtain better chip performance than if we had implemented CCA3(1/2), which would have 1 NC and 2 CCs. With more than one CC, the fault-free performance of each core decreases, due to added wires and multiplexing, and the ability to tolerate more faults does not increase much.

If a single fault occurs in either Core 1 or Core 3, it is preferable to just not

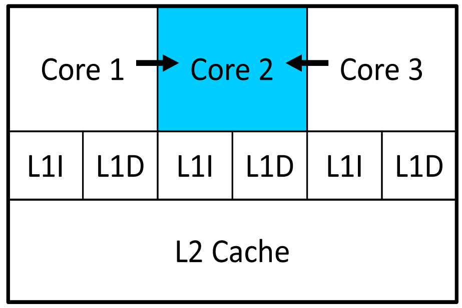


Figure 2.3: CCA3(2/1) Chip. Cores 1 and 3 are NCs, Core 2 is a CC. Arrows indicate the CC that provides spare components for each NC.

Table 2.1: Number of Inputs/Outputs per Stage for OR1200.

Stage	# Input signals	# Output signals
Fetch	56	65
Decode	38	115
Execute	110	61
Writeback	87	52

use that core, rather than cannibalize Core 2. Not using a core leads to a total chip performance of an NC and a CC combined, while borrowing a stage yields a chip performance of an NC and a borrowing NC. As we show in Section 2.5.2, the performance of an NC borrowing a stage is always lower than a fault-free CCA core, which is why we favor not using the faulty core.

CCA3(2/1)'s reconfigurability requires some extra hardware and wires, similar to the overhead required to be able to use spare components. Each NC (Core 1 and Core 3) has multiplexors (muxes) at the input to each stage that allow it to choose between signals from its own other stages (the majority of which are from the immediate predecessor stage) and those from the CC (Core 2). Similarly, Core 2 has multiplexors at the input to each stage that allow it to choose between signals from its other stages and signals from the two NCs. Table 2.1 shows the number of wires that are the inputs and outputs of each stage.

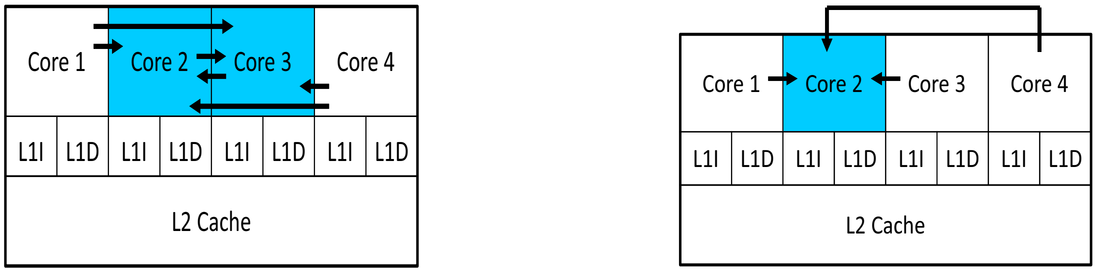
In CCA3(2/1)'s chip layout, the distance to route from Core 1 or Core 3 to Core

2 and back is short. The cores are small, and the distance each way is approximately 1mm in 90nm technology. Furthermore, because these simple cores are designed for power efficiency rather than for maximum clock frequency, we do not expect them to be clocked aggressively. Thus, given a clock frequency in the 400 MHz range and such short wires, the penalty of routing to and from a cannibalized stage is a relatively small fraction of the clock period (as we show in Section 2.5.2). Rather than add wire delay pipe stages to avoid lengthening the clock period (which we consider for our 4-core implementations in Section 2.4.3), we simply slow the clock slightly. For chips with larger cores, adding wire delay pipe stages may be preferable.

One way to mitigate the impact of lengthening the clock period is to use clock borrowing [129]. Consider a fault in Core 1. If Core 1's normal clock period is T and its extra wire delay to and from Core 2 is W (for our CCA chips W is twice the distance to access a spare component), then a simplistic solution is to increase Core 1's clock period to $T'=T+W$. Clock borrowing can mitigate this performance impact by amortizing time sharing W across the two neighboring stages [129]. By sharing this delay, we can reduce the clock period penalty to 1/3 of W, *i.e.*, $T'=T+W/3$. As a concrete example, if Core 1 has a 50ns clock period (T=50ns) when fault-free and W=15ns, then we can use time borrowing to achieve a clock cycle of T'=55ns. We borrow 5ns from both of the neighboring stages, pushing them from 50ns to 55ns. Thus, we have 65ns-10ns=55ns for the longer stage.

2.4.3 CCA4: 4-Core CCA Implementations

For the 4-core CCA chips we consider two viable CCA4 arrangements as illustrated in Figure 2.4. CCA4(3/1) chips are natural extensions of the CCA3(2/1) chip. In addition, we also propose the CCA4(2/2) configuration, which has two cannibalizable cores, and differs from CCA4(3/1) in how CCs share stages. In CCA4(2/2) Core 1 can use a stage from Core 2 or Core 3, Core 2 and Core 3 can use stages from each



(a) CCA4(2/2)

(b) CCA4(3/1)

Figure 2.4: CCA4 Chips. CCs are colored. Arrows indicate the CCs that provide spare components for each NC.

other, and Core 4 can use a stage from Core 3 or Core 2. This sharing policy allows CCs to share with each other, and it allows the NCs to share from their more distant CCs.

An important distinction between CCA3 and CCA4 chips (of any kind) is that, in a CCA4 chip, an NC may have to borrow a stage from a CC that is not an immediate neighbor. For example, in Figure 2.4(b), Core 4 is approximately twice as far from a CC as Core 3 is. Furthermore, as shown in Figure 2.4(a), a given NC might have different distances to the two CCs (*e.g.*, Core 4's distance to Core 2 and Core 3).

The increase in distance from an NC to a CC may, for some core microarchitectures, discourage the simple approach of lengthening the clock period of an NC that is using a cannibalized stage. In Figure 2.4(a), for example, there might be an unacceptable clock frequency penalty if we slow Core 1 to accommodate using a cannibalized stage from Core 3. Based on this clock penalty, we consider two approaches: the clock period lengthening we have already discussed and adding clock cycles to the pipeline. The first approach sacrifices clock frequency while the second approach sacrifices IPC and chip area. The preferred approach, in terms of overall performance, depends on the details of the core, so we discuss both configurations next.

CCA4-clock

The CCA4-clock design relies on increasing the clock period for distant CC accesses. This design is advantageous when the performance penalty of slowing the clock is preferable to adding pipeline stages. The only new issue for CCA4-clock, with respect to CCA3, is that it is possible that we want to have different pipeline stages of the same CC operate at different frequencies. For example, in Figure 2.4(b), if Core 1 is using Core 2's Decode stage and Core 4 is using Core 2's Execute stage, then we want Core 2's Decode stage to be at a higher frequency than its Execute stage. This difference results from Core 4 being further from the CC than Core 1 is from the CC. Prior work has shown how to provide different clocks within a single core [67]. However, if such a solution is considered too costly, then Core 2's clock frequency must be lowered to match the lowest frequency needed, such as the one imposed by Core 4 in the example. We use the CCA4-clock design for both CCA4(2/2) and CCA4(3/1) configurations. We refer to the latter as CCA4-clock(3/1) to differentiate it from its CCA4-pipe implementation that we describe next.

CCA4-pipe

The CCA4-pipe design, like CCA3, assumes that routing from an NC to an immediately neighboring CC can be efficiently accommodated by lengthening the clock period of the NC and the CC. However, it allows routing from an NC to a CC that is not an immediate neighbor to take one additional cycle, and routing back from the CC to the NC to account for another cycle. We do not lengthen the clock, because the wire and mux delays fit well within a cycle for a simple, relatively low-frequency core. To avoid adding too much complexity to the NC's control, we do not allow a single NC to borrow more than one stage that requires adding cycles.

When we add wire delay pipeline stages to a core's pipeline, we must add extra pipeline latches and solve four problems:

1. Conditional Branch Resolution. In the OR1200, the decision to take a branch is determined by a single signal, BranchFlag, that is continuously propagated from Execute back to Fetch. This BranchFlag is explicitly set/unset by instructions. Because the OR1200 has a single delay slot, the Fetch stage expects to see a BranchFlag signal that corresponds to the instruction that is exactly two instructions ahead of the current instruction in program order. However, adding cycles between Fetch and Execute can cause the BranchFlag signal seen by Fetch to be stale because it corresponds to an instruction that is more than two cycles ahead of it. To address this issue, we slightly modify the pipeline to predict that the stale BranchFlag value is the same as the value that would have been seen in the unmodified pipeline. We add a small amount of hardware to remember the program counter of a branch in case of a misprediction. If the prediction is correct, there is no penalty. A misprediction causes a penalty of two cycles.

2. Branch/Jump Target Computation. The target address is computed using a small piece of logic in the Decode stage, and having this unit close to the Fetch stage is critical to performance. As mentioned in Section 2.4.1, we treat this logic separately from the rest of the Decode stage, and we consider it to be logically associated with Fetch. Thus, if there is a fault in the rest of the NC's Decode stage, it still uses its original target address logic. This design avoids penalties for jump address computation.

3. Operand Bypassing. When an NC uses a CC's Execute stage, there are some additional bypassing possibilities. The output of the CC's Execute stage may need to be bypassed to an instruction that is in the wire delay stage of the pipeline right before Execute. Instead of adding a bypass path, we simply latch this data and bypass it to this instruction when it reaches the usual

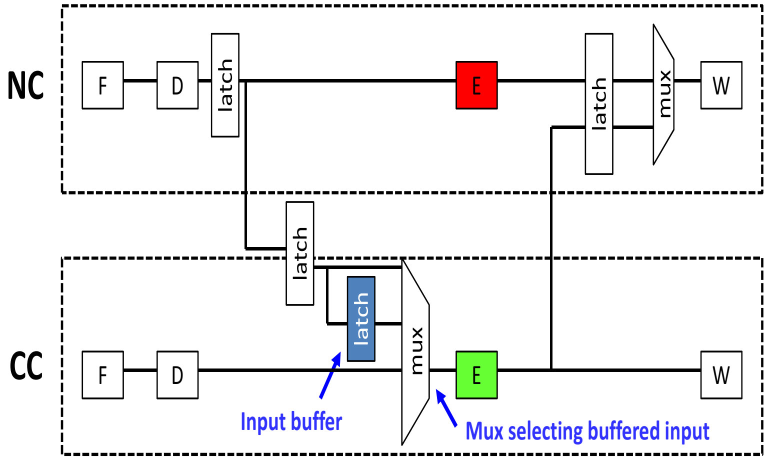


Figure 2.5: Input Buffering for CC's Execute Stage.

place to receive bypassed data (*i.e.*, when it reaches the Execute stage). We also slightly modify the Decode stage to set the correct values for the signals selecting the sources of the instruction's operands.

4. Pipeline Latch Hazards. The extra stages introduce two structural hazards for pipeline latches. First, if a cannibalized stage can incur an unexpected stall, then we must buffer this stage's inputs so they do not get overwritten. For the OR1200, Fetch and Execute require input buffering as illustrated in Figure 2.5, due to I-cache and D-cache misses, respectively. Second, if a cannibalized stage is upstream from (closer to Fetch than) a stage that can incur an unexpected stall, then the stall will reach the cannibalized stage late. To avoid overwriting the output of that stage, we buffer its output. For the OR1200, the Fetch and Decode stages require output buffering (Figure 2.6), because the Execute stage can stall on D-cache misses.

   If the area costs of buffering are considered unacceptably high, it is possible to squash the pipeline to avoid the structural hazards. For example, a D-cache miss triggers a squash of younger instructions. In our evaluation of CCA's area, we pessimistically assume the use of buffering rather than squashes, even

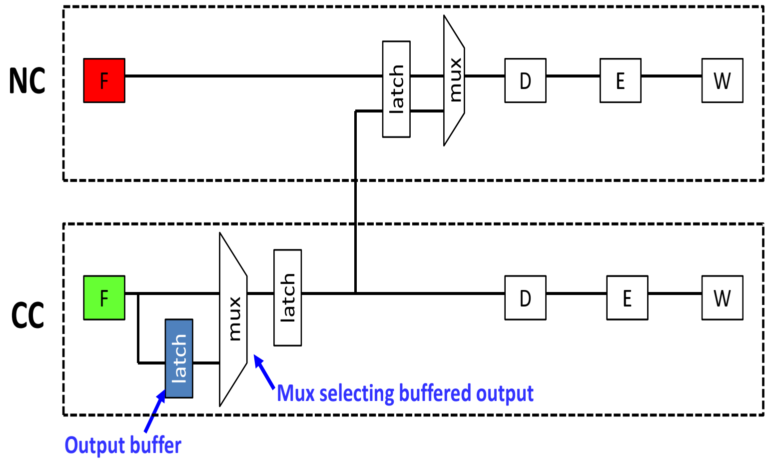


Figure 2.6: Output Buffering for CC's Fetch Stage.

though squashing on D-cache misses would have no IPC impact on the OR1200 because the pipe would refill before the D-cache miss resolves.

2.4.4 Many-core CCA Chips

Although we described until now CCA configurations with just three or four cores, CCA is easily extendable to many-core chips. One feasible and straightforward way to apply CCA to chips with more cores is to design these chips as groups of CCA3 or CCA4 clusters. We leave for future work the exploration and evaluation of unclustered designs for chips with greater numbers of cores.

2.5 Evaluation

Evaluating CCA designs requires us to consider two aspects. First, what is CCA's design impact over the baseline chip in terms of area and clock period? Second, how well do processors consisting of CCA3 and CCA4 clusters perform, compared to CS processors? In this section, we address both of these issues.

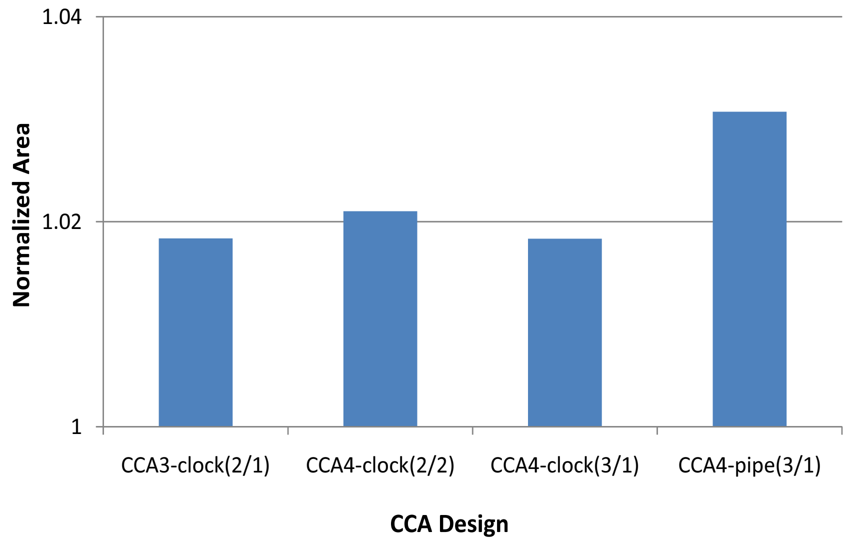


Figure 2.7: CCA Designs Area Overhead. Results are normalized with respect to the areas of CS designs with the same number of cores.

2.5.1 CCA Chip Area Overhead

CCA's area overhead is due to the logic and wiring that enable stages from CCs to be connected to NCs. In Figure 2.7, we plot the area overheads (compared to a CS chip with same number of cores) for various CCA chip implementations in 90nm technology. These areas include the entire chip: cores and the L1 I-caches and D-caches, which are both 8KB and 2-way set-associative (we do not consider L2 caches for our chips). We consider all of the following CCA designs: CCA3(2/1), CCA4-clock(3/1), CCA4-pipe(3/1), and CCA4-clock(2/2).

We observe that no CCA chip has an area overhead greater than 3.5%. CCA3(2/1) incurs less than 2% overhead, which is a difference so small that it requires more than 50 cores on the chip (*i.e.*, approximately 18 CCA3(2/1) clusters), before the additional area is equivalent to a single baseline core. The CCA4 overheads are comparable to the CCA3 overhead, except for CCA4-pipe, which requires some input/output buffering and modified control logic in the cores.

2.5.2 Lifetime Performance

The primary goal of CCA is to provide better lifetime chip performance than CS. We demonstrate in this section that CCA achieves this goal, despite the small per-core performance overheads introduced by CCA. To better understand these results, we first present our fault model, then evaluate fault-free single core performance (for both NCs and CCs) and the performance of an NC using a cannibalized stage.

We evaluate the performance for all cores and chips using the MediaBench benchmark suite [76] on the OpenRISC simulator [74]. We consider a core's performance to be the average runtime for all benchmarks in the suite relative to a baseline fault-free OR1200 core (*i.e.*, the relative average instructions per second (IPS)). Thus, the performance of a core is dictated by its frequency and the average IPC across benchmarks. We consider the performance of a fault-free OR1200 core to be 1. A CCA core that yields the same average IPC, but has a frequency of 10% less than the baseline core, has an overall performance of 0.9. The same performance characterizes a core operating at the same frequency as the baseline OR1200, but that has an average IPC degradation of 10%.

Fault Model

We consider only hard faults, and we choose fault rates for each pipeline stage that are based on prior work by both Blome *et al.* [20] and Srinivasan *et al.* [119]. Blome *et al.* [20] decomposed the OR1200 core into 12 structures (*e.g.*, fetch logic, ALU, load-store unit, *etc.*) and, for each structure, determined its mean time to failure in 90nm technology. Their analysis considered the utilization of each structure, and they studied faults due only to gate oxide breakdown. Thus, actual fault rates are expected to be greater [119] due to electromigration, NBTI, thermal stress, *etc.* Srinivasan *et al.* [119] assume that fault rates adhere to a lognormal distribution with a variance of 0.5. The lognormal distribution is generally considered more realistic for hard faults due to wearout because it captures the increasing rate of faults at the end of a chip's expected lifetime. The variance of 0.5 is a typical value for wearout phenomena. By combining these two results, we compute fault rates for each pipeline stage. We also consider faults in CCA-specific logic (including added latches and muxes), and we assume that these faults occur at a rate that is the average of the pipeline stage fault rates.

As industrial data regarding failure rates is not publicly available, in our experiments we consider the above-mentioned fault rates to be the nominal fault rates, and we also explore fault rates that are both more pessimistic (2x and 4x nominal) and less pessimistic (1/4x and 1/2x nominal). We assume that there are no faults present at time zero due to fabrication defects. The presence of fabrication defects would improve the relative lifetime performance of CCA with respect to CS by reducing the time until there are enough faults that CCA outperforms CS. We also do not consider faults in the cache interface logic, which CCA could handle, and thus we slightly further bias our results against CCA.

Fault-Free Single Core Performance

A fault-free NC or CC pays a modest performance penalty due to the multiplexors that determine from where each stage chooses its inputs. These muxes, which affect every pipeline stage, require a somewhat longer clock period to accommodate their latency. Also, CCA's additional area introduces some extra wiring delays, but the CAD tools revealed that this effect on the clock frequency is less than 0.3%. The mux delays are identical for NCs and CCs, and they are not a function of the number of cores or number of CCs. In CCA3(2/1), each NC is choosing from among two inputs (itself or the CC). The CC is choosing from among three inputs (itself and both NCs), and thus has a 3-to-1 mux. However, at least one of those inputs is not changing, so the critical path of this 3-to-1 mux is the same as that of a 2-to-1 mux.

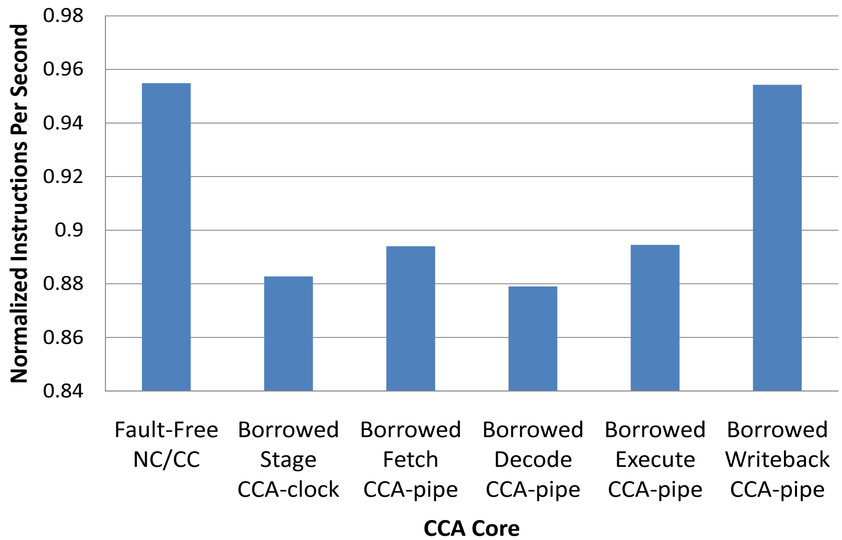


Figure 2.8: Performance of CCA Cores.

In the other CCA chips, the NC and CC muxes are either 2-to-1 or 3-to-1, but we can leverage the same observation about non-changing inputs. Thus, in all CCA chips, each NC and each CC has a clock period penalty that is equal to the latency of one 2-to-1 mux. This clock period penalty is 4.5% in 90nm technology.

Single NC Performance When Using CC

An NC's use of cannibalized stages introduces some performance degradation. In Figure 2.8, we plot the performance of an NC in several situations: fault-free, using any immediate neighbor CC's stage and extending the clock period, and using a CC's stage and adding pipeline stages (*i.e.*, for CCA4-pipe). Results are normalized to the performance (instructions per second) of a single baseline core that has none of CCA's added hardware. We compute wire delays based on prior work by Ho *et al.* [58], and we assume that the wires between NCs and CCs are routed using middle and upper metal layers. We use a modified version of the OpenRISC simulator to evaluate the IPC overhead for CCA4-pipe as a function of the cannibalized stage.

The results show that, when an NC borrows a CC's stage, the NC's slowdown is between 5% and 13%. Most slowdowns are in the 10-13% range, except when

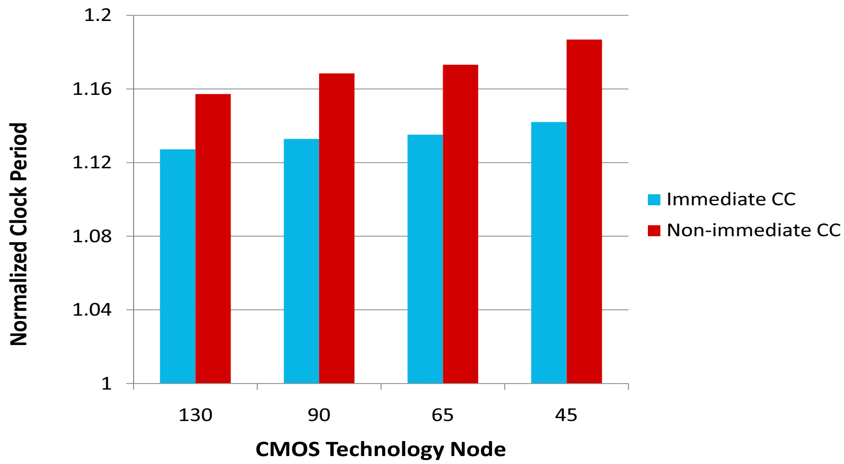
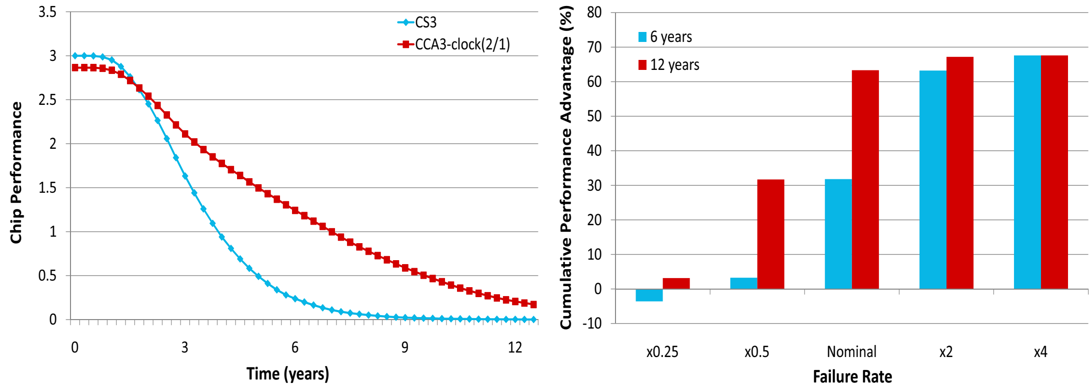


Figure 2.9: Relative Delay for Accessing Cannibalized Stages Function of Technology Node. Results are normalized with respect to the clock periods of the baseline core for the corresponding technology.

we add pipeline stages to borrow a Writeback stage; extending the Writeback stage incurs only a miniscule IPC penalty because exceptions are rare. The performance when slowing the clock to accommodate a borrowed stage (the second bar from the left in Figure 2.8) is a function of the technology node. In Figure 2.8, we assume a 90nm technology. For larger/smaller CMOS technologies, the wire delays are smaller/greater [58]. Figure 2.9 shows the delay to access a borrowed stage across different technologies. Even at 45nm, the delays remain under 15% and 19% for immediate and non-immediate neighbors, respectively. Even the worst-case 19% clock degradation for a core is still preferable to disabling the core.

Lifetime Processor Performance

CCA addresses faults that occur over the lifetime of the processor, and that have a probabilistic rate of occurrence. Therefore, we consider in our evaluation a chip's expected lifetime performance as a consistent measure unit. We extend the performance definition for a single core, and define chip performance as the aggregated performance of the chip's functioning cores. A CS3 chip with no faults has an expected performance of 3. CCA3(2/1) with no faults has an expected performance



(a) Lifetime performance for nominal fault rate

(b) CCA3-clock(2/1)'s cumulative performance advantage compared to CS3

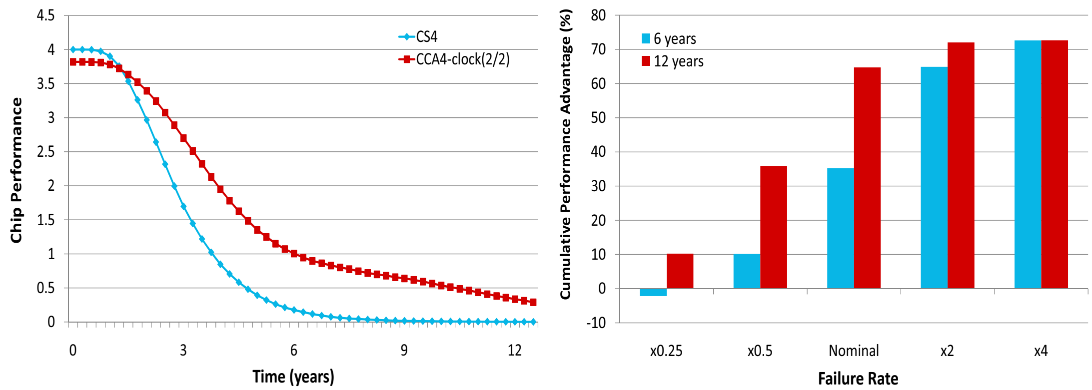
Figure 2.10: Lifetime Performance of 3-core Chips.

of 2.85, due to CCA3(2/1)'s clock penalty for mux delays. For brevity, we refer to "expected performance" as simply "performance".

To determine the aggregate chip performance in the presence of faults, we use Monte Carlo simulation. We develop Petri Net models of the CS and CCA chips that compute the expected performance of a chip as a function of time. We model each chip at the same 12-structure granularity as Blome *et al.* [20]. To evaluate a given chip, the Petri Net uses one million Monte Carlo simulations in which we inject hard faults in each of the processor structures (including CCA logic and latches), using the distributions previously specified (the million runs allow the results to converge). Once a fault occurs in a structure, the corresponding stage is considered unusable. For example, a fault in the ALU triggers the failure of the Execute stage. We do not consider the time needed to detect failures and reconfigure the chip. For each experiment we report values after 6 and 12 years, respectively, since we consider that a common industrial usage for a chip is between these time intervals.

We first evaluate chips with an equal number of cores, then compare performance of equal-area chips.

3-core Chips. Figure 2.10 plots performance over the lifetime of the chips. Fig-



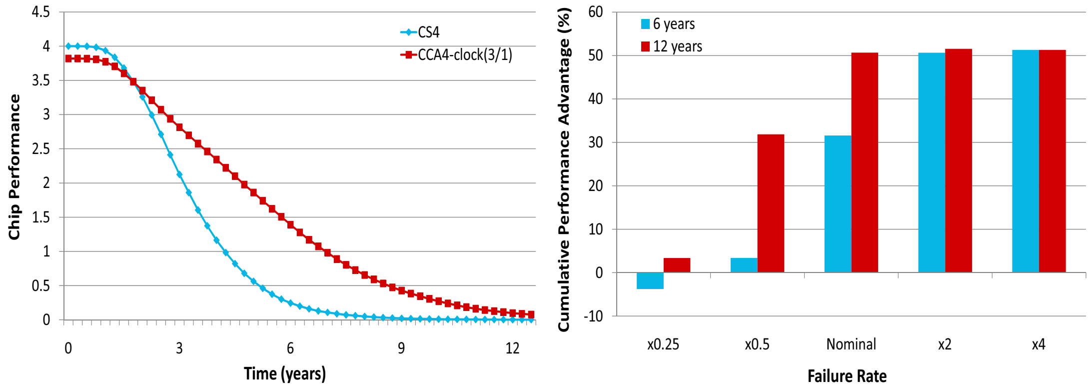
(a) Lifetime performance for nominal fault rate

(b) CCA4-clock(2/2)'s cumulative performance advantage compared to CS4

Figure 2.11: Lifetime Performance of CCA4-clock(2/2) Chips.

ure 2.10(a) shows the performance of 3-core chips, assuming the nominal fault rate. The difference between the curves at time zero reflects CCA's fault-free performance overhead. We observe that the crossover point (*i.e.*, the time at which the performances of CS3 and CCA3(2/1) are identical) is at a little under 2 years. After this early crossover point, CCA3(2/1)'s performance degradation is far less steep than CS3's. The CCA3 chip does not become instantaneously more advantageous, as it still has to recoup the performance loss during the fault-free case. For example, after 6 years, CCA3(2/1) outperforms CS3 by one fault-free baseline core.

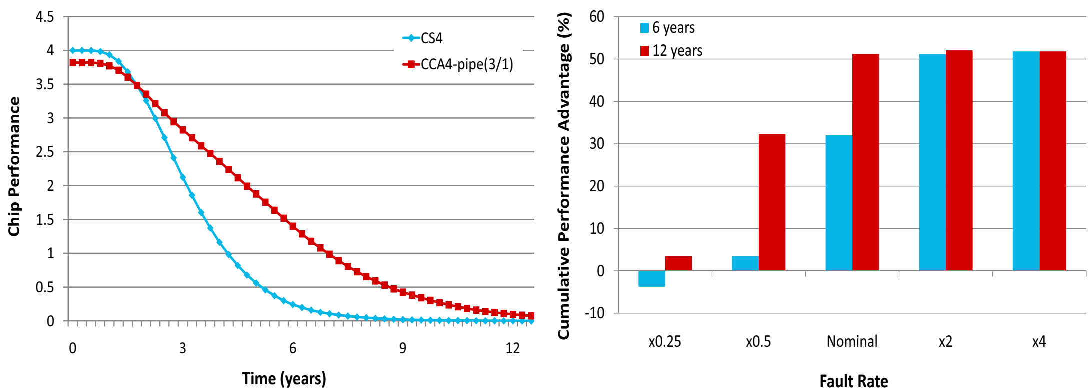
To better illustrate the importance of the gap between the curves in Figure 2.10(a), Figure 2.10(b) shows the cumulative performance for a variety of fault rates. The two bars for each fault rate represent the cumulative performance after 6 and 12 years, respectively. The cumulative performance is the integral (area under the curve) of the performance in Figure 2.10(a). For nominal fault rates or greater, CCA3(2/1) provides substantially greater cumulative lifetime performance. After only 6 years at the nominal fault rate, CCA3(2/1) has a 30% advantage, and this advantage grows to over 60% by 12 years. Even at only half of the nominal fault rate, CCA3(2/1) has achieved a 30% improvement at 12 years. For very low fault



(a) Lifetime performance for nominal fault rate

(b) CCA4-clock(3/1)'s cumulative performance advantage compared to CS4

Figure 2.12: Lifetime Performance of CCA4-clock(3/1) Chips.



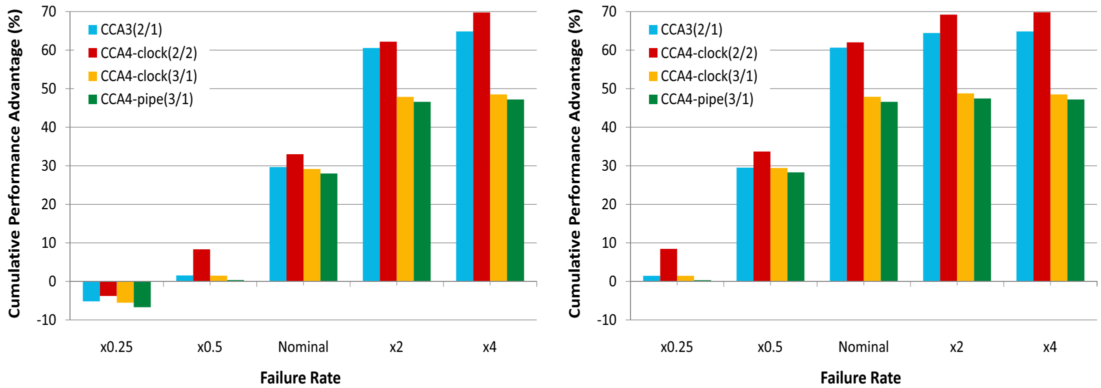
(a) Lifetime performance for nominal fault rate

(b) CCA4-pipe(3/1)'s cumulative performance advantage compared to CS4

Figure 2.13: Lifetime Performance of CCA4-pipe(3/1) Chips.

rates, CCA3(2/1) has slightly less cumulative performance after 6 years and slightly more cumulative performance after 12 years, but neither difference is substantial.

4-core Chips. We present the results for 4-core chips in Figures 2.11, 2.12 and 2.13, respectively. Similar to the CCA3 results, the crossover point when CCA chip outperforms CS is around 2 years for all CCA configurations (Figures 2.11(a), 2.12(a), and 2.13(a)). Figure 2.12(b) shows that CCA4-clock(3/1) achieves a greater than 50% improvement in cumulative lifetime performance for the nominal and twice-



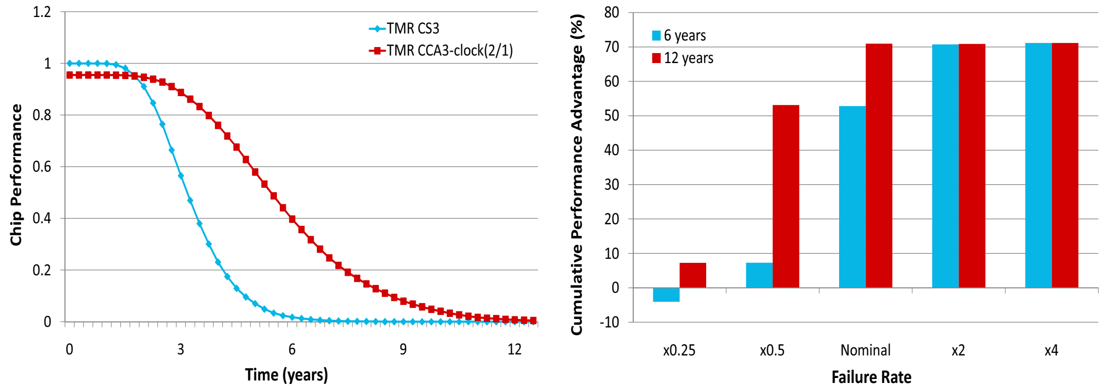
(a) 6-year cumulative results

(b) 12-year cumulative results

Figure 2.14: Lifetime Performance of Equal-Area Chips.

nominal fault rates. The results for the CCA4-pipe(3/1) are similar (Figure 2.13(b)). CCA4-clock(2/2) achieves the best performance improvement over CS, by taking advantage of the two CCs (Figure 2.11(b)). CCA4-clock(2/2) outperforms both CCA4(3/1) configurations, yielding improvements of 35% and 65% for the nominal fault rates over 6 years and 12 years, respectivelly.

Equal-Area Comparisons. The three-core and four-core results presented thus far are not equal-area comparisons. CCA chips are slightly (less than 3.5%) larger than CS chips. To provide another comparison point, we now compare chips of equal area. The ratio of the chips' performances is independent of the chip size. Figure 2.14 plots the cumulative performance advantages of the CCA chips. The figure demonstrates that the CCA3(2/1) and CCA4-clock(2/2) configurations are the most cost-effective designs for 90nm technology. These results are quite similar to the earlier results, because CCA's area overheads are fairly small. In addition, we bias the results against CCA by not considering L2 caches.



(a) Performance for nominal failure rate

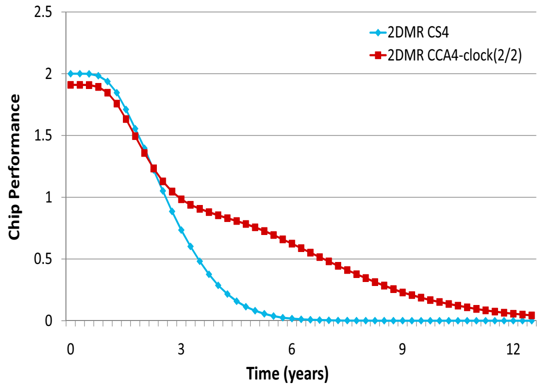
(b) CCA3-clock(2/1)'s cumulative performance advantage compared to CS3 for TMR configurations

Figure 2.15: Lifetime Performance of TMR Chips

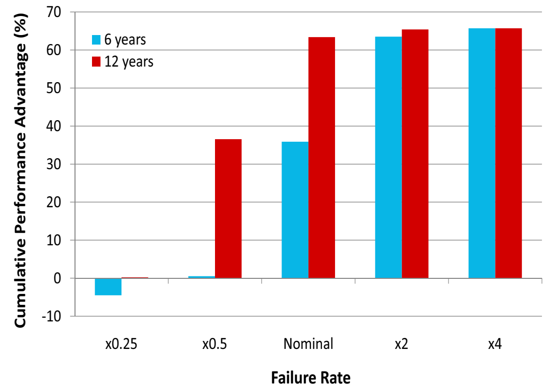
2.5.3 Performance of Chips Using TMR/DMR

We demonstrated that CCA outperforms CS chip by increasing core availability. Sustained availability is especially desired in fault tolerant architectures that use DMR or TMR configurations to provide resilience against failures. If multiple cores are used to provide error detection with DMR or error correction with TMR, then CCA is beneficial as it allows for more cores to be available. We consider the performance of a chip to be the performance of the slowest core in a DMR or TMR configuration. If fewer than 2 cores are available, the chip has zero performance (we assume the user is unwilling to use the processor without at least DMR to detect errors).

TMR. We plot the performance of 3-core chips that are being used in a TMR configuration in Figure 2.15. The crossover point is at about 2 years, similar to the comparison between CCA3 and CS3 in non-TMR configurations. However, the difference in cumulative performance is even greater. CCA3 provides more than 50% more cumulative performance for nominal and higher fault rates, even after only 6 years. At just half of the nominal fault rate, which is an optimistic assumption, CCA3 still has a 45% edge. The intuition for CCA's large advantage is that it greatly



(a) Performance for nominal failure rate



(b) CCA4-clock(2/2)'s cumulative performance advantage compared to CS4 for DMR configurations

Figure 2.16: Lifetime Performance of DMR Pair Chips

prolongs the chip's ability to operate in DMR mode. This analysis also applies to chips with more cores where the cores are grouped into TMR clusters.

DMR. We consider the performance of 4-core chips that comprise of two DMR pairs of cores (*i.e.*, 4 cores total). The first fault in any core leads to the loss of one core, and thus one DMR pair, for both CS4 and CCA4. Additional faults, however, are often tolerable with CCA4. Figure 2.16 shows the results for CCA4-clock(2/2), which is the best CCA4 design for this situation. Between approximately 2 and 2.5 years, CS4 and CCA4-clock(2/2) have similar performances. After that, though, CCA4-clock(2/2) significantly outperforms CS4. The cumulative results show that, for nominal and greater fault rates, CCA4-clock(2/2) provides lifetime advantages greater than 35% over 6 years and greater than 63% over 12 years.

Therefore, CCA is especially beneficial in supporting the high-availability requirements of TMR and DMR configurations.

2.6 Related Work

We compare CCA to prior work in self-repair, pooling of core resources, and lifetime reliability.

2.6.1 Multicore-Specific Self-Repair

Multicore processors are inherently redundant in that they contain multiple cores. Aggarwal *et al.* [10] proposed a reconfigurable approach to using multiple cores to provide redundant execution. When three cores are used to provide TMR, a hard fault in any given core will be masked. This use of redundant cores is related to the traditional fault tolerance schemes of multi-chip multiprocessors, such as IBM mainframes [117]. CCA is complementary to this work in that CCA enables a larger fraction of on-chip cores to be available for TMR or DMR use. Concurrently with our work, Gupta *et al.* [48] developed the StageNet multicore processor that is similar to the CCA concept [106], and in which the cores' pipeline stages are connected by routers. The StageNet chip enables greater flexibility in sharing resources than CCA, but incurs a greater performance overhead for this flexibility. Thus, CCA processors outperform StageNet ones for medium chip lifetimes of up to 10-12 years, while the latter outperform CCA chips over longer lifetimes.

2.6.2 Self-Repair for Superscalar Cores

Numerous researchers have observed that a superscalar core contains a significant amount of redundancy. Bower *et al.* [27] diagnose where a hard fault is–at the granularity of an ALU, reservation station, ROB entry, *etc.*–and deconfigure it. Shivakumar *et al.* [113] and Srinivasan *et al.* [119] similarly deconfigure components that are diagnosed by some other mechanism (*e.g.*, post-fabrication testing). Rescue [111] deconfigures an entire "way" of a superscalar core if post-fabrication testing uncovers a fault in it. CCA differs from all of this work by targeting simple cores with little intra-core redundancy. Finally, Powell *et al.* [97] proposed thread migration if a hard fault precludes the thread from executing on a core. The fault-and-migrate technique is efficient if the faulty unit is rarely used (*i.e.*, the fault impacts only a rarely executed set of instructions), such that migration does not occur often. Thus, their solution is targeted mostly towards multi-scalar cores and has limited applicability to simple cores.

2.6.3 Pooling of Core Resources

There have been proposals to group cores together during phases of high ILP. Both Voltron [134] and Core Fusion [65] allow cores to be dynamically fused and un-fused to accommodate the software. These schemes both add a substantial amount of hardware to allow tight coupling of cores, in the pursuit of performance and power-efficiency. CCA differs from this work by being less invasive. CCA's goals are also different in that CCA seeks to improve lifetime performance.

2.6.4 Lifetime Reliability

Srinivasan *et al.* [118, 119] have explored ways to improve the lifetime reliability of a single superscalar core. These techniques include adding spare components, exploiting existing redundancy in a superscalar core, and adjusting voltage and frequency to avoid wearing out components too quickly. CCA is complementary to this work.

2.7 Conclusions

For many-core processors with simple cores, there is an opportunity to improve lifetime performance by enabling sharing of resources in the presence of hard faults. The Core Cannibalization Architecture represents a class of designs that can retain performance and availability despite such faults. Although incurring slight performance overhead in fault-free scenarios, the CCA's advantages over the course of time outweigh this initial disadvantage. From among the CCA designs, we believe that CCA-clock designs are preferable to CCA-pipe designs. Even in those situations when CCA-pipe designs might yield a slightly better performance, it is not clear that their added complexity is worth this slight performance benefit. However, for future CMOS technologies, other core models, or cores with faster clocks, the CCA-pipe design may be worth its complexity.

Based on our results, we expect CCA (or similar designs) to excel in two domains in particular. First, for many embedded applications, the key metric is availability at a reasonable performance, more so than raw performance. Many embedded chips must stay available for long periods of time–longer than the average lifetime of a desktop, for example–and CCA improves this availability. Second, the CCA's significant benefits for chips that use cores in TMR and DMR configurations suggest that the design is a natural fit for chips using redundant cores to provide reliability.

3

Address Translation-Aware Memory Consistency

Current processors are vulnerable to design bugs in their address translation (AT) systems [2, 3, 4, 59, 61, 62, 63]. Possible causes for the multitude and constant occurrence of these design faults include the increased complexity of AT operations, as well as a lack of complete specifications for the interactions between the AT and the rest of the memory system. Such lack of precise specifications increases the difficulty of AT's pre-deployment testing and runtime verification. Consequently, we are unaware of any existing dynamic verification solutions that target AT. The result is that the AT system is vulnerable to design bugs, and any such design fault leads to costly processor deployment delays as in the recent case of the TLB coherence bug in the AMD Barcelona processor [131].

We believe that AT-related design bugs in modern processors are a direct result of designers' tendency to over-simplify memory consistency, and not account for how it is impacted by AT. Thus, memory consistency is considered a monolithic AT-independent interface between hardware and software. In this chapter, we address this problem by developing a framework for specifying AT-aware memory consistency models. We expand and divide memory consistency into 1) the physical address memory consistency (PAMC) model that defines the behavior of operations on physical address and 2) the virtual address memory consistency (VAMC) model that defines the behavior of operations on virtual addresses. As part of this expansion, we show what AT features are required to bridge the gap between PAMC and VAMC.

This chapter is structured as follows. We first describe the characteristics of the AT system that we consider in our analysis (Section 3.1). We continue by discussing the various levels of memory consistency that a system presents to its programmers (Section 3.2). We then focus on the two consistency models that are closely related to AT: PAMC (Section 3.3) and VAMC (Section 3.4), and formalize the crucial role of address translation in supporting a VAMC model. We then show how AT-operations can be integrated within a complete specification of VAMC models (Section 3.5), and describe how commercially available systems handle AT-related operations (Section 3.6). Finally, we discuss conclusions and future work (Section 3.7).

3.1 AT Fundamentals and Assumptions

Address translation is a level of indirection that regulates a software entity's (*i.e.*, thread or process) access to physical memory given a virtual address. We restrict our discussion to page-based AT systems and leave as future work other virtual memory paradigms such as segmentation. Architectures facilitate this level of indirection through translations, which are supported by a set of software managed structures called page tables.

A translation is a tuple <mapping(VP,PP), permissions, status>, where the mapping converts the virtual page VP to a physical page PP. PP, permissions, and status information are specified by the page table entry (PTE) defining the translation and that is uniquely identified by the VP. This association is unique within the virtual

```
generic MRF{
  acquire page table lock(s);
  create/modify the translation;
  enforce translation coherence (e.g., send TLB invalidations to other cores);
  release page table lock(s);
}
```

Figure 3.1: Pseudo-code for a Generic MRF.

memory context of the corresponding software entity. The permission bits include whether the page is owned by the user or the kernel and whether the page is readable, writeable, or executable. The status bits denote whether the page has been accessed or is dirty. In addition to these metadata bits, translations also contain a Valid bit that indicates if cores can access them in the page tables (*i.e.*, the translations are valid within the software's context). With respect to our analysis, all operations on this bit can be treated identically to operations on the mapping. Therefore, for simplicity, we do not consider separately the Valid bit in this chapter.

Accessing a translation is on the critical path of a memory access for most systems. Consequently, cores cache copies of the translations in private or shared translation caches (*i.e.*, translation lookaside buffers–TLBs) to speed up translation accesses. Changes to the PTEs result in translations being modified or invalidated in the page tables, and coherence must be maintained between the cached copies of the translations and the page table defined translations.

Translation updates. To create or delete a translation, or to modify a translation's mapping and/or permission bits, the privileged software (*i.e.*, kernel) relies on dedicated software routines that we refer to as *map/remap functions (MRFs)*. An MRF typically assumes the operations illustrated in Figure 3.1. Some of the activities in an MRF require complicated actions to be performed by the software or hardware. For example, ensuring translation coherence may require invalidating copies of the translation from all TLBs. This can be implemented by delivering TLB invalidations through either inter-processor interrupts or a global TLB invalidation instruction that relies on hardware for distributing the invalidations. We discuss in depth translation coherence in Chapter 5, while Section 5.1 describes the procedure typically used for ensuring translation coherence.

Status bits updates can be performed either explicitly by the kernel (*i.e.*, privileged programmer), or implicitly by the system (*i.e.*, hardware and possibly software). Status bits updates are usually not performed in MRFs as they do not require translation coherence, and occur atomically for the TLB-cached translation with respect to the memory PTE defining the translation. In an architecture with hardware-managed TLBs, the hardware is responsible for eventually updating the status bits. If the TLBs are software-managed, status bits updates occur in exception handlers.

AT's System Impact. AT encompasses both hardware and system software, and supports a system's virtual addresses memory operations. By sustaining the virtual address memory interface, AT can impact two aspects that determine the functionality of the memory system: memory coherence and memory consistency. While memory coherence refers to the visibility of writes to a single memory location by all cores in the system, memory consistency specifies the order in which a core's accesses to different locations in memory are observed by cores. The focus of this chapter is exclusively on AT's impact on memory consistency, motivated by the high number of AT design faults that are related to this aspect.

3.2 Memory Consistency Levels

A memory consistency specification defines the legal software-visible orderings of loads and stores performed by multiple threads. The consistency models serves as a contract between the system and the programmer. This contract is defined for a specific memory interface and is valid only for the programmer operating at this

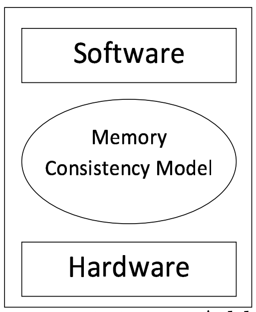


Figure 3.2: Address Translation-Oblivious Memory Consistency.

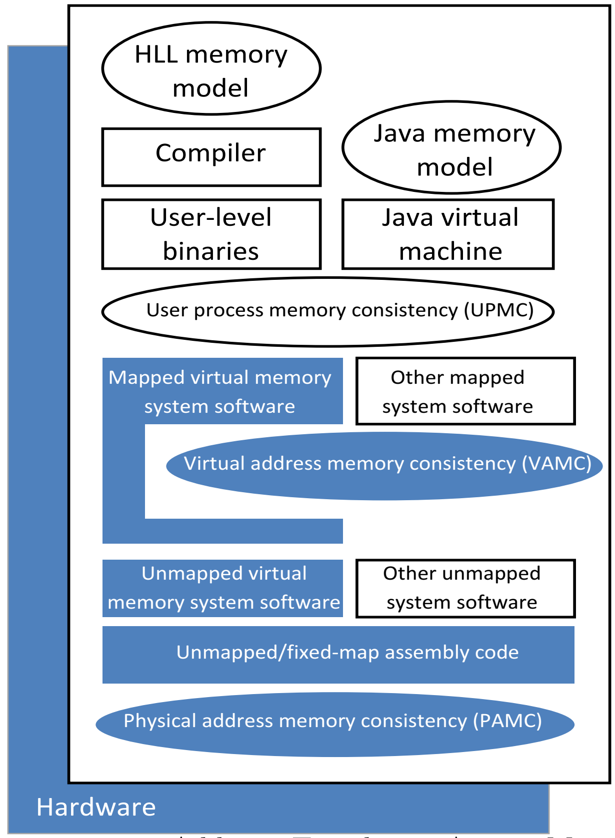


Figure 3.3: Address Translation-Aware Memory Consistency. Shaded portions are the focus of this chapter.

interface. Therefore, before specifying a consistency model, it is crucial to determine the interface at which the model applies to. Given this observation, in order to understand AT's impact on memory consistency, we must consider the different levels at which memory consistency specifications are defined and identify the ones that are impacted by AT.

The traditional view of memory consistency is that of one monolithic interface between the hardware and the software, as illustrated in Figure 3.2. Memory consistency, however, is a set of interfaces between the hardware and various levels of software, as illustrated in Figure 3.3. These memory consistency layers are a direct consequence of the different levels of abstractions that hardware and software support in a computing system [115].

Although Adve and Gharachorloo previously explained the multi-level nature of memory consistency [5], this more comprehensive definition of memory consistency is not always adopted in the community. For example, classical architecture books do not specify whether the model refers to virtual or physical addresses [54, 115]. In addition, existing consistency models such as sequential consistency (SC), processor consistency, weak ordering, release consistency, *etc.*, do not distinguish between virtual and physical addresses. Lamport's original definition of SC [73] is typical in that it specifies a total order of operations (loads and stores), but it does not specify whether the loads and stores are to virtual or physical addresses. Implicitly, most existing consistency models assume either unmapped software or software with a fixed one-to-one mapping from virtual to physical addresses. We refer to these consistency models as *AT-oblivious*.

In contrast with AT-oblivious models, understanding the impact of AT on memory consistency requires considering the hierarchical levels of memory consistency models described in Figure 3.3 and identifying which of these levels are impacted by AT. At each of these levels, the consistency model defines the legal orderings of the memory operations available at that level. We position hardware below all levels, as the microarchitecture represents the lowest level that provides mechanisms that can be used to enforce consistency models at various levels (*e.g.*, the core provides in-order instruction commit). We limit our discussion to four levels relevant to programmers that are present in most current computing systems. These consistency models are necessary interfaces that are included in the specifications of the ISA, ABI, and API. However, for the purposes of our current analysis, we do not need to consider which interfaces belong in which specifications. We discuss these levels, starting at the lowest level:

- *Physical address memory consistency* (*PAMC*): Some software such as unmapped code or boot code, as well as the code managing the AT system, rely exclusively on PAMC. Implementing PAMC is the hardware's responsibility and, as such, is specified precisely and completely in the architectural manual (*i.e.*, ISA).

- *Virtual address memory consistency* (*VAMC*): VAMC is the level just above the PAMC. All mapped software (*i.e.*, software that executes using virtual addresses) relies upon VAMC, including mapped system software. VAMC builds upon PAMC, and requires support from both hardware and, usually, AT software (we are unaware of a system that currently relies exclusively on hardware for supporting VAMC, although such a system might prove feasible to build considering the increasing number of on-die available transistors). Perhaps one non-intuitive aspect of VAMC is that mapped virtual memory system software both relies upon VAMC and helps to support it.

- *User process memory consistency* (*UPMC*). UPMC is specified by the software whenever additional ordering is required on memory accesses beyond VAMC. Thus, UPMC may either be identical to VAMC, or it could differ, as in the case of software transactional memory or software distributed shared memory.

- *High-level language consistency*: At the highest level, user-level programmers see the consistency model specified by the high level language [6], such as the consistency models provided by C++ [23] or Java [83]. These models are supported by the compilers, runtime systems, and lower level consistency models.

As shown in Figure 3.3, PAMC and VAMC are important interfaces that support different layers of software. Correct PAMC is required for unmapped code to work correctly, and correct VAMC is required for mapped code to work correctly. The AT

Table 3.1: SC PAMC. Loads and stores are to physical addresses. An X denotes an enforced ordering.

		Operation 2	
		Load	Store
Operation 1	Load	X	X
	Store	X	X

Table 3.2: Weak Order PAMC. Loads and stores are to physical addresses. MemBar denotes a memory barrier. An X denotes an enforced ordering. An A denotes an ordering that is enforced if the operations are to the same physical address. Empty entries denote no ordering.

		Operation 2		
		Load	Store	MemBar
Operation 1	Load		A	X
	Store	A	A	X
	MemBar	X	X	X

system intermediates the transition between the two consistency levels, and directly impacts the upper layer, VAMC. Without a correct AT system, a system with virtual memory cannot enforce *any* VAMC model.

In the next sections, we focus on these two consistency layers and explain how to adapt well-known existing consistency models to these levels. We present a VAMC specification and show how it differs from PAMC, discuss how AT bridges the gap between PAMC and VAMC, and describe how AT impacts both system programmers and verification.

3.3 Specifying PAMC

We specify consistency models at all levels using a table-based scheme like those of Hill *et al.* [56] and Arvind and Maessen [14]. The table specifies which program orderings are enforced by the consistency model. Some consistency models have atomicity constraints that cannot be expressed with just a table (*e.g.*, stores are atomic as is the case for TSO). We can specify these models by augmenting the table with a specification of atomicity requirements, as in prior work [14], although we do not consider such models in this chapter.

The specifications for PAMC can be straightforwardly adapted from the AT-oblivious consistency model specifications, by precisely stating that PAMC rules are applicable to physical addresses only. Thus, for a sequentially consistent PAMC model (SC PAMC), the specifications would state that (a) there must exist a total order of all loads and stores to physical addresses that respects the program orders of the threads, and (b) the value of each load is equal to the value of the most recent store to that physical address in the total order. Table 3.1 presents the specifications for the SC PAMC, while Table 3.2 presents the adaptation for a Weak Ordering PAMC, respectively. Under SC, all physical address memory operations must appear to perform in program order. Under Weak Ordering, memory operations are unordered.

3.4 Specifying VAMC

VAMC extends the PAMC specifications to also include mapped instructions. Although adapting an AT-oblivious consistency model for PAMC is straightforward, there are three challenges when adapting an AT-oblivious consistency model for VAMC: 1) synonyms, 2) mapping and permission changes, and 3) load/store side effects. These challenges are based on AT aspects that directly impact VAMC orderings, and we discuss both their impact on the programmer as regulated through the VAMC interface, as well as on the verification of the VAMC level.

3.4.1 Synonyms

The first challenge is the possible existence of *synonyms*, *i.e.* multiple virtual addresses (VAs) that map to the same physical address (PA). Consider the example in Figure 3.4, in which VA1 and VA2 map to PA1. SC requires a total order in which the value of a load equals the value of the most recent store to the same address.

Thread 1	Thread 2
Store VA1=1	
	Store VA2=2
	Load y=VA1
Load x=VA2	

Figure 3.4: Example of Synonym Problem. Assume VAMC sequential consistency and that VA1 and VA2 map to PA1. Assume that PA1 is initially zero. A naive VAMC implementation incorrectly allows (x,y)=(2,1).

Unfortunately, naively applying SC at the VAMC level allows an execution in which x=2 and y=1. The programmer expects that the loads in both threads will be assigned the value of the most recent update to PA1. However, a naive definition of VAMC that did not consider the level of indirection introduced by AT, would allow x to receive the most recent value of VA2 and y to receive the most recent value of VA1, without considering that they both map to PA1. To overcome this challenge, we re-formulate AT-oblivious consistency models for VAMC by applying the model to *synonym sets of virtual addresses* rather than individual addresses. For example, we can define SC for VAMC as follows: there must exist a total order of all loads and stores to virtual addresses that respects program order and in which each load gets the value of the most recent store to *any virtual address in the same virtual address synonym set*. Similar modifications can be made to adapt other AT-oblivious consistency models for VAMC.

Impact on Programming. Programmers that utilize synonyms generally expect ordering to be maintained between accesses to synonymous virtual addresses. Incorporating synonyms explicitly in the consistency model enables programmers to reason about the ordering of accesses to virtual addresses.

Impact on VAMC Verification. Explicitly stating the ordering constraints of synonyms is necessary for verification. An error in the address translation hardware could result in a violation of ordering among synonyms that might not be detected

Buggy Code: Thread 1	Buggy Code: Thread 2	Correct Code: Thread 1	Correct Code: Thread 2
MRF {map VA1 to PA2; tlbie VA1; // invalidate // translation // (VA1→PA1) }		MRF {map VA1 to PA2; tlbie VA1; // invalidate // translation // (VA1→PA1) }	
		tlbsync // fence for MRF	
sync; // memory barrier for // regular memory ops		sync; // memory barrier for // regular memory ops	
Store VA2 = B		Store VA2 = B	
sync	while (VA2!=B) {spin}	sync	while (VA2!=B) {spin}
	sync		sync
	Store VA1 = C		Store VA1 = C
	sync		sync
	Store VA2 = D		Store VA2 = D
while (VA2 != D) {spin}		while (VA2 != D) {spin}	
sync		sync	
Load VA1 // can get C or A		**Load VA1 // can only get C**	

Figure 3.5: Power ISA Code Snippets to Illustrate the Need to Consider MRF Ordering. Initially, VA1 is mapped to PA1, and the value of PA1 is A. Enforcing MRF serialization through *tlbsync* (right-hand side) eliminates result ambiguity (left-hand side).

without the formal specification.

3.4.2 Mapping and Permission Changes

The second challenge is that there is a richer set of memory operations at the VAMC level than at the PAMC level. User-level and system-level programmers at the VAMC interface are provided with OS software routines to map and remap or change permissions on virtual memory regions (*i.e.*, MRFs), such as the *mk_pte()* ("make new page table entry") or *pte_mkread()* ("make page table entry readable") functions in Linux 2.6.

Impact on Programming. The code snippet in the left-hand side of Figure 3.5, written for a system implementing the Power ISA, illustrates the need to consider MRFs and their ordering. We expect that the load by Thread 1 should return the value C written by Thread 2, because that appears to be the value of the most recent write (in causal order, according to the Power ISA's weak ordered memory model). However, this code snippet does not guarantee *when* the translation coherence request (*i.e.*, *tlbie* instruction) will be observed by Thread 2 and thus Thread 2 could continue to operate with the old translation of VA1 to PA1. Therefore, Thread 2's Store to VA1 could modify PA1. When Thread 1 performs its load to VA1, it could access PA2 and thus obtain B's old value.

The problem with the code is that it does not guarantee that the invalidation generated by the *tlbie* instruction will execute on Thread 2's core before Thread 2's store to VA1 accesses its translation in its TLB. Understanding only the PAMC model is not sufficient for the programmer to reason about the behavior of this code; the programmer must also understand how MRFs are ordered. We show a corrected version of the code on the right-hand side of Figure 3.5. In this code, Thread 1 executes a *tlbsync* instruction that is effectively a fence for the MRF and the associated translation coherence operation. Specifically, the *tlbsync* guarantees that the *tlbie* instruction executed by Thread 1 has been observed by other cores as for Power ISA the memory barriers (*i.e.*, *sync*) only order normal load and stores, and not MRFs.

Impact on VAMC Verification. Similar to the above programming example, a runtime hardware error or design bug could cause a TLB invalidation to be dropped or delayed, resulting in TLB incoherence. A formal specification of MRF orderings is required to develop proper verification techniques, and PAMC is insufficient for this purpose.

3.4.3 Load/Store Side Effects

The third challenge in specifying VAMC is that loads and stores to virtual addresses have side effects. The AT system includes status bits (*e.g.*, Accessed and Dirty bits) for each page table entry. These status bits have an informative aspect for the kernel and are part of the architectural state, and the ordering of updates to those bits must thus be specified in VAMC. To achieve this we add two new operations to the specification tables: Ld-sb (load's impact on status bits) and St-sb (store's impact

```
Store VA1=1; // VA1 maps to PA1
Load VA2; // VA2 maps to the page table entry of VA1

 /* The load is used by the VM system to determine if
the page mapped by VA1 needs to be written back to
secondary storage. */
```

Figure 3.6: Code Snippet to Illustrate the Need to Consider Load/Store Side Effects. If the two instructions are reordered, a Dirty bit set by the store could be missed and the page incorrectly not written back.

on status bits).

Impact on Programming. Consider the example in Figure 3.6. Without knowing how status updates are ordered, the OS cannot be sure what state will be visible in these bits. It is possible that the load of the page table entry occurs before the first store's Dirty bit update. The OS could incorrectly determine that a writeback is not necessary, resulting in data loss.

Impact on VAMC Verification. Without a precise specification of status bit ordering, verification could miss a situation analogous to the software example above. A physical fault could lead to an error in the ordering of setting a status bit, and this error could be overlooked by dynamic verification hardware and lead to silent data corruption.

3.5 AT-aware VAMC Specifications

Considering the AT aspects that influence VAMC, we present two possible VAMC adaptations of SC and Weak Ordering in Table 3.3 and Table 3.4, respectively. These specifications include MRFs and status bit updates, and loads and stores apply to synonym sets of virtual addresses (not individual virtual addresses). The weak ordering VAMC allows status bits to be reordered with respect to loads, stores, and other status bit updates. These specifications provide both a contract for programmers and enable development of techniques to verify correct memory system operation.

Table 3.3: SC VAMC. Loads and stores are to synonym sets of virtual addresses. An X denotes an enforced ordering.

		Operation 2				
		Ld	Ld-sb	St	St-sb	MRF
Operation1	Ld	X	X	X	X	X
	Ld-sb	X	X	X	X	X
	St	X	X	X	X	X
	St-sb	X	X	X	X	X
	MRF	X	X	X	X	X

Table 3.4: Weak Order VAMC. Loads and stores are to synonym sets of virtual addresses. MemBar denotes a memory barrier. An X denotes an enforced ordering. An A denotes an ordering that is enforced if the operations are to the same physical address. Empty entries denote no ordering.

		Operation 2					
		Ld	Ld-sb	St	St-sb	MemBar	MRF
Operation1	Ld			A		X	X
	Ld-sb					X	X
	St	A		A		X	X
	St-sb					X	X
	MemBar	X	X	X	X	X	X
	MRF	X	X	X	X	X	X

Alternative VAMC Models

The two VAMC models that we presented in the previous section are clearly not the only possibilities. For example, both of these adaptations strictly order MRFs, but other MRF orderings are possible. We are unaware of any current system that relaxes the ordering between MRFs that modify mappings and other memory operations, but at least one ISA (Power ISA) allows MRFs that upgrade permissions to be reordered with respect to certain memory operations. For example, an MRF that adds write permission to a region that currently only has read permission can be reordered with respect to loads since they are unaffected by the permission change [125]. However, we expect most VAMC models to order this type of MRF with respect to stores.

Another example of an alternative VAMC model is one in which all MRFs can be reordered unless an explicit fence-like instruction for MRFs is used, which could be a

Table 3.5: Address Translation in Commercial Architectures.

ISA	PAMC	AT Mechanisms: TLB Mgmt.	AT Mechanisms: TLB Coherence Mechanisms	Architecture's Impact on VAMC: Invalidation Processing	Architecture's Impact on VAMC: Permissions Consistency
MIPS	SC	software	inter-processor interrupt (IPI)	immediate	strict
IA-32, Intel64	processor consistency	hardware	IPI	immediate	relaxed
IA-64	release consistency	hardware & software	IPI and global TLB invalidation	deferred	relaxed
AMD64	processor consistency	hardware	IPI	immediate	relaxed
SPARC	TSO, PSO, RMO	software	IPI (sent directly to the MMU)	immediate	strict
PowerISA	weak consistency	hardware	IPI and global TLB invalidation	deferred	strict

Memory Barrier (MemBar) or a dedicated instruction for ordering MRFs. Analogous to relaxed memory consistency models, software uses a serializing instruction, like the Power ISA's *tlbsync*, to enforce order when it wishes to have order, but the default situation allows a core to defer invalidations due to MRFs.

3.6 Commercial VAMC Models

In Table 3.5, we compare the PAMC models and AT systems of six currently available commercial architectures. There is a considerable diversity in PAMC models and hardware support for AT. For example, while all platforms implement TLB coherence, some architectures provide inter-processor interrupts for maintaining TLB coherence, whereas other architectures support TLB coherence by providing privileged instructions for invalidating TLB entries on other cores.

Current architectures cannot specify VAMC because their VAMC models require software support. As mentioned in Section 3.2, this is not a fundamental constraint, and a hardware-only AT implementation might allow future ISAs to also specify VAMC. An architecture can state what software should do to achieve a particular VAMC model (*e.g.*, as part of the ABI). Some commercial architectures consider AT's impact on memory consistency to a limited extent. For example, SPARC v9 [128] assumes that a store to one virtual address modifies the values of all other synonyms. Intel's IA-64 model [60] assumes a one-to-one mapping between virtual and physical addresses. In the rightmost two columns of Table 3.5 we list, for each architecture, its impact on two aspects of VAMC: (a) whether a TLB invalidation must be processed immediately or can be deferred and (b) whether translation permission bits must be strictly coherent. Thus, PAMC and the AT mechanisms impact the VAMC model that can be supported by a platform. For example, an architecture with relaxed permissions coherence might not be able to enforce some of the orderings in VAMC tables like Tables 3.3 and 3.4.

3.7 Conclusions and Future Work

In this chapter we have developed a framework for specifying a system's memory consistency at two important levels: PAMC and VAMC. Having a thorough, multi-level specification of consistency enables programmers, hardware designers, and design verifiers to reason easily about the memory system's correctness.

The current analysis represents a first step to the exploration of AT's impact on the memory system. We foresee future research into VAMC models and AT systems, as well as the relationship between them. One important aspect of future work is to explore AT models and determine what is required to yield weaker VAMC models. More relaxed VAMC specifications are only viable if designers and verifiers can convince themselves that these models are correct. Our framework for specifying VAMC enables these explorations.

The incentive to explore weaker VAMC models is that, similar to weaker PAMC models, they might lead to increased performance. Such performance gains depend on what VAMC aspects can be relaxed, as well as the frequency of these serialization points in current applications. A first direction to pursue is to reduce the overly constraining requirement of MRF serialization with respect to other MRFs, as well as regular instructions. Current models do not distinguish between MRFs to different translations and require MRF serialization with respect to all instructions, even if they are unaffected by the MRF. Such weaker VAMC models might prove beneficial, especially for systems that rely heavily on MRFs.

Another possible research direction is the implementation of a hardware-only AT system. The increasing number of available transistors allows us to consider the design of an AT coprocessor that handles page table management, memory allocation and paging. This coprocessor would allow the ISA to fully specify VAMC, and the system to perform AT operations faster than using software routines. An in-depth analysis is required to establish if the hardware can perform all required functions more efficiently than software, considering the complex data structures used by virtual memory management or the per-process paging bookkeeping.

Finally, the framework we introduced in this chapter can be extended to incorporate segmentation and virtualization aspects. Including these aspects results in a complete specification of virtual address memory consistency. In this context, segmentation can be approached analogously to paging, both concepts representing levels of indirection from virtual to physical addresses.

4

Dynamically Verifying Address Translation

Although dynamic verification schemes exist for AT-oblivious memory systems [29, 87, 88], no such solutions exist for AT-aware models. The framework we proposed in the previous chapter allows us to consider such solutions by decomposing the verification procedure into PAMC and AT-related mechanisms. Because there are no existing solutions for checking AT correctness, we develop DVAT, a scheme to dynamically verify AT. We demonstrate that for a particular AT model, combining DVAT with an existing scheme for dynamic verification of PAMC [29, 87, 88] is sufficient for dynamic verification of VAMC.

In this chapter, we first discuss the AT model we consider in our evaluation, AT_{SC}, that can be formally proven to bridge the gap between two specific PAMC and VAMC models (Section 4.1). We then construct a framework for specifying AT systems (Section 4.2) that helps architects to reason about correct AT functionality and to develop checkers for runtime verification of AT. Based on this framework, we propose a dynamic verification mechanism for AT_{SC} (Section 4.3). When combined with PAMC dynamic verification and timeouts, our AT dynamic verification solution can capture the AT-related design bugs mentioned in Section 1.2. We experimentally evaluate DVAT's fault detection efficiency and performance impact using a full system simulator (Section 4.4). We then compare our work to prior work (Section 4.5), and discuss conclusions and future work (Section 4.6).

4.1 AT Model: AT_{SC}, a Provably Sufficient Sequential AT Model

In our analysis we consider an AT model that, when combined with SC PAMC ($PAMC_{SC}$ - see Table 3.1), is provably sufficient for providing SC VAMC ($VAMC_{SC}$ - Table 3.3). This AT model, which we call AT_{SC}, is quite similar, but not identical, to the model characterizing current Linux platforms. Compared to existing AT models, AT_{SC} is more restrictive and conservative. Nevertheless, AT_{SC} is realistic as, for example, the AT system of the Sequoia machines [107] fits this model.

AT_{SC} is a *sequential model* of an AT system. Because it is a model, it is a logical abstraction that encompasses the behaviors of a variety of possible physical implementations. The three key aspects of this model are:

- MRFs logically occur instantaneously and are thus totally ordered with respect to regular loads and stores and other AT operations. For example, Linux enforces this aspect of the model using locks.

- A load or store logically occurs instantaneously and simultaneously with its corresponding translation access (accessing the mapping, permissions, and status bits) and possible status bit updates. A core can adhere to this aspect of the model in many ways, such as by snooping TLB invalidations between when a load or store executes and when it commits. A snoop hit forces the load or store to be squashed and re-executed. Another possibility to enforce this behavior is for the core to flush the pipeline before executing a TLB translation invalidation or a full TLB flush.

- A store atomically updates all the values in the synonym set cached by the core executing the store, and a coherence invalidation atomically invalidates all of the values in the synonym set cached by the core receiving the invalidation. To our knowledge, current systems adhere to this aspect of the model either by using physical caches or by using virtual caches with same index mapping for synonym set virtual addresses.

These properties ensure that AT_{SC} bridges the gap between PAMC_{SC} and VAMC_{SC}.

$$PAMC_{SC} + AT_{SC} = VAMC_{SC}$$

PAMC_{SC} specifies that all loads and stores using physical addresses are totally ordered. AT_{SC} specifies that a translation access occurs instantaneously and simultaneously with the load or store. Under AT_{SC}, all MRFs are totally ordered with respect to each other and with respect to loads and stores. AT_{SC} also specifies that accesses to synonyms are ordered according to PAMC_{SC} (*e.g.*, via the use of physical caches). Therefore, all loads and stores using virtual addresses are totally ordered. Finally, AT_{SC} specifies that status bit updates are performed simultaneously with the corresponding load or store, and thus status bit updates are totally ordered with respect to all other operations. Hence, PAMC_{SC} plus AT_{SC} results in VAMC_{SC}, where ordering is enforced between all operations (see Table 3.3).

4.2 A Framework for Specifying AT Correctness

AT_{SC} is just one possible model for AT and thus one possible bridge from a PAMC model to a VAMC model. In this section, we present a framework for specifying AT models, including AT models that are more relaxed than the one presented in Section 4.1. A precisely specified AT model facilitates the verification of the AT system and, in turn, the verification of VAMC. We have not yet proved the sufficiency of AT models other than AT_{SC} (*i.e.*, that they bridge any particular gap between a PAMC and VAMC), and we leave such analysis for future work. However, the framework that we propose is applicable to most currently available AT models, including AT_{SC}.

Our framework consists of two invariants that are enforced by a combination of hardware and privileged software:

- The page table is correct (Section 4.2.1).
- Translations are "coherent" (Section 4.2.2). We put quotes around coherent, because we consider a range of definitions of coherence, depending on how reordered and lazy the propagation of updates is permitted to be. All systems of which we are aware maintain translation mapping coherence and coherence for permissions downgrades, either using software routines, an all-hardware protocol [105], or a combined hardware/software approach. Systems may or may not specify that status bits and/or permissions upgrades are also coherent. In our analysis, without loss of generality, we assume that translations in their entirety are coherent.

4.2.1 Page Table Integrity

For AT to behave correctly, the contents of the page table must contain the correct translations. This definition of correctness includes aspects such as: translations have the correct mappings (*e.g.*, the physical page exists), the metadata bits are consistent (*e.g.*, a translation is writeable, but not readable), and the translation's mappings maintain a correct page table structure as specified by the ISA, if the ISA specifies such a structure.

The page table is simply a data structure in memory that we can reason about in two parts. The first part is the root (or lowest level of the table) of the page table. The root of the address space is at a fixed physical address and uses a fixed mapping from virtual to physical address. The second part, the page table content, is dynamically mapped and thus relies on address translation.

To more clearly distinguish how hardware and software collaborate in the AT system, we divide page table integrity into two sub-invariants:

- [**PT-SubInv1**] The translations are correctly defined by the page table data structure.

  This sub-invariant is enforced by the privileged code that maintains the page table.

- [**PT-SubInv2**] The root of the page table is correct.

  Cores rely on a correct root to access PTEs during page table walks. This sub-invariant is enforced by hardware (as specified by PAMC), since the root has a fixed physical address.

4.2.2 Translation Coherence

Translation coherence is similar but not identical to cache coherence for regular memory. All cached copies of a translation (in TLBs) should be coherent with respect to the page table. The notion of TLB coherence is not new [125], although it has not previously been defined precisely, and there have been many different implementations of AT systems that provide coherence (we detail these implementations in Chapter 5). Briefly, there are many possible definitions of translation coherence. The differences between these definitions of coherence are based on when translation updates must be made available to other cores (*e.g.*, immediately or lazily) and whether updates may be reordered. Our focus in this work is on a specific definition of coherence that is consistent with AT_{SC}, where translation updates are immediately made visible to other cores, and updates cannot be reordered.

We specify AT correctness using a set of invariants that an AT system must maintain to provide translation coherence. These invariants are independent of the protocol that is implemented to maintain the invariants, and provide an implementation-transparent correctness specification. We choose to specify the translation coherence invariants in a way that is similar to how cache coherence invariants were specified in Martin *et al.*'s Token Coherence [84] paper, with AT-specific differences highlighted. We have chosen to specify the invariants in terms of tokens, as is done in Token Coherence, in order to facilitate our specific scheme for dynamically verifying the invariants, as explained in Section 4.3. This framework is just one possible approach. Depending on the purpose they serve, other AT models might rely on a different set of invariants.

We consider each translation to logically have a fixed number of tokens, T, associated with it. Ideally, for a translation, there should be one token for each active (*i.e.*, running) thread in the system that can access the translation. However, for multithreaded processors, threads share the processor's TLB and thus we require one token per TLB. Hence, T must be at least as great as the number of TLBs in the system. Tokens may reside in TLBs or in memory. The following three sub-invariants are required:

- **[Coherence-SubInv1]** At any point in logical time [72], there exist exactly T tokens for each translation.

  This "conservation law" does not permit a token to be created, destroyed, or converted into a token for another translation.

- **[Coherence-SubInv2]** A core that accesses a translation (to perform a load or store) must have at least one token for that translation.

- **[Coherence-SubInv3]** A core that performs an MRF to a translation must have all T tokens for that translation before completing the MRF (*i.e.*, before releasing the page table lock - see Figure 3.1) and making the new translation visible.

This invariant can be interpreted as, conceptually, each MRF destroys a translation and creates a new one. All old tokens must be destroyed alongside the old translation, and a new set of tokens must be created for the new translation. The invariant ensures that there is a single point in time at which the old (pre-modified) translation is no longer visible to any cores.

The first two sub-invariants are almost identical to those of Token Coherence (TC). The third sub-invariant, which is analogous to TC's invariant that a core needs all tokens to perform a store, is subtly different from TC because an MRF is not an atomic write. In TC, a core must hold all tokens throughout the entire lifetime of the store, but an MRF only requires the core to hold all tokens before releasing the page table lock.

As with normal cache coherence, there are many ways to implement AT coherence such that it obeys these three sub-invariants. For example, instead of using explicit tokens, an AT system could use a snooping-like protocol with global invalidations or inter-processor interrupts for maintaining translation coherence. In our evaluation, we use a system that relies on inter-processor interrupts for maintaining translation coherence.

4.3 DVAT: Proposed Solution for Dynamic Verification of Address Translation

To check the correctness of the AT system at runtime, we propose DVAT, a mechanism that dynamically verifies the invariants described in our AT framework. In this section, we develop a first DVAT implementation that targets AT_{SC}. We refer to this implementation as $DVAT_{SC}$. When used with existing methods to dynamically verify $PAMC_{SC}$ [36, 87, 88], $DVAT_{SC}$ supports the dynamic verification of $VAMC_{SC}$

per Section 4.1.

4.3.1 System Model

Our baseline system is a cache-coherent multicore chip. Similar to most modern processors, each core uses virtually-indexed, physically-tagged caches. Physical caches ensure a store's atomicity with respect to loads from the same synonym set. Cores have hardware-managed TLBs, and updates to the status bits occur atomically in both the TLB and the page table when the corresponding load or store commits. The MRF procedure is slightly conservative and restricts parallelism. A core that performs an MRF locks the page table for the entire duration of the MRF, changes the PTE, triggers the inter-processor interrupt, waits for the acknowledgments from all other cores (instead of lazily collecting acknowledgments), and then signals the other cores that they may continue. All other cores flush their entire TLBs (instead of invalidating only affected translations), and spin after sending interrupt acknowledgments (instead of continuing immediately) until they receive the signal from the MRF initiator. In contrast, some current AT systems allow the other cores to continue their regular executions once they acknowledge the TLB flush.

We assume the existence of a checkpoint/recovery mechanism [98, 116] that can be invoked when DVAT_{SC} detects an error. The ability to recover to a pre-error checkpoint enables us to take DVAT_{SC}'s operations off the critical path; an error can be detected somewhat lazily as long as a pre-error checkpoint still exists at the time of detection.

4.3.2 DVAT_{SC} Overview

To dynamically verify AT_{SC}, we must dynamically verify both of its invariants: page table integrity and translation mapping coherence.

Checking Page Table Integrity

PT-SubInv1 is an invariant that is maintained by software. Fundamentally, there is no hardware solution that can completely check this invariant because the hardware does not have semantic knowledge of what the software is trying to achieve. Hardware could be developed to perform some "sanity checks", but, software checking is fundamentally required. One existing solution to this problem is self-checking code [21].

To check that PT-SubInv2 is maintained, we can adopt any of the previously proposed dynamic verification schemes for PAMC [36, 87, 88].

Checking Translation Coherence

The focus of DVAT_{SC} is the dynamic verification of the three translation coherence sub-invariants (Section 4.2.2). Because we have specified these sub-invariants in terms of tokens, we can dynamically verify the sub-invariants by adapting a scheme called TCSC [89] that was previously used to dynamically verify cache coherence. TCSC's key insight is that cache coherence states can be represented with token counts that can be periodically checked; this same insight applies to translation coherence. Even though the specification of coherence is in terms of tokens, the coherence protocol implementation is unrestricted; the protocol simply needs to maintain the invariants. For example, Martin *et al.* [84] showed that snooping and directory cache coherence protocols can be viewed as maintaining the token invariants. Thus, any DVAT solution, including DVAT_{SC}, are not architecturally visible, nor are they tied to any specific TLB coherence protocol.

Similar to TCSC, but for TLBs instead of normal caches, DVAT_{SC} adds explicit tokens to the AT system. Each translation has T tokens that are initially held by the translation's home memory and physically collocated with the translation's PTE. Because PTEs usually have some unused bits (*e.g.*, 3 for IA-32 and 4 for the Power

ISA), we can use these bits to store tokens. If we need more than the number of unused bits to hold T tokens, then we extend the memory block size to hold the extra bits. Because translations are dynamic and DVAT$_{SC}$ does not know *a priori* which blocks will hold PTEs, we must extend every memory block. A core that brings a translation into its TLB acquires one token corresponding to the PTE defining the translation. This token is held in the corresponding TLB entry, which requires us to slightly enlarge every TLB entry. The token is relinquished by the core and returned to memory once the translation is evicted from the TLB due to a replacement. In the case of a TLB invalidation, the token is sent to the core that requested the invalidation.

Each "node" in the system (*i.e.*, either a core/TLB or the memory) maintains a fixed-length signature of its token transfer history. This signature is a concise representation of the node's history of translation coherence events. Whenever a token is acquired or released, the signature is updated using a function that considers the physical address of the PTE to which the token corresponds and the logical time [72] of the transfer. Because extracting the translation mapping's virtual address from a TLB entry would require re-designing the TLB's CAM, the signature function operates on the PTE's physical address instead of its virtual-to-physical mapping. The PTE's physical address is a unique identifier for the translation. The challenge is that we now require that the SRAM portion of each TLB entry be expanded to hold the physical address of the PTE (this address does not need to be added to the page table PTEs). Thus, *signature*^{new} = *function (signature*^{old}*, PTE's physical address, logical time).*

In a correctly operating AT$_{SC}$ system, the exchanges of tokens will obey the three coherence sub-invariants of AT$_{SC}$ that we presented in Section 4.2.2. DVAT$_{SC}$ thus checks these three sub-invariants at runtime in the following fashion:

Coherence-SubInv1. Periodically, the signatures of all nodes are aggregated at one central verification unit that can check whether the conservation of tokens has been maintained. Updating signatures and checking them are off the critical path because we assume that we can recover to a pre-error checkpoint if an error is detected. The signature update function should be chosen so that it is easy to implement in hardware and avoids aliasing (*i.e.*, hashing two different token event histories to the same signature) as best as possible. We use the same function as TCSC [89] because it achieves these goals, but other functions could be chosen. Any basis of logical time can be used as long as it respects causality, and thus we use a simple one based on loosely synchronized physical clocks, similar to one used in prior work [116]. It is critical for DVAT_{SC} to consider the mapping (as represented by its PTE's physical address) and the time of the transfer in order to detect situations in which errors cause tokens to be sent for the wrong translations or tokens to be transferred at the wrong times.

Coherence-SubInv2. Checking this sub-invariant is straightforward. All that needs to be done is for each core to check that a token exists for a translation that it accesses in its TLB. This check can be performed in parallel with the TLB access and thus does not impact performance.

Coherence-SubInv3. Checking this sub-invariant is similar to checking Coherence-SubInv2. In parallel with completing an MRF for a translation, a core checks that it has all T tokens for that translation.

4.3.3 Implementation Details

DVAT_{SC} must address three challenges related to PTEs and token handling. The first issue is how to identify memory locations that contain PTEs. One simple option is to have the kernel mark pages that hold PTEs. Another option would be to monitor page table walks performed by the dedicated hardware; the first page table walk performed on a PTE marks the location accordingly and assigns it T tokens.

The second issue is determining where to send tokens when evicting a TLB entry to make room for a new translation (*i.e.*, not in response to an invalidation). With a typical TLB, we would not be able to identify the home node for an evicted translation. However, because we already hold the physical address of the PTE in each TLB entry for other purposes (as explained in Section 4.3.2), we can easily identify the translation's home node.

The third problem is related to which tokens need to be sent to the initiator of a full TLB flush. Many ISAs, such as the Power ISA, specify that the ability to invalidate specific translations is an optional feature for implementations, and thus implementations without this feature rely on full flushes of TLBs. As a consequence, a core that is requested to flush its TLB is unlikely to know which translations, if any, are actually being modified by the MRF that triggered the flush. One solution to this situation is for the core to send the tokens for all of its TLB entries to the initiator of the flush. The initiator keeps the tokens it wants (*i.e.*, tokens for the translations it is modifying) and forwards the rest of them to their home nodes. Considering the case of full TLB flushes rather than single translation invalidations maximizes DVAT_{SC}'s impact on systems's performance. Thus, our evaluation provides an upper bound for DVAT_{SC}'s performance impact.

If the AT system behaves safely (*i.e.*, does not behave incorrectly) but fails to make forward progress (*e.g.*, because a node refuses to invalidate a translation that is required by another node), then DVAT_{SC} will not detect this situation. Fortunately, timeout mechanisms are a simple approach for detecting liveness problems, and we have added such timeouts to our DVAT_{SC} implementation.

4.4 Evaluation

In this section, we evaluate DVAT_{SC}'s error detection ability, performance impact, and hardware cost.

Table 4.1: Target System Parameters for DVAT_{SC} Evaluation.

Parameter	Value
Cores	2, 4, 8, 16 in-order scalar cores
L1D/L1I	128KB, 4-way, 64B block, 1-cycle hit
L2 cache	4MB, 4-way, 64B block, 6-cycle hit
Memory	4GB, 160-cycle hit
TLBs	1 I-TLB and 1 D-TLB per core, all 4-way set-assoc.; 64 entries for 4K pages and 64 entries for 2/4MB pages
Coherence	MOSI snooping
Network	broadcast tree
DVAT_{SC} tokens	each PTE has T = 2C tokens
DVAT_{SC} signature	64 bits

4.4.1 Methodology

System Model and Simulator

Because AT involves system software, we use full-system simulation in our experiments. We use Simics [81] for functional simulation of an IA-32 multicore processor augmented with a TLB module (for controlling TLB behavior and fault injection) and GEMS [85] for timing simulation of the memory system. The operating system is Fedora Core 5 (kernel 2.6.15). Our target system, described in Table 4.1, is one particular implementation that satisfies the system model presented in Section 4.3.1. Because our target system conforms to the IA-32 architecture, TLB management and page walks are performed in hardware, and inter-processor interrupts are used to communicate translation invalidations. The interrupt handler at the invalidated node performs the invalidation.

Benchmarks

We evaluate DVAT_{SC} using several scientific benchmarks and one microbenchmark. The five scientific workloads, described briefly in Table 4.2, were developed as part of the Hood user-level threads library [22]. We wrote the microbenchmark specifically

Table 4.2: Scientific Benchmarks for DVAT$_{SC}$ Evaluation.

Benchmark	Description
knary	spawn tree of threads
mm	dense matrix multiplication
lu	LU factorization of dense matrix
msort	Merge-Sort of integers
barnes-hut	N-body simulation

to stress DVAT$_{SC}$'s error coverage, which is difficult to do with typical benchmarks. This microbenchmark has two threads that continuously map and remap a shared memory region, thus forcing translation coherence events to occur.

Error Injection

We inject faults into the AT system, many that correspond to published bugs [2, 3, 4, 59, 61, 62, 63], including: corrupted, lost, or erroneously delayed TLB coherence messages, TLB corruptions, TLB invalidations that are acknowledged but not applied properly (*e.g.*, flushes that do not flush all TLB entries), and errors in DVAT$_{SC}$ hardware itself. These fault injection experiments mimic the behavior of real processor bugs, since identically modeling these bugs is impossible for an academic study. Because our simulation infrastructure accurately models the orderings of translation accesses with respect to MRFs, we can accurately evaluate DVAT$_{SC}$'s error detection coverage.

4.4.2 Error Detection Ability

Prior work has already shown how to comprehensively detect errors in PAMC [36, 87, 88]. Thus we focus on the ability of DVAT$_{SC}$ to detect errors in AT$_{SC}$. We can evaluate its error coverage both empirically and analytically.

Empirical Evaluation. When DVAT$_{SC}$ is combined with PAMC verification (*e.g.*, TCSC) and timeouts, it detects errors that mimic published AT bugs. Figure

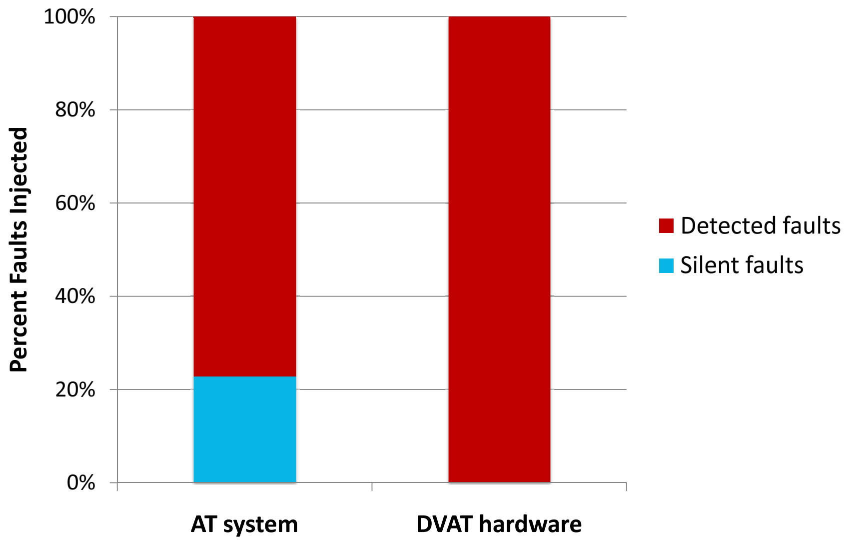


Figure 4.1: DVAT$_{SC}$'s Fault Detection Efficiency.

4.1 demonstrates how DVAT is efficient in detecting all injected faults in both the AT system and the DVAT hardware. For example, the four bugs in Table 1.1 are detected when they violate the following Coherence Sub-invariants, respectively: 1 or 2 (the bug violates both sub-invariants and will be detected by the checker for whichever sub-invariant it violates first), 1 or 2, 3, and 3. Some of the injected faults are masked, and do not result in erroneous execution. Consider the case when a core is not included in the MRF's translation coherence procedure (*i.e.*, corresponding interrupt is not delivered to the core). It is possible however that the excluded core does not contain a copy of the translation, and thus the MRF can successfully finish. In such cases, the fault is silent (*i.e.*, does not lead to an error).

Analytical Evaluation. Like TCSC, DVAT$_{SC}$ detects all single errors (and many multiple-error scenarios) that lead to violations of safety and that are not masked by signature aliasing. This error coverage was mathematically proved and experimentally confirmed for TCSC [89]. With a 64-bit signature size and a reasonable algorithm for computing signature updates, the probability of aliasing approaches 2^{64}. We have performed some fault injection experiments to corroborate

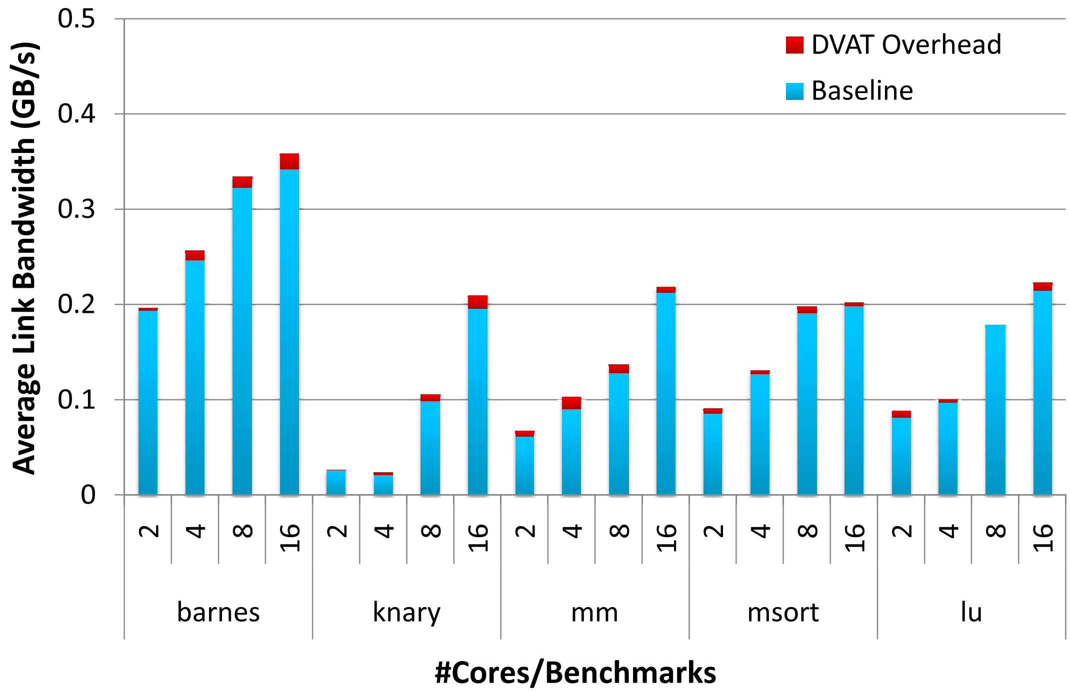


Figure 4.2: DVAT$_{SC}$'s Bandwidth Overhead Compared to Baseline System.

this result, but the number of experiments necessary to draw conclusions about such an extremely unlikely event is prohibitive.

4.4.3 Performance Impact

Checking PAMC has been shown to have little performance impact [36, 87, 88]. The rest of DVAT$_{SC}$'s actions are off the critical path because we use checkpoint/recovery to handle a detected error. DVAT$_{SC}$ can impact performance by increasing interconnection network congestion due to token exchanges, sending the physical address of a PTE along with the translation, and the periodic aggregation of signatures at a central verifier. With respect to checking the tokens before the end of the MRF procedure, there is enough slack in the Linux MRF procedure at the initiating core from starting the translation coherence events such that DVAT$_{SC}$ does not interfere with regular execution. We describe an MRF-independent method for ensuring that DVAT$_{SC}$ does not directly impact the regular execution flow in the future work sec-

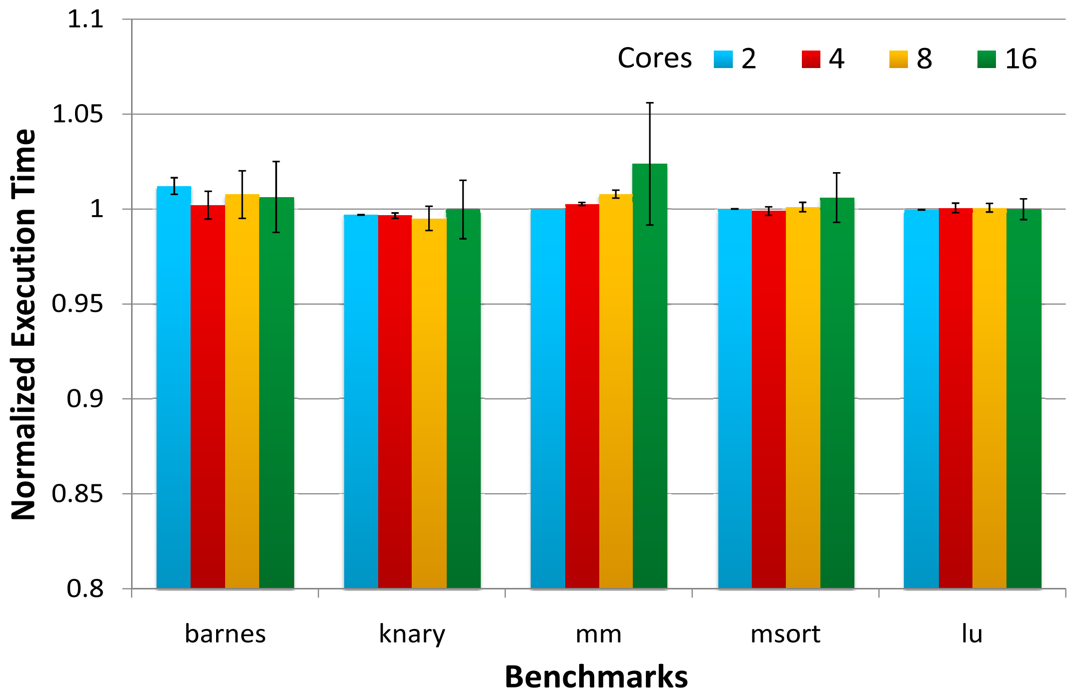


Figure 4.3: DVAT$_{SC}$'s Performance Impact. Results are normalized to baseline system. Error bars represent standard deviation.

tion (Section 4.6). DVAT$_{SC}$ aggregates and checks signatures at fixed intervals of logical time; in our experiments, we use an interval length of 10,000 snooping coherence transactions because this interval corresponds to our checkpointing interval.

In Figure 4.2, we plot the average link utilization in the interconnection network, both with and without DVAT$_{SC}$. For each benchmark data point, we plot the highest overhead observed across 100 runs that are perturbed to have slightly different timings to avoid underestimating utilization due to a particularly fortuitous timing. We observe that, for all benchmarks and all numbers of cores, the increase in utilization due to DVAT$_{SC}$ is small, below 2%.

The extra bandwidth consumption required by DVAT$_{SC}$ has a negligible impact on performance as shown in Figure 4.3. DVAT incurs a slowdown of less than 2.5% on average, with the most affected application being matrix multiply. Thus, DVAT provides error-coverage with minimal system intrusion.

4.4.4 Hardware Cost

DVAT_{SC} has five hardware costs: the hardware required to dynamically verify PAMC (shown in prior work [36, 87, 88] to be small), the storage for tokens, the extension to each TLB entry to hold the address of the PTE, the hardware to hold and update signatures (shown in TCSC [89] to be small), and the small amount of logic for checking the Coherence sub-invariants. The most significant hardware cost is the storage for tokens. For a system with C cores and 2 TLBs per core (I-TLB and D-TLB), DVAT_{SC} adds 2C tokens to each PTE, thus requiring $\log_2 2C$ bits. For systems with few cores, these bits are likely to fit in the unused bits of the PTE. For systems with many cores, one way to reduce the token storage cost is to extend the coherence sub-invariants to the coarser granularity of a memory block (instead of a PTE), *i.e.*, associate T tokens with a memory block. For a 128-core system with 8 PTEs per memory block, we can keep the storage cost to only 11 bits per block (minus those bits that can be fit into unused PTE bits). The overhead is thus only 4.3% and 2.1% for 32 and 64 byte blocks, respectively. As with any error detection mechanism, DVAT_{SC} benefits from the existence of a checkpoint/recovery mechanism [98, 116] to recover from detected errors. The cost of checkpoint/recovery depends on the specific implementation and is decoupled from the DVAT cost.

4.5 Related Work

We discuss prior work in specifying and dynamically verifying correctness, as well as ad-hoc detection of design bugs.

We categorize this prior work based on which part of the system it considers.

Memory Systems. Meixner and Sorin [87, 88] and Chen *et al.* [36] dynamically verified AT-oblivious memory consistency models. These schemes apply directly to PAMC, and they can be applied to VAMC if one assumes a one-to-one mapping from VA to PA (*i.e.*, no synonyms). Similarly, Chen *et al.* [37] dynamically verified the consistency of AT-oblivious transactional memory systems. Cain and Lipasti also developed algorithms for checking AT-oblivious memory consistency [29], but they did not pursue a full implementation. Other work has developed checkers for AT-oblivious cache coherence, which is a necessary sub-invariant of AT-oblivious memory consistency [30, 89]. Our work differs from this prior work by considering address translation.

Processor Cores. The ISA specifies the correct behavior of the processor core, including the exact semantics of every instruction, exception, interrupt, *etc.* The first dynamic verification scheme for processor cores is DIVA [16]. The insight behind DIVA is that we can check a complicated, superscalar core with a simple, statically verifiable core that has the same ISA. The checker core is so simple that its design can be statically verified (*e.g.*, using a model checker), and thus it detects all design bugs in the superscalar core. Another approach to specification and verification is Argus [86]. Argus is based on the observation that a core's behavior can be verified by checking the correctness of three tasks: control flow, dataflow, and computation. The Argus-1 implementation uses checkers for each of these tasks to dynamically verify the core. Other work by Reddy and Rotenberg [101] has specified microarchitectural invariants that can be dynamically verified. These invariants are necessary but not sufficient for correctness (as defined by the ISA). Our work differs from Reddy and Rotenberg by considering architectural correctness.

Ad-Hoc Bug Detection. Rather than formally specify correctness and then dynamically verify it, another option is for the system to look for known buggy states or anomalies that might indicate that a bug has been exercised. Wagner *et al.* [127] use a pattern matching technique to detect when the system is in a known buggy state. Work by Narayanasamy *et al.* [94] and Sarangi *et al.* [109] proposes to detect design bugs by monitoring a certain subset of processor signals for potential anomalies. If a bug is detected, the authors propose patching it with a piece of programmable hardware. Li *et al.* [79] take a similar approach to detecting errors (due to physical faults, but the same approach applies to hardware design bugs), but instead of observing hardware anomalies they detect anomalies at the software level. Our work differs from this work in anomaly detection by formally specifying correctness and dynamically verifying that specification, rather than observing an ad-hoc set of signals.

4.6 Conclusions and Future Work

This chapter proposed an AT dynamic verification method that can, at runtime, detect errors due to design bugs and physical faults, including AT-related design bugs we identified in processors errata. We demonstrated the scheme's efficiency in detecting AT errors, and its low impact of application performance. In addition, we proved that for a specific AT model, this method can be used in conjunction with PAMC verification to guarantee VAMC correctness.

An interesting future direction of research is to further analyze the connection between PAMC, AT, and VAMC models. The AT framework we proposed in this chapter satisfies most current AT models. However, a formal proof is required to demonstrate more generally that just AT correctness is sufficient for a correct PAMC to guarantee a correct VAMC. Nevertheless, understanding the complex interactions between PAMC and AT is crucial for designing future virtual memory based systems.

DVAT$_{SC}$ represents an initial exploration of the DVAT solutions. There are several aspects that can be considered for extending the current implementation to cover more relaxed AT systems. The most important constraint that current systems relax is the requirement that all cores wait for the MRF to finish, even if they acknowledge the translation coherence events. DVAT$_{SC}$ can be extended to support such systems by relying on two sets of *logical* tokens: an "old" set that is gathered by the core triggering the MRF and corresponds to the old translation, and a "new" set that corresponds to the new translation. These logical tokens can be supported by simply extending the token holding locations with an additional bit that indicates the token's type. A core that releases the old token is allowed to acquire a new token, such that it can access the new translation as soon as it is created.

Finally, another research avenue is represented by the incorporation of the Page Table Integrity invariants in the DVAT mechanism. As specified in Section 4.3.2, hardware can support checking these invariants only with additional information provided by software. A possible solution is for the software to embed "sanity checks" in the page table translation's when translations are created. These properties can be later checked by the hardware during page table accesses (*i.e.*, page table walks), and thus provide guarantees about the integrity of the page table.

5

Unified Instruction, Data and Translation Coherence Protocol

Current systems rely on different protocols for maintaining coherence of translation caches, and instruction and data caches, respectively. Thus, systems rely on software procedures for maintaining translation coherence, while instruction/data coherence is invariably maintained by a hardware-based protocol. Unfortunately, the TLB shootdown routine, the software procedure for enforcing translation coherence, is performance costly and non-scalable [44, 75, 121].

In this chapter, we propose UNified Instruction/Translation/Data (UNITD) Coherence, a hardware coherence framework that integrates translation coherence into the existing cache coherence protocol. In UNITD coherence, the TLBs participate in the cache coherence protocol just like instruction and data caches. UNITD is more general than the only prior work in hardware TLB coherence [126], which requires specific assumptions about allowable translation caching (*e.g.*, copy-on-write is disallowed).

This chapter is organized as follows. Section 5.1 discusses translation coherence, by focusing on TLB shootdown (Section 5.1.1), the procedure generally used for maintaining translation coherence, and its impact on application runtime (Section 5.1.2). We describe the UNITD coherence protocol in Section 5.2. In Section 5.3, we discuss implementation issues, including platform-specific aspects and optimizations. In Section 5.4, we evaluate snooping and directory-based UNITD coherence protocols on multicore processors and show that UNITD reduces the performance penalty associated with TLB coherence to almost zero, performing nearly identically to a system with zero-latency TLB invalidations. We discuss related work in Section 5.6 and conclude in Section 5.7.

5.1 Existing Solutions for Maintaining Address Translation Coherence

Maintaining coherence between the TLBs and the page tables has historically been named "TLB consistency" [126], but we will refer to it as "TLB coherence" due to its much closer analogy to cache coherence than to memory consistency.

One important difference between cache coherence and TLB coherence is that some systems do not require maintaining TLB coherence for each datum (*i.e.*, TLBs may contain different values for the same translation). Such incoherence is allowed with respect to permission and status bits, but never for the mapping. Thus, these architectures require TLB coherence only for unsafe changes [125] made to address translations. Unsafe changes include mapping modifications, decreasing the page privileges (*e.g.*, from read-write to read-only), and marking the translation as invalid. The remaining possible changes (*e.g.*, increasing page privileges, updating the Accessed/Dirty bits) are considered to be safe and do not require TLB coherence. Consider one core that has a translation marked as read-only in the TLB, while a second core updates the translation in the page table to be read-write. This translation update does not have to be immediately visible to the first core. Instead, the first core's TLB data can be lazily updated if the core executes a store instruction.

Initiator	Victim
• disable preemption and acquire page table lock	
• construct list of victim processors	
• construct list of translation(s) to invalidate	
• flush translation(s) in local TLB	
• if (victim list not empty) send interrupts to victims	
	• service interrupt & get list of translation(s) to invalidate
• while (victim list not empty) wait;	• invalidate translation(s) from TLB
	• acknowledge interrupt & remove self from victim list
• release page table lock and enable preemption	

Figure 5.1: TLB Shootdown Routines for Initiator and Victim Processors.

The execution of the store leads to either an access violation (*i.e.*, page fault) or an attempt to update the translation as read-write. In either case, the second core detects that the page table translation has already been marked accordingly and updates the TLB cached copy.

Systems usually enforce translation coherence through TLB shootdowns, a procedure that we discuss in depth in Section 5.1.1. However, there are some architectures that rely on alternative mechanisms, and we discuss these in the related work section (Section 5.6).

5.1.1 TLB Shootdown

TLB shootdown [19, 35, 107] is a software routine for enforcing TLB coherence that relies on inter-processor interrupts (considering the present multicore processors, the procedure is more precisely an inter-core interrupt; for consistency, we use "processor" instead of "core" when referring to this type of interrupts) and has the generic structure presented in Figure 5.1. The shootdown is triggered by one processor (*i.e.*, initiator) that programs an inter-processor interrupt for all other processors sharing the same address space (*i.e.*, victims). In the interrupt handler, these processors invalidate the translation(s) from their TLBs. Because managing the address translation system is the responsibility of privileged software, TLB shootdowns are invisible to the user application, although shootdowns directly impact the user application's performance. This performance impact depends on several factors, including the position of the TLB in the memory hierarchy, the shootdown algorithm used, and the number of processors affected by the shootdown (victim processors). We discuss the first two factors in this section, and we analyze the impact of the number of victim processors on the TLB shootdown cost in Section 5.1.2.

TLB position. TLBs can be placed at different levels of the memory system between the core and the physical memory [99]. Most microarchitectures implement per-core TLBs associated with virtually-indexed physically-tagged caches, as this implementations simplifies the cache management (*i.e.*, it eliminates the need to address synonyms, as discussed in Section 3.4.1). These designs, however, pose scalability problems for many-core systems because the performance penalty for the shootdown initiator increases with the number of victim processors, as we show in Section 5.1.2. The initiator must wait for more cores to acknowledge the interrupt, while the victims contend for updating the variable defining the cores who acknowledged the interrupt. Because this solution is most common, we also assume per-core TLBs in this chapter. Another option is to position the TLB at the memory [126], such that a translation occurs only when a memory access is required. This design might appear attractive for many-core chips, since TLB coherence must be ensured only at memory controllers, whereas cache coherence is ensured using virtual addresses. However, virtual caches suffer from the well-known problem of virtual synonyms [32, 33].

Shootdown algorithm. The TLB shootdown procedure can be implemented using various algorithms that trade complexity for performance. Teller's study [125] is an excellent description of various shootdown algorithms. In this chapter, we assume the TLB shootdown routine implemented in Linux kernel 2.6.15, which follows the generic structure described in Figure 5.1. The procedure leverages Rosenburg's observation that a shootdown victim can resume its activity as soon as it has acknowledged the shootdown (*i.e.*, has removed itself from the shootdown list) [107]. The algorithm thus reduces the time spent by victims in the shootdown interrupt.

5.1.2 Performance Impact of TLB Shootdown

In this section, we analyze the extent to which TLB coherence affects the performance of an application in current systems. This impact depends on two factors: the penalty associated with TLB shootdown routines as dictated by the OS and supporting hardware, and the frequency that these routines are utilized by the application, respectively. The former is platform-dependent while the latter is application-dependent.

We perform these experiments on a real machine consisting of 32-Xeon processors with 64GB RAM running Suse Enterprise Linux Server Edition 10 (kernel 2.6.15). We study systems with fewer cores by disabling cores in the system such that the functional cores are the most closely located (*i.e.*, physically) cores in the machine.

Figure 5.2 shows the latency of a single TLB shootdown for both the initiator and victims as a function of the number of processors involved in the shootdown. We measure the latency by instrumenting the kernel such that we read the processor's timestamp counter at the beginning and end of the shootdown routines. This allows us to determine the latency of the operations with minimal system intrusion.

The latency of a shootdown is application-independent and is determined by the microarchitectural characteristics, the number of processors involved, and the OS. Figure 5.2 shows that the latency of a shootdown increases roughly linearly with the number of processors involved for both the initiator as well as the victim cores. This latency does not capture the side effects of TLB shootdowns such as the

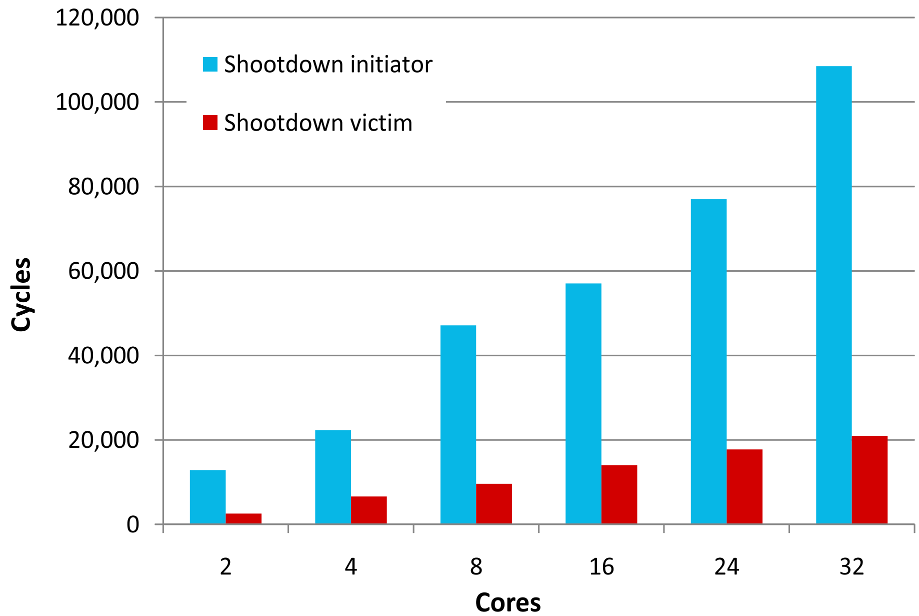


Figure 5.2: Average TLB Shootdown Latency on Xeon Processors/Linux Platform.

TLB invalidations that result in extra cycles spent in repopulating the TLB with translations after the shootdown. This additional cost depends on the applications's memory footprint, as well as the position of the corresponding cache blocks in the memory hierarchy. For an Intel 64 architecture, filling a translation in the TLB requires two L1 cache accesses in the best-case scenario; the worst-case scenario requires four main memory accesses. On x86/Linux platforms, this additional cost is sometimes increased by the fact that, during shootdowns triggered by certain events, the OS forces both the initiator and the victims to flush their entire TLBs rather than invalidate individual translations.

The experiment reveals that as the number of cores increases, maintaining TLB coherence is likely to have an increasingly significant impact on performance if it is enforced through the current TLB shootdown routine. To alleviate this performance impact, architects need to either change the way pages are shared across threads or change the mechanism for maintaining TLB coherence. The solution that we propose in this chapter is the latter, by maintaining TLB coherence in hardware.

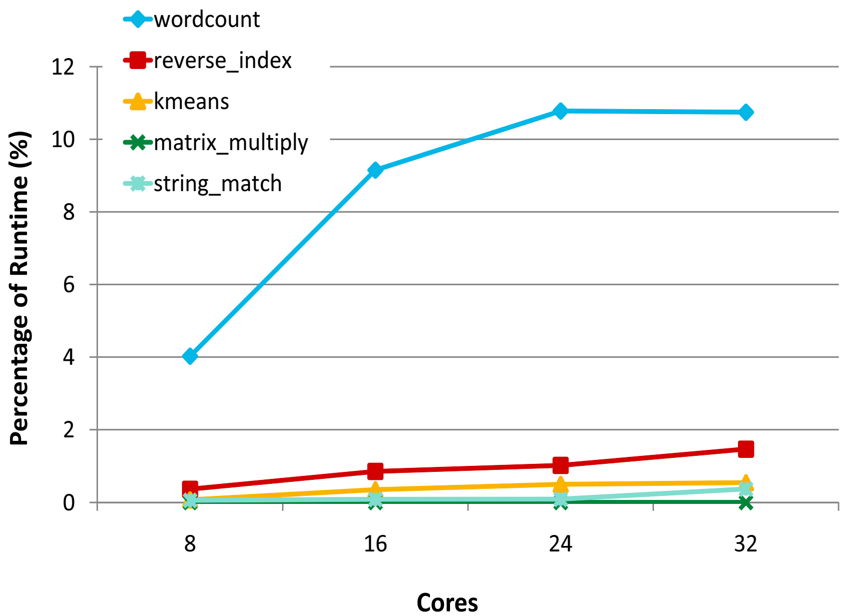


Figure 5.3: TLB Shootdown Performance Overhead on Phoenix Benchmarks.

Our second experiment analyzes the impact of TLB shootdowns on real applications. For this study, we choose several benchmarks from the Phoenix suite [100] that cover a wide range in terms of the number of TLB shootdowns incurred within a given amount of application code. We use Oprofile [78] to estimate the percent of total runtime spent by the applications in TLB shootdowns. We consider this number to be the percent of the total Oprofile samples that are reported to be taken within either the shootdown initiator or victim routines. Figure 5.3 shows the fraction of total runtime associated with the TLB shootdowns, which becomes significant for applications that require translation coherence more often. It is also important to observe that there are applications, such as matrix multiply, that do not make changes to the page tables and thus do not exercise TLB shootdowns. Nevertheless, there is a class of applications, such as wordcount and the software mentioned in Section 1.3, that rely heavily on the shootdowns and for which these routines can represent a major fraction of the total runtime. Considering these large variations in the usage patterns of TLB shootdowns across applications, we evaluate UNITD across a wide range of shootdown frequencies (Section 5.4).

5.2 UNITD Coherence

In this section, we introduce the framework for unifying TLB coherence with cache coherence in one hardware protocol, as well as describing the details of UNITD, the proposed unified protocol. At a high level, UNITD integrates the TLBs into the existing cache coherence protocol that uses a subset of the typical MOESI coherence states (we assume a MOSI coherence protocol in our UNITD implementations; we discuss in Section 5.3.3 how to extend UNITD to protocols that implement the Exclusive state). Fundamentally, TLBs are additional caches that participate in the coherence protocol like coherent, read-only instruction caches. In the current implementation, UNITD has no impact on the cache coherence protocol and thus does not increase its complexity. In addition, we design UNITD to be easily integrated with existing microarchitectural components.

With respect to the coherence protocol, TLBs are read-only caches similar to the instruction caches: TLB entries (*i.e.*, translations) are never modified in the TLBs themselves. Thus, only two coherence states are possible: Shared (read-only) and Invalid. When a translation is inserted into a TLB, it is marked as Shared. The cached translation can be accessed by the local core as long as it is in the Shared state. The translation remains in this state until either the TLB receives a coherence message invalidating the translation, or the translation is invalidated through a coherence-independent mechanism (*e.g.*, the execution of a specific instruction that invalidates translations such as *invlpg* for Intel 64 ISA or the replacement of the translation). The translation is then Invalid and thus subsequent memory accesses depending on it will miss in the TLB and reacquire the translation from the memory system. Given that a translation is valid for core accesses while in the Shared state, UNITD uses the existing Valid bit of the cached translation to maintain a TLB entry's coherence state. This Valid bit is specific to the translation cached by the TLB and is independent of the Valid bit for the translation present in the memory page tables, which restricts TLBs from accessing and caching the respective translation if the bit is not set.

Despite the similarities between TLBs and instruction and data caches, there is one key difference between caches and TLBs: cache coherence is based on physical addresses of data, but a datum cached in a TLB (*i.e.*, a translation) is not directly addressable by the physical addresses on which it resides (*i.e.*, the physical address of the PTE defining the translation, not to be confused with the physical address to which the translation maps a virtual address). This is a consequence of current implementations that rely on the TLB being content-addressable and not address-accessible. For the TLBs to participate in the coherence protocol, UNITD must be able to perform coherence lookups in the TLB based on the physical addresses of PTEs. The association between the PTE address and the translation provides a unique physical address for each translation, as each translation is uniquely defined by a translation (Section 5.2.1 discusses the case when a translation is defined by multiple PTEs). To overcome this key difference between TLBs and caches, we must address two issues:

Issue 1: For each translation in a TLB, UNITD must discover the physical address of the PTE associated with that translation at runtime.

Issue 2: UNITD must augment the TLBs such that they can be accessed with a physical address.

We discuss UNITD's solutions to these two issues in the following two subsections.

5.2.1 Issue 1: Discovering the Physical Address of a Translation's PTE

We start by describing the concept behind discovering the PTE associated with a translation, followed by a description of how to determine the physical address of the

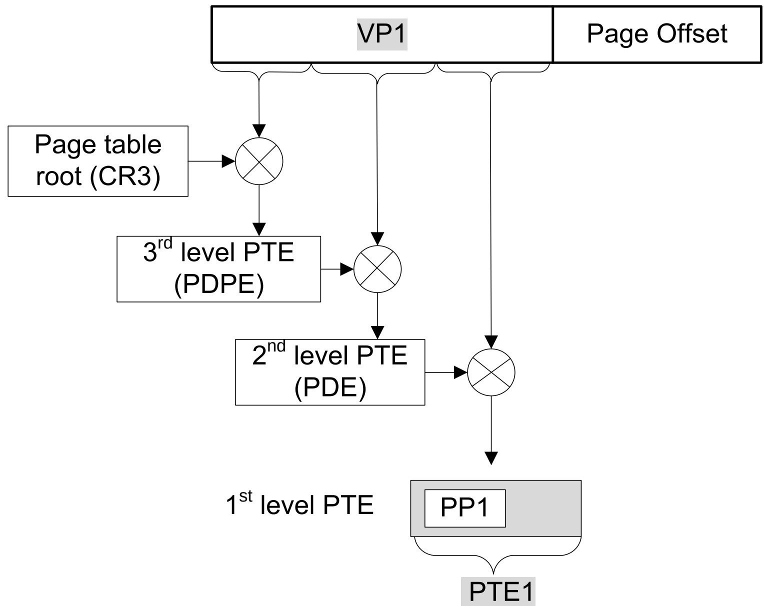


Figure 5.4: 3-level Page Table Walk in IA-32. UNITD associates PTE1 with the VP1→PP1 translation.

PTE in practice.

Concept. The issue of associating a translation with its PTE's physical address assumes there is a one-to-one association between translations and PTEs. This assumption is straightforward in systems with flat page tables, but less obvious for systems using hierarchical page tables.

For architectures that implement hierarchical page tables, a translation is defined by a combination of multiple PTEs in the hierarchy. Figure 5.4 illustrates the translation, on an IA-32 system, from virtual page VP1 to physical page PP1, starting from the root of the page table (*i.e.*, CR3 register) and traversing the intermediate PTEs (*i.e.*, PDPE and PDE). Conceptually, for these architectures, translation coherence should be enforced when a modification is made to *any* of the PTEs on which the translation depends. Nevertheless, we can exploit the hierarchical structure of the page tables to relax this constraint to a single-PTE dependency by requiring that any change to a PTE propagates to a change of the last-level PTE. Thus, a translation is identifiable through the last-level PTE address, and we thus guarantee a unique translation-physical address assignment.

To understand why such an assumption is justifiable, consider the case of a modification to an intermediary PTE. PTE modifications can be divided into changes to mappings and changes to the metadata bits. In the case of mapping changes, the previous memory range the PTE was mapping to must be invalidated. Moreover, for security reasons, the pages included in this space must be cleared such that whenever this memory space is reused, it does not contain any previous information. With respect to the metadata bits, any unsafe changes (*i.e.*, to the permission bits) must be propagated down to the last-level PTE. In both cases, we can identify when translation coherence is required by determining when changes are made to the last-level PTE that the translation depends on.

Therefore, independent of the structure of the page tables, a translation is identifiable through the last-level PTE address. Of course, this requires the identification of the last-level PTEs associated with each translation.

Implementation. How the last-level PTE's physical address is identified depends on whether the architecture assumes hardware or software management of TLB fills and evictions. Designs with hardware-managed TLBs rely on dedicated hardware ("page table walker") that walks iteratively through the page table levels in case of a TLB miss. The number of iterative steps in a walk depends on the architecture (*i.e.*, structure of the page table) and the values stored at each level's PTE. As a consequence, the walker knows when it is accessing the last-level PTE and can provide its physical address to the TLB (*i.e.*, this is the address from where the state machine will read the physical address of the translation's mapping).

For architectures with software-managed TLB fills/evictions, UNITD requires software support for notifying the hardware as to the last-level PTE associated with a translation. The software can easily identify the PTE since the software follows the same algorithm as the hardware walker. Once the PTE address is found, it can be written to a dedicated memory address such that the hardware associates it with the translation that will be inserted in the TLB. An alternative solution for systems with software-managed TLBs is for the software to explicitly insert this physical address in the TLB through a dedicated instruction. Because our evaluation targets an x86 system with hardware management of TLB fills/evictions, in our analysis we assume a system with hardware-managed TLBs, but UNITD is equally applicable to systems with software-managed TLBs.

5.2.2 Issue 2: Augmenting the TLBs to Enable Access Using a PTE's Physical Address

Concept. To perform coherence lookups in the TLBs, UNITD needs to be able to access the TLBs with physical addresses and invalidate the translations associated with the PTEs that reside at those physical addresses, if any. In this discussion, we assume a one-to-one correspondence between translations and PTEs as discussed in the previous subsection. Thus, a TLB translation moves to the Invalid state whenever the core receives a coherence invalidation request for the translation (*i.e.*, PTE defining the translation is modified).

Implementation. To render the TLB accessible by physical address, we record the physical addresses of PTEs associated with the translations cached by the TLB. As these addresses must be stored as long as the translations are present in the TLB, we associate with each TLB an additional hardware structure. We refer to this structure that intermediates between TLBs and the coherence protocol as the *Page Table Entry CAM* (*PCAM*). The PCAM has the same number of entries as the TLB, and it is fully-associative because the location of a PTE within a set-associative TLB is determined by the TLB insertion algorithm and not by the PTE's physical address.

Figure 5.5 shows how the PCAM is integrated into the system, with interfaces

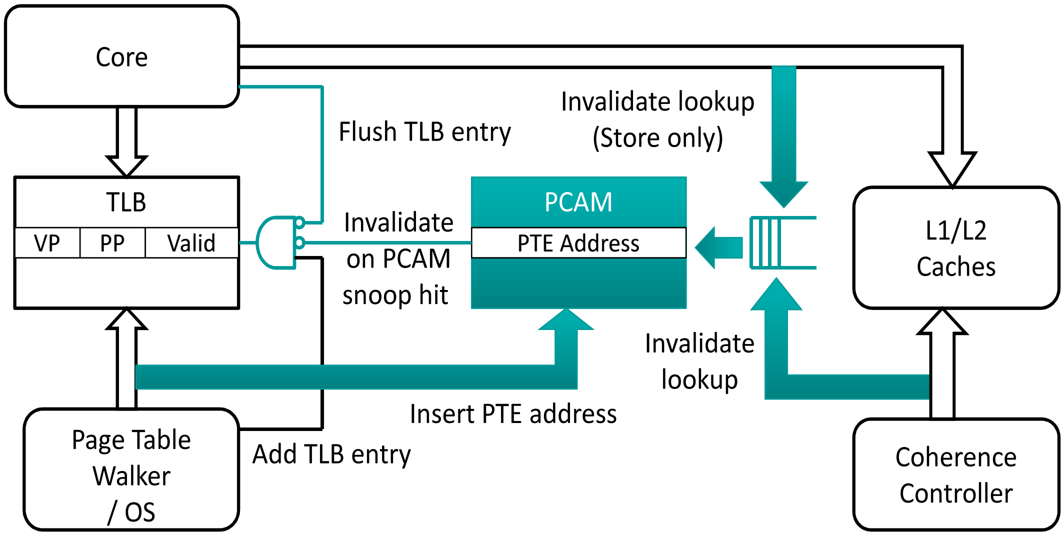


Figure 5.5: PCAM's Integration with Core and Coherence Controller. UNITD introduced structures are colored.

to the TLB insertion/eviction mechanism (for inserting/evicting the corresponding PCAM entries), the coherence controller (for receiving coherence invalidations), and the core (for a coherence issue discussed in Section 5.3.2). The PCAM is off the critical path of a memory access; it is not accessed during regular TLB lookups for obtaining translations, but only at TLB insertions and coherence invalidation lookups.

The PCAM is logically a content addressable memory and could be implemented with a physical CAM. For small PCAMs, a physical CAM implementation is practical. However, for PCAMs with large numbers of entries (*e.g.*, for use with a 512-entry 2nd-level TLB), a physical CAM may be impractical due to area and power constraints. In such situations, the PCAM could be implemented with a hardware data structure that uses pointers to connect TLB entries to PCAM entries. Such a structure would be similar to the indirect index cache [47], for example. Henceforth, we assume a physical CAM implementation, without loss of generality.

Maintaining coherence on physical addresses of PTEs requires bookkeeping at a fine granularity (*e.g.*, double-word for a 32-bit architecture). In order to integrate

TLB (before)

VP	PP	Valid
VP3	PP1	1
VP2	PP6	1
VP6	PP0	0
VP5	PP4	0

PCAM (before)

PA
12
134
30
76

Insert translation VP1→PP9 which is at PA 12 $\Longrightarrow$

TLB (after)

VP	PP	Valid
VP3	PP1	1
VP2	PP6	1
VP1	PP9	1
VP5	PP4	0

PCAM (after)

PA
12
134
12
76

(a) Inserting an entry into the PCAM when a translation is inserted into the TLB

TLB (before)

VP	PP	Valid
VP3	PP1	1
VP2	PP6	1
VP1	PP9	1
VP5	PP4	0

PCAM (before)

PA
12
134
12
76

Process coherence invalidation for PA 12 $\Longrightarrow$

TLB (after)

VP	PP	Valid
VP3	PP1	0
VP2	PP6	1
VP1	PP9	0
VP5	PP4	0

PCAM (after)

PA
12
134
12
76

(b) Processing a coherence invalidation for a physical address (two PTEs reside at the corresponding block address)

Figure 5.6: PCAM Operations. PA represents physical address.

TLB coherence with the existing cache coherence protocol with minimal microarchitectural changes, we relax the correspondence of the translations to the memory block containing the PTE rather than the PTE itself. Maintaining translation granularity at a coarser grain (*i.e.*, cache block, rather than PTE) trades a small performance penalty for ease of integration. This performance penalty depends entirely on the application's pattern of modifying translations. Because multiple PTEs can be placed in the same cache block, the PCAM can hold multiple copies of the same datum. For simplicity, we refer to PCAM entries simply as PTE addresses. A coherence invalidation request for the same block address leads to the invalidation of all matching translations. A possible solution for avoiding false-invalidations is extending UNITD to a sub-block coherence protocol for translations only, as previously proposed for regular cache coherence [38].

Figure 5.6 shows the two operations associated with the PCAM: (a) inserting an entry into the PCAM and (b) performing a coherence invalidation at the PCAM.

PTE addresses are added in the PCAM simultaneously with the insertion of their corresponding translations in the TLB. Because the PCAM has the same structure as the TLB, a PTE address is inserted in the PCAM at the same index as its corresponding translation in the TLB (physical address 12 in Figure 5.6(a)). Note that there can be multiple PCAM entries with the same physical address, as in Figure 5.6(a). This situation occurs when multiple cached translations correspond to PTEs residing in the same cache block.

PCAM entries are removed as a result of the replacement of the corresponding translation in the TLB or due to an incoming coherence request for read-write access. If a coherence request hits in the PCAM, the Valid bit for the corresponding TLB entry is cleared. If multiple TLB translations have the same PTE block address, a PCAM lookup on this block address results in the identification of all associated TLB entries. Figure 5.6(b) illustrates a coherence invalidation of physical address 12 that hits in two PCAM entries.

5.3 Platform-Specific Issues, Implementation Issues, and Optimizations

In this section, we discuss several implementation issues that target both functional and performance aspects of UNITD, including: the integration with speculative execution in superscalar cores (Section 5.3.1), the handling of translations that are currently in both the TLB and data cache of a given core (Section 5.3.2), UNITD's compatibility with a wide range of system models and features (Section 5.3.3), and a method of reducing the number of TLB coherence lookups (Section 5.3.4).

5.3.1 Interactions with Speculative Execution

UNITD must take into account the particularities of the core, especially for superscalar cores. Many cores speculatively execute a load as soon as the load's address is known. In a multithreaded or multicore environment, it is possible for another thread to write to this address between when the load speculatively executes and when it becomes ready to commit. In an architecture that enforces sequential consistency (*i.e.*, obeys a sequentially consistent VAMC model), these situations require that the load (and its consumers) be squashed. To detect these mis-speculations, cores adopt one of two solutions [46]: either snoop coherence requests that invalidate the load's address or replay the load at commit time and compare the replayed value to the original.

With UNITD, an analogous situation for translations is now possible. A load can read a translation from the TLB before it is ready to commit. Between when the load reads the translation and is ready to commit, the translation could be invalidated by a hardware coherence request. This analogous situation has analogous solutions: either snoop coherence requests that invalidate the load's translation or replay the load's TLB access at commit time. Either solution is more efficient than the case for systems without UNITD; in such systems, an invalidation of a translation causes an interrupt and a flush of the entire pipeline.

5.3.2 Handling PTEs in Data Cache and TLB

UNITD must consider the interactions between TLBs and the core when a page table walk results in a hit on a block present in the Modified state in the local core's data cache. This scenario requires special consideration because it leads to data being present in apparently incompatible coherence states in both the data cache and the TLB. Consider the following example, when the data cache contains an exclusive copy of the translation in Modified state, and the core performs a page table walk on the translation. This will lead to the data cache block remaining in Modified, while also being present in Shared in the TLB. A subsequent write by the core might find the data block in Modified and perform a translation change without triggering any coherence invalidations. Thus, the TLB will contain an invalid copy of the translation.

We present three viable solutions to this situation.

Solution #1. Because the page table walk results in the TLB having this block Shared, we can maintain the coherence invariant of "single writer or multiple readers" (SWMR) by having the block in the core's data cache transition from Modified to Shared. The drawback of this solution is that, because the page table walker uses the core's regular load/store ports to insert requests into the memory system, the cache controller must distinguish between memory accesses of the same type (*e.g.*, loads) originating from the core's pipeline. For example, a regular (non-page-table-walk) load leaves the data cache block in the Modified state, whereas a page-table-walk load transitions the data cache block to Shared.

Solution #2. We can introduce an additional coherence state for cache blocks: Modified-TLBCached. A block transitions to this state from Modified following a page table walk. As long as the block remains in this state, a copy of the translation it contains *might* be cached in the TLB (it is possible that the TLB evicted the translation since the access). Consequently, a store on a data block in this state requires a local TLB coherence invalidation. The main disadvantage of this solution is that it modifies the original cache coherence protocol, although it minimizes the required TLB invalidation accesses.

Solution #3. Because Solutions #1 and #2 require changing the coherence controller, we instead adopt an alternative solution that does not affect the cache coherence protocol. If a page table walk results in a hit on a block in the Modified state in the data cache, we leave the block in the Modified state in the data cache, while inserting the block in the Shared state in the TLB. Despite the apparent violation of the SWMR invariant, UNITD ensures that the TLB always contains coherent data by probing the TLB on stores by the local core. This situation is the only case in which UNITD allows a combination of seemingly incompatible coherence states. Because cores already provide mechanisms for self-snoops on stores for supporting self-modifying code [64], UNITD can take advantage of existing resources, which is why we have chosen Solution #3 over the other two in our UNITD implementations.

5.3.3 UNITD's Non-Impact on the System

UNITD is compatible with a wide range of system models, and we now discuss some system features that might appear to be affected by UNITD.

Cache Coherence Protocol

We have studied UNITD in the context of systems with both MOSI snooping and directory coherence protocols. UNITD has no impact on either snooping or directory protocols, and it can accommodate a MOESI protocol without changing the coherence protocol.

Snooping. By adopting the self-snooping solution previously mentioned in Section 5.3.2, no change is required to the cache protocol for a snooping system.

Directory. It might appear that adding TLBs as possible sharers of blocks would require a minor change to the directory protocol in order to maintain an accurate list of block sharers at the directory. However, this issue has already been solved for coherent instruction caches. If a core relinquishes ownership of a block in its data cache due to an eviction and the block is also present in its instruction cache or TLB, it sets a bit in the writeback request such that the directory does not remove the core from the block's list of sharers. Also, the coherence controller must be enhanced such that it allows invalidation acknowledgments to be sent if the address is found in the PCAM.

MOESI Protocols

UNITD also applies to protocols with an Exclusive state (*i.e.*, MOESI protocol) without modifying the protocol. For MOESI protocols, the TLBs must be integrated into the coherence protocol to determine if a requestor can obtain a block in the Exclusive state. Once again, the TLB behaves like a coherent instruction cache; it is probed in parallel with the cores' caches and contributes to the reply sent to the requestor.

Memory Consistency Model

UNITD is applicable to any memory consistency model. Because UNITD's TLB lookups are performed in parallel with cache snoops, remote TLB invalidations can be guaranteed through the mechanisms provided by the microarchitecture to enforce global visibility of a memory access, given the consistency model.

Virtual Address Synonyms

UNITD is not affected by synonyms because it operates on PTEs that uniquely define translations of virtual addresses to physical addresses. Each synonym is defined by a different PTE, and changing/removing a translation has no impact on other translations corresponding to virtual addresses in the same synonym set.

Superpages

Superpages rely on "coalescing neighboring PTEs into superpage mappings if they are compatible" [124]. The continuity of PTEs in physical addresses makes TLB snooping on superpages trivial with simple UNITD extensions (*e.g.*, the PCAM can include the number of PTEs defining the superpage to determine if a snoop hits on any of them).

Virtual Machines

Virtualization does not affect UNITD. UNITD operates on PTEs using physical addresses, and not machine addresses. A PTE change will affect only the host for which the PTE defines a translation. If multiple VMs access a shared physical page, they will access it using their own physical PTEs, as assigned by the host OS. In fact, we expect UNITD performance benefits to increase on virtualized systems because the TLB shootdown cost (which is eliminated by UNITD) increases due to host-guest communication for setting up the procedure.

Status Bits Updates

As discussed in Section 5.1, some systems do not require translation coherence for safe changes. In the current implementation, UNITD does not distinguish between safe and unsafe changes and enforces coherence on all translation updates. In theory, this can adversely impact the application, as the UNITD system will incur additional TLB translations invalidations compared to the system relying on TLB shootdowns. In reality, the impact of treating all translation updates as unsafe depends on the application's behavior.

Consider the case of the update of a translation's Dirty bit by Core 1, where Core 2 has the translation cached as read-only. On the translation update, the UNITD system invalidates the translation cached by Core 2. Thus, Core 2 incurs a page table walk penalty when trying to access the translation, that will be then acquired with the Dirty bit set. Thus, a subsequent Store by Core 2 incurs no additional penalty. Under the same series of events, in the baseline system relying on shootdowns, Core 1's update leaves Core 2's cached translation unaffected. Thus, a store by Core 2 results in a page fault which also includes a page table walk. However, it is possible that Core 2 never writes to the page and only reads from it. In this case, UNITD's penalty over the baseline is the page walk incurred by Core 2.

Therefore, UNITD yields a smaller penalty than the baseline system in the first case, while it downgrades performance in the second situation. The overall impact on the application is thus determined by the prevalence of either of the two scenarios. We believe that the first case that benefits UNITD is more frequent for most applications, as these synchronize threads that exhibit a consumer-producer behavior. The consumer thread does not try to read the data until the producer writes it (otherwise, the consumer reads stale data). This approach guarantees that the consumer thread's Dirty bit update precedes any translation accesses by other threads.

5.3.4 Reducing TLB Coherence Lookups

Because UNITD integrates TLBs into the coherence protocol, UNITD requires TLB coherence lookups (*i.e.*, in the PCAM) for local stores and external coherence requests for ownership. The overwhelming majority of these lookups result in TLB misses, since PTE addresses represent a small, specific subset of the memory space. To avoid wasting power on unnecessary TLB coherence lookups, UNITD can easily filter out these requests by using one of the previously proposed solutions for snoop filters [91].

5.4 Experimental Evaluation

In this section, we evaluate UNITD's performance improvement over systems relying on TLB shootdowns. We also evaluate the filtering of TLB coherence lookups, as well as UNITD's hardware cost.

5.4.1 Methodology

We use Virtutech Simics [81] to simulate an x86 multicore processor. For the memory system timing simulations we use GEMS [85]. We extend the infrastructure to accurately model page table walks and TLB accesses. We do not model the time to

Table 5.1: Target System Parameters for UNITD Evaluation.

Parameter	Value
Cores	2, 4, 8, 16 in-order scalar cores
L1D/L1I	128KB, 4-way, 64B block, 1-cycle hit
L2 cache	4MB, 4-way, 64B block, 6-cycle hit
Memory	4GB, 160-cycle hit
TLBs	1 I-TLB and 1 D-TLB per core; all 4-way set-assoc.; 64 entries for 4K pages and 64 entries for 2/4MB pages
Coherence	MOSI snooping and directory protocols
Network	broadcast tree (snooping); 2D mesh (directory)

Table 5.2: Microbenchmarks for UNITD Evaluation.

	single initiator	**multiple initiators**
COW	single_cow	multiple_cow
Unmap	single_unmap	multiple_unmap

deliver interrupts, an approximation that favors the systems with shootdowns, but not UNITD. As the Simics infrastructure updates the status bits in the background (*i.e.*, status bits are not part of the simulated system's visible state), we do not simulate their updates.

The parameters of our simulated system are given in Table 5.1. The baseline OS consists of a Fedora Core 5 distribution with a 2.6.15 SMP kernel. For the UNITD systems, we use the same kernel version recompiled without TLB shootdown procedures (*e.g.*, *flush_tlb_mm(), flush_tlb_range(), smp_invalidate_interrupt()*). We report results averaged across twenty simulated executions, with each simulation having a randomly perturbed main memory latency as described by Alameldeen *et al.* [11].

Benchmarks

Ideally, we would like to test UNITD on a set of real applications that exhibit a wide range of TLB shootdown activity. Unfortunately, we are bound to the constraints imposed by running the applications on a simulator, and not the real hardware, and therefore the real time that we can simulate is greatly decreased. For example, the wordcount results presented in Figure 5.3 were obtained for an input file of size 1GB. However, the Simics infrastructure crashed when trying to run the benchmark with an input file of just 100MB, an order of magnitude smaller.

In addition, with the exception of the wordcount benchmark from the Phoenix suite [100], we are unaware of existing benchmarks that exercise TLB shootdown mechanisms. We also do not have access to any of the applications mentioned in Section 1.3 that exercise translation coherence. As a consequence, we created a set of microbenchmarks that spend various fractions of their runtime in TLB shootdown routines triggered by one of two OS operations: copy-on-write (COW) and page unmapping.

The microbenchmarks are modeled after the map phase of the wordcount benchmark. They consist of one or multiple threads parsing a 50 MB memory-mapped file and either performing stores to the mapped pages (this triggers the kernel's COW policy if the file is memory-mapped with corresponding flags set) or unmapping pages. For the benchmarks in which multiple threads trigger shootdowns, the number of threads equals the number of cores in the system. The pairing of how many threads can trigger shootdowns (one or more shootdown initiators) with the two types of operations (COW/unmap) leads to a total of four types of microbenchmarks as shown in Table 5.2. For the benchmarks with multiple shootdown initiators, we divide the workload evenly across the threads. This yields a runtime between 150 million and 1.5 billion cycles per thread.

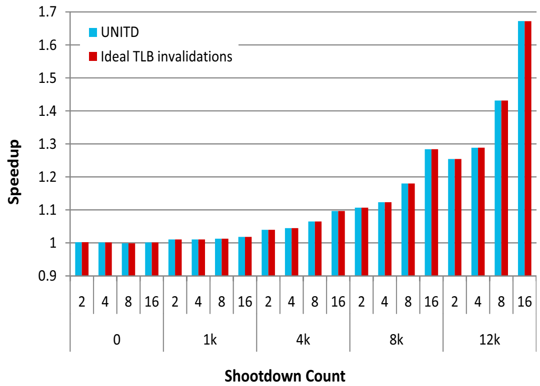
The frequency of COW/unmap operations is parameterizable and allows us to test UNITD's efficiency across a range of TLB shootdowns counts. We use the shootdown count as our parameter rather than the time spent in shootdowns because the latter varies with the number of cores in the system, as shown in Section 5.1.2. Thus, we can use the shootdown count as a constant unit of measure for performance improvements across systems with different number of cores. In our experiments, we vary the number of shootdowns between 0 and 12,000 (the 50MB input file allows for up to 12,500 4KB pages). Varying the number of TLB shootdowns reveals the benefits of UNITD, as well as creating a correspondence between the possible benefits and the time spent by the baseline system in shootdowns.

In addition to these microbenchmarks, we study UNITD's performance on applications that exhibit no shootdowns, including swaptions from the Parsec suite [18] and pca, string-match, and wordcount (with a much smaller input file than the one used in Figure 5.3, leading to a negligible number of shootdowns) from the Phoenix suite [100]. We perform these experiments to confirm that UNITD does not degrade common-case performance.

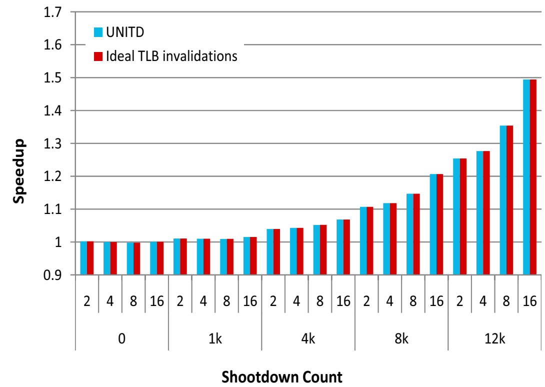
5.4.2 Performance

In all performance experiments, we compare UNITD to two systems. The first comparison is to a baseline system that relies on TLB shootdowns. All results are normalized with respect to the baseline system with the same number of cores. For each benchmark, the x-axis shows both the number of shootdowns present in the baseline execution and the number of cores.

The second comparison is to a system with ideal (zero-latency) translation invalidations. This ideal-invalidation system uses the same modified OS as UNITD (*i.e.*, with no TLB shootdown support) and verifies that a translation is coherent whenever it is accessed in the TLB. The validation is done in the background and has no performance impact. If the cached translation is found to be incoherent, it is invalidated and reacquired; the re-acquisition of the translation is not ideal (*i.e.*, it has non-zero latency). We do not refer to this system as "ideal translation coherence" because such a system would be one that updates the TLB cached translations



(a) Snooping protocol



(b) Directory protocol

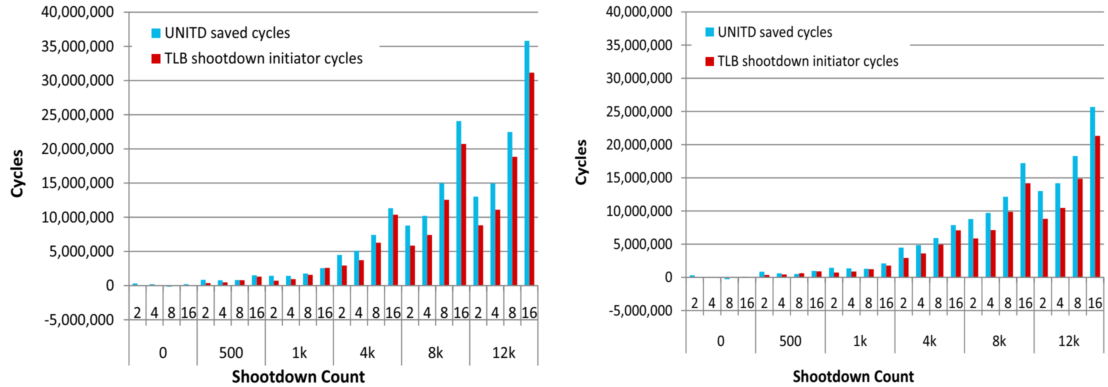
Figure 5.7: UNITD Speedup Over Baseline System for Single_unmap Benchmark.

rather than invalidating them. Besides demonstrating UNITD's efficiency, the comparison with the system with ideal TLB invalidations reveals if UNITD incurs any performance degradation due to ensuring coherence on PTE's block addresses rather than full addresses.

Single_unmap

Figure 5.7 shows UNITD's performance on the single_unmap benchmark as a function of the number of shootdowns and number of cores on systems with both snooping and directory protocols. For this benchmark, the application's runtime is determined by the thread performing the unmaps. Thus, the impact of TLB shootdowns on the runtime is represented by the shootdown initiator routine's effect on the application. With respect to this microbenchmark, there are three main conclusions.

First, UNITD is efficient in ensuring translation coherence, as it performs as well as the system with ideal TLB invalidations. In a few cases, UNITD even outperforms the ideal case although the performance gain is a statistically insignificant artifact of the invalidation of translations in the TLB, which aids the set-associative TLBs. In the ideal case, the invalidation occurs if the invalid translation is accessed. Thus,



(a) Snooping protocol

(b) Directory protocol

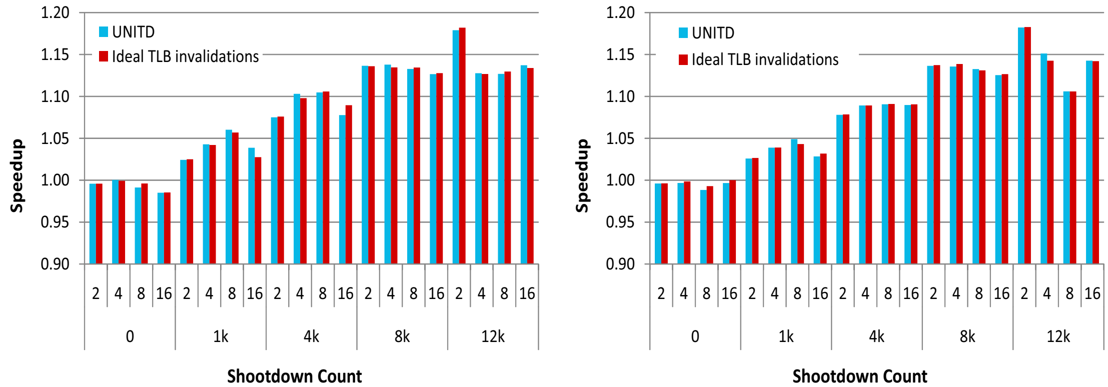
Figure 5.8: Runtime Cycles Eliminated by UNITD Relative to Baseline System for Single_unmap Benchmark.

it is possible for the system to evict a useful translation (*i.e.*, one that will be soon accessed) because it is the least recently used translation, although there is a more recently-accessed translation that became stale after the access.

Second, UNITD speedups increase with the number of TLB shootdowns and with the number of cores. If the shootdown count is large, the performance benefits scale accordingly, up to 68% speedup for the 16-core configuration for the snooping system and up to 50% for the directory protocol. In addition, even for the same number of shootdowns, UNITD's improvements increase with the increasing number of cores. For 4000 shootdowns, UNITD's speedup increases from 3% for 2 cores to 9% for 16 cores. The difference increases for 12000 shootdowns, from 25% for 2 cores to 68% for 16 cores. Therefore, we expect UNITD to be particularly beneficial for many-core systems.

Third, as expected, UNITD has no impact on performance in the absence of TLB shootdowns. UNITD can impact performance only through invalidations of TLB cached translations. In the absence of such invalidation requests, we expect the runtime to be identical.

Understanding UNITD's Performance Benefit. To better understand the



(a) Snooping protocol

(b) Directory protocol

Figure 5.9: UNITD Speedup Over Baseline System for Multiple_unmap Benchmark.

performance benefits of UNITD, Figure 5.8 shows a comparison for the single_unmap benchmark between UNITD's runtime and the time spent triggering the TLB shootdowns routines in the baseline system. UNITD's runtime is shorter than the baseline's runtime by a number of cycles that is greater than the cycles spent by the baseline in TLB shootdowns. As mentioned in Section 5.1.2, the latency associated with the TLB shootdowns on the baseline x86/Linux system is increased by the full flush of the TLBs during certain shootdowns because full flushes lead to subsequent page table walks. UNITD avoids this extra penalty, thus resulting in a runtime reduction greater than the number of TLB shootdown cycles.

Multiple_unmap

Figure 5.9 shows the speedup when there are multiple threads unmapping the pages for snooping and directory systems, respectively. For this benchmark, we measure the time required by all threads to finish their work. The impact of TLB shootdown on execution time of the baseline system is represented by both the time spent by threads in triggering shootdowns, as well as the time they spend in servicing other threads' shootdowns.

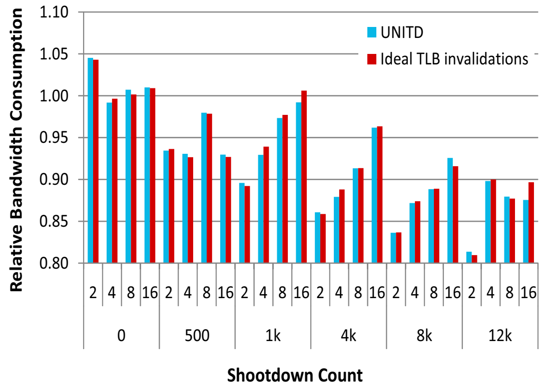
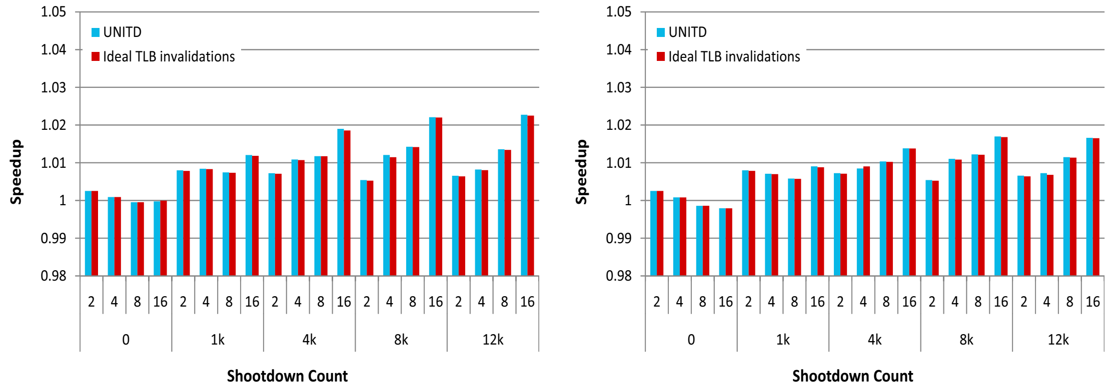


Figure 5.10: UNITD Relative Bandwidth Consumption For Multiple_unmap Benchmark with Snooping Coherence. Results are normalized to the baseline system.

UNITD once again matches the performance of the system with ideal TLB invalidations. Moreover, UNITD proves beneficial even for a small number of TLB shootdowns. For just 1000 shootdowns, UNITD yields a speedup of more than 5% for 8 cores. Compared to single_unmap, UNITD's speedups are generally lower, particularly for greater numbers of shootdowns and cores. The reason for this phenomenon is contention among the multiple initiators for locks, which decreases the percent of overall runtime represented by the shootdown routines.

We also observe small speedups/slowdowns for the executions with zero shootdowns. These are artifacts caused by the differences between the baseline kernel and our modified kernel, as evidenced by UNITD's trends also being exhibited by the system with ideal TLB invalidations. These differences are likely caused by the placement of the kernel instructions/data at different addresses from the baseline configuration.

Because UNITD reduces both the number of instructions executed and the number of page table walks, an additional UNITD benefit is lower interconnect network bandwidth traffic compared to the baseline system. Figure 5.10 presents the relative bandwidth consumption, compared to the baseline, during the execution of multiple_unmap on a snooping system. UNITD consistently requires less bandwidth,



(a) Snooping protocol

(b) Directory protocol

Figure 5.11: UNITD Speedup Over Baseline System for Single_cow Benchmark.

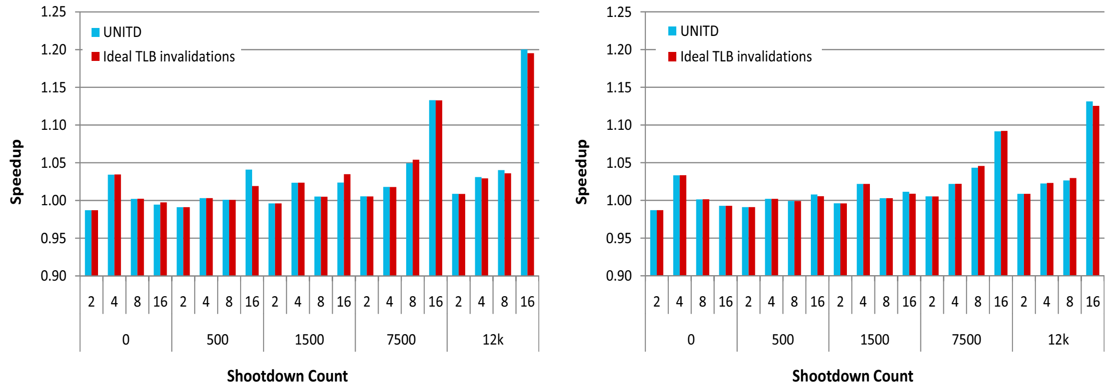
yielding up to a 12% reduction in bandwidth consumption for 16 cores.

Single_cow

Figure 5.11 shows the performance when a single thread triggers shootdown by relying on the COW procedure. In this case, the TLB shootdown is a smaller percentage of runtime for COW (due to long-latency copy operations) than unmap, and therefore there is less opportunity for UNITD to improve performance. For this microbenchmark, the baseline runtime is affected only by the time the initiator spends in triggering the shootdowns. This leads to neglegible improvements for the UNITD system, of less than 2%. Nevertheless, UNITD performs as well as the system with ideal invalidations.

Multiple_cow

The application behavior changes with multiple threads executing the COW operations. Performance is affected by the time spent by threads in TLB shootdown initiation, as for single_cow, but also by the time to service TLB shootdown interrupts triggered by other threads. The cost of executing the interrupt handler increases with the number of cores as shown in Section 5.1.2.



(a) Snooping protocol **(b)** Directory protocol

Figure 5.12: UNITD Speedup Over Baseline System for Multiple_cow Benchmark.

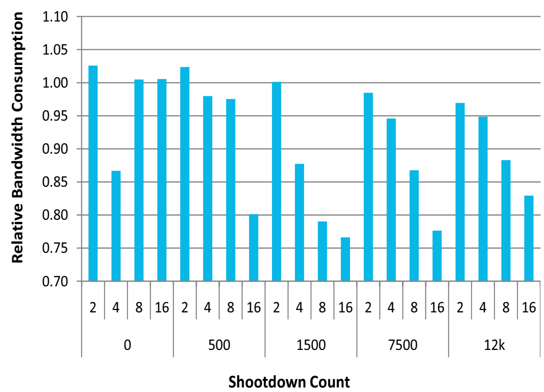


Figure 5.13: UNITD Relative Bandwidth Consumption for Multiple_cow Benchmark with Snooping Coherence. Results are normalized to the baseline system.

As a consequence, performance is greatly affected by TLB shootdowns for multiple_cow, as shown in Figure 5.12, which reveals the differences with respect to the single_cow microbenchmark. This trend is especially clear for 16 cores. In this case, UNITD outperforms the base case by up to 20% for the snooping protocol.

Similar to the results shown for multiple_unmap benchmark, UNITD's benefits translate in a direct reduction of the interconnect bandwidth consumption as shown in Figure 5.13. In this case, UNITD yields up to a 24% reduction in bandwidth consumption.

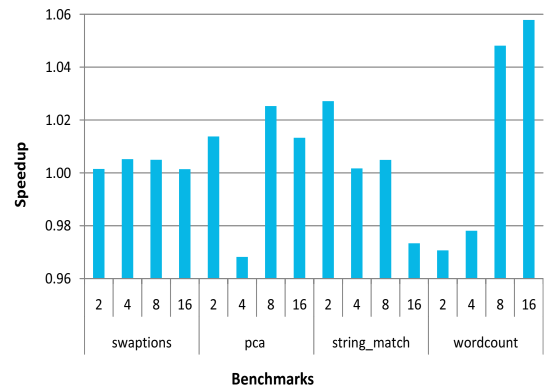


Figure 5.14: UNITD Speedup on Real Benchmarks.

Real Benchmarks

For applications that perform no TLB shootdowns when run on the baseline system, we expect UNITD to have negligible performance impact. UNITD's only performance impact occurs in situations when there are stores to PTEs that invalidate TLB entries. Figure 5.14 presents the results for such benchmarks. All of the applications, including wordcount (because of its smaller input size), spend a negligible amount of time in TLB shootdowns (less than 0.01% of total execution time). The results are as expected: for these applications, UNITD performs as well as the baseline, with small, statistically insignificant variations that are caused by the difference between the baseline kernel and the UNITD one.

TLB Coherence Lookup Filtering

Despite UNITD's performance transparency, UNITD's TLB coherence lookups result in wasted PCAM power, as most lookups miss in the PCAM. As described in Section 5.3.4, a large fraction of these lookups can be avoided by using a simple filter. We evaluate the efficiency of this solution by implementing a small include-JETTY filter [91]. The filter consists of 2 blocks of 16 entries each, indexed by bits 19-16 and 15-12 of the physical address. We use bits 19-12 for filtering in order to isolate the pages that contain PTEs and that are likely to not be accessed by the applications. Using

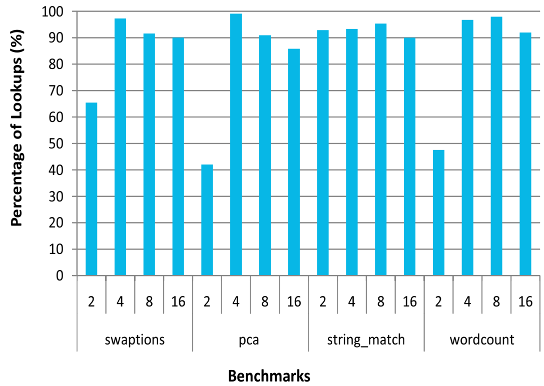


Figure 5.15: Percentage of TLB Coherence Lookups Filtered with a Simple JETTY Filter.

the upper address bits will result in increased filter accuracy, but will also increase the size of the filter. Even with this simple filter, we can filter around 90% of the coherence lookups for most systems, as Figure 5.15 shows.

We must note however that any filtering mechanism must take advantage of the specific placement of page table entries in memory. Although most operating systems adopt common placement of the page tables (*e.g.*, in the lowest memory pages), this information is system-specific. Consequently, the operating system could provide the filter with hints about the regions of physical memory where it stores the page tables.

5.5 UNITD Hardware Cost

The hardware and power costs associated with UNITD are almost entirely represented by the PCAM and depend on its implementation. Conceptually, the PCAM can be viewed as a dual-tag extension of the TLB. Thus, for a 32-bit system with 64-byte cache blocks, the PCAM tags require 26 bits compared to the 20 bits of the TLB tags (for 4-Kbyte pages). For a 64-bit system, the PCAM tags increase to 38 bits due to the 44-bit physical addresses. The hardware and power costs for a PCAM with a small number of entries (*e.g.*, 64 or fewer) are comparable to those for a core's store queue with the same number of entries. For a PCAM with a large number of entries, a physical CAM may exceed desired area and power budgets. In this case, one could use an alternate, lower-cost implementation for a logical CAM, as described in Section 5.2.2.

Independent of the implementation, accesses to the TLB for TLB coherence purposes (rather than accesses for translation lookups) are off the critical path of a memory access. Therefore, the PCAM implementation can be clocked at a lower frequency than the rest of the core or can be implemented as a 2-level structure with pipelined accesses. The latter case supports a filtering of the invalidation lookups, as not finding a match at the first level implies that the PCAM does not contain the address. For example, if the first level consists of bits 19-12 of the physical address, most lookups can be filtered after the first level as shown by our JETTY filter experiment.

5.6 Related Work

Section 5.1.1 described the software TLB shootdown routine as the most common technique of maintaining TLB coherence. Previous research on translation coherence has focused on three areas: speeding up the shootdown procedure by providing dedicated hardware support, reducing the number of processors involved in the shootdown, and proposing alternative solutions for maintaining translation coherence.

Hardware support for shootdowns. Shootdown's complexity and latency penalty can be reduced by using mechanisms other than inter-processor interrupts. Among current commercial architectures, both Power ISA and Intel IA64 support microarchitectural mechanisms for global TLB invalidations. These hardware designs are still architecturally visible and thus provide less flexibility than UNITD.

Reducing the number of shared translations. Several OS implementations have indirectly reduced the impact of TLB shootdowns on application performance, by reducing the number of shared translations. Tornado [45] and K42 [12] introduce the concept of clustered objects that are associated with each thread, thus reducing the contention of the kernel managed resources. Corey [130] follows the same concept by giving applications the power to decide which PTEs are core-private and thus eliminate shootdowns for these PTEs.

Alternative translation coherence mechanisms. Teller has proposed several hardware-based mechanisms for handling TLB coherence [126], but they restrict the system model in significant ways, such as prohibiting the copy-on-write policy. Wood *et al.* [132] proposed a different approach to handling translations, by using virtual caches without a memory-based TLB. Translations are cached in the data cache and thus translation coherence is maintained by the cache coherence protocol. A drawback of this approach is that it requires special handling of the status and protection bits that must be replicated at each data block [133]. The design also complicates the handling of virtual memory based optimizations such as concurrent garbage collection or copy-on-write [13].

5.7 Conclusions and Future Work

We believe the time has come to adopt hardware support for address translation coherence. We propose UNITD, a unified hardware coherence protocol that incorporates address translation coherence together with cache coherence. UNITD eliminates the performance costs associated with translation coherence as currently implemented through TLB shootdown software routines. We demonstrate that, on systems with 16 cores, UNITD can achieve speedups of up to 68% for benchmarks that make frequent changes to the page tables. We expect the benefits yielded by UNITD to be even greater for many-core systems. Finally, we demonstrate that UNITD has no adverse performance impact for other applications, while incurring a small hardware cost.

One of the challenges to address in the current implementation of UNITD is the power consumption of the PCAM structure. Although we demonstrated that filtering can eliminate many of the coherence lookups, the filtering mechanisms adds its own power consumption to the system. Next, we briefly describe a possible solution to reduce the number of PCAM accesses by modifying the coherence protocol such that the PCAM is probed only when translation coherence is required. The key concept of the solution is to mark blocks containing PTEs and probe the PCAM only on coherence requests for these blocks. Cache or memory blocks are marked as PTE holders once the first page table walk occurs on a resident PTE. If no such table walk exists, then no TLB contains a cached copy of the corresponding translation. The "PTE holder" information is maintained by the owner of the block. If the protocol does not have an Owned state, the information resides with the valid copies of the block, either at memory or with the cache that has block in the Modified state. A core specifically marks coherence requests that require PCAM lookups once it determines that the block it operates on is a "PTE holder". This information might become available to the core once it receives the block, which requires the core to lock the block and issue a coherence request targeting only PCAMs. The solution guarantees the reduction of PCAM lookups to only coherence requests for cache blocks containing PTEs, and trades power consumption for increased complexity of the coherence protocol.

We expect future research to extend beyond improvements to the UNITD framework. One of the key aspects facilitated by UNITD is the integration of I/O devices and other non-processor components in a single shared-address memory space. Architects can take advantage of this opportunity to explore new performance-oriented design paradigms. Previous research showed the advantages of supporting translations in network cards [102]. We envision that these improvements can be extended to other devices too. For example, supporting translations in graphics processors allows the hardware to migrate threads between main cores and graphics cores without software intervention for increased performance.

6

Conclusions

Harnessing the full performance potential of many-core processors requires hardware designers to consider not only the advantages, but also the problems introduced by these new architectures, and design and provision resources accordingly. The hardware challenges arise from both the processor's increased structural complexity and the reliability problems of the silicon substrate. In this thesis, we addressed these challenges on three coordinates: tolerating permanent faults, facilitating static and dynamic verification through precise specifications, and designing scalable coherence protocols.

We introduced the Core Cannibalization Architecture, a design paradigm for increased processor availability and performance in the presence of hard faults in cores. Relying on a novel reconfiguration mechanism, CCA allows cores to replace faulty components with structures borrowed from neighboring cores. To support the cannibalization process, CCA exploits the on-chip locality of cores. Therefore, CCA benefits if cores are clustered in small groups (we used three-core and four-core groups in our experiments), as these configurations reduce the performance cost of borrowing components.

The evaluation of the four-core CCA processors confirmed our initial hypothesis about CCA's performance, which is determined by the time required to access remote resources, as well as the partitioning of cores in CCs/NCs. For 90nm technology, slowing down the clock to accommodate the access to a cannibalized structure is preferable to adding an extra pipeline stage, as demonstrated by the CCA4-clock(3/1) design outperforming the CCA4-pipe(3/1) configuration. For future technologies, this trend might be reversed as the wire delays for the remote access become a larger fraction of the clock period. Nevertheless, for the CCA4-pipe configurations to become cost-effective, architects must propose solutions to reduce the buffering required by the extra pipe stage and, in particular, the buffers used to avoid pipeline hazards.

With respect to assignment of cores as NCs and CCs, we demonstrated that supporting more reconfiguration possibilities by assigning multiple cores to be CCs provides cost-effective performance gains. The CCA4-clock(2/2) design has an area overhead of 1% compared to CCA4-clock(3/1), but takes advantage of the 2 CCs to yield significantly better performance especially over longer periods of time–12% better for 12 years assuming our expected failure rate.

Maximizing the performance of any CCA configuration also depends on minimizing the penalty during fault-free execution, especially if the expected utilization period for the chip is small (*e.g.*, 3-4 years). In such situations, the CCA processors might not benefit from the reconfiguration mechanism and will underperform regular processors. In this respect, the tight integration between cores assumed by CCA gives CCA chips an advantage over more flexible solutions such as StageNet [48]. Processors based on the latter concept incur a bigger fault-free penalty, and thus need a longer period of time to become advantageous. For common industrial lifetimes of 10-12 years, CCA offers a better compromise between reconfiguration flexibility and performance gains given the expected failure rates for future silicon technologies.

We also identified address translation as a system that is prone to design faults and that currently lacks solutions for detecting incorrect behavior. We believe one cause of these correctness problems is the designer's tendency to over-simplify memory consistency and especially to neglect translations' impact on memory consistency. We addressed this issue by proposing a framework for precise specifications of translation-aware memory consistency models. Our framework emphasizes the importance of considering the hierarchical structure of memory consistency models, as previously described by Adve and Gharachorloo [5]. As part of this framework, we discussed in detail two levels of memory consistency, PAMC and VAMC, and described the AT aspects that impact VAMC.

The precise specifications of VAMC models simplify the programmer's reasoning about correctness of AT-related code, support static and dynamic verification, and facilitate designing hardware that involves AT. In addition, the framework allows architects to evaluate more easily the tradeoffs between design decisions and the hardware/software support required for a specific VAMC model. Consider the case of status bits updates. In a system with software managed TLBs, these updates occur in exception handlers and, consequently, are serialized with respect to any other user-level instruction (*i.e.*, instructions outside the handler), including the instruction triggering the update. If the designer's intention is to support a VAMC model that relaxes the orderings between status bits updates and memory operations, then the system should rely on hardware rather than software to manage the TLBs, or at least to handle the updates.

To support checking correctness of VAMC implementations, we proposed a set of implementation-independent invariants that characterize AT, and we developed DVAT, a mechanism for dynamic verification of AT. The AT correctness framework is applicable to all commercial AT systems that we are aware of. Representative of the framework's coverage is that all AT-related design bugs described in recent processor errata [2, 3, 4, 59, 61, 62, 63] break at least one of the framework's invariants. Consequently, we expect DVAT to detect all such design faults, as successfully demonstrated in our DVAT error detection experiments.

The current DVAT implementation assumes a specific AT model. However, DVAT can be extended to check correctness of more relaxed AT models. As long as architects prove that the AT model bridges the gap between a specific PAMC-VAMC pair, DVAT can be used in association with previous solutions for checking PAMC [89] to provide runtime error detection for the VAMC implementations.

The last direction of this thesis addressed scalable translation coherence protocols. We proposed to take advantage of the hardware's benefits, such as speed and architectural decoupling, and move translation coherence into hardware. Our solution, UNITD, integrates translation coherence into the regular cache coherence protocol. By having TLBs participate in cache coherence such as instruction/data caches, UNITD reduces the performance penalty associated with translation coherence to almost zero. In addition, compared to TLB shootdown routines, UNITD avoids additional performance penalties due to cache pollution (*i.e.*, due to shootdown-related instructions/data), pipeline flushes for servicing shootdown interrupts, or page table walks caused by full TLB flushes.

UNITD's performance benefits depend on how often the running application requires translation coherence. Moreover, our microbenchmark analysis reveals that translation coherence has a higher performance impact if it is triggered by page unmapping rather than COW operations. Thus, for a single thread generating 12,000 translation coherence operations on a 16-core system, UNITD yields speedups of 68% for page unmap, compared to less than 3% for COW. For COW, translation coherence operations are a smaller fraction of the total runtime, compared to the associated page copying operations. Even for COW, translation coherence has a higher performance impact on systems with TLB shootdowns if multiple cores are involved in the procedure, as cores must service shootdown interrupts. We expect UNITD's speedups for COW to increase on systems relying on copying accelerators [69], where there is a larger performance opportunity that UNITD can exploit.

Nevertheless, UNITD would benefit applications that rely heavily on translation coherence, such as hardware transactional memory (*e.g.*, XTM [40]), user-level memory management for debugging [43], and concurrent garbage collection [39].

Our solutions cover a small subset of the challenges related to correct execution and performance in many-core processors design. CCA increases processor availability by targetting faults in cores, and future research should evaluate the feasibility of extending the concept to other processor structures. The framework we propose for translation-aware memory consistency specifications supports not only static or dynamic verification of consistency, but also the exploration of new VAMC models and the analysis of possible performance benefits of translation-relaxed consistency models. UNITD bridges the gap to a single-address memory-shared space that extends beyond the conventional processor to include graphics processors and I/O devices. These directions represent just a few possible avenues of future research exploring the space of dependability and performance of many-core processors that are facilitated by the research contributions described in this thesis.

Bibliography

[1] M. E. Acacio, J. González, J. M. García, and J. Duato. Owner Prediction for Accelerating Cache-to-cache Transfer Misses in a cc-NUMA Architecture. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, pages 1–12, 2002.

[2] Advanced Micro Devices. Revision Guide for AMD Athlon64 and AMD Opteron Processors. Publication 25759, Revision 3.59, September 2006.

[3] Advanced Micro Devices. Revision Guide for AMD Family 10h Processors. Technical Report 41322, September 2008.

[4] Advanced Micro Devices. Revision Guide for AMD Family 11h Processors. Technical Report 41788, July 2008.

[5] S. V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, 29(12):66–76, December 1996.

[6] S. V. Adve, V. S. Pai, and P. Ranganathan. Recent Advances in Memory Consistency Models for Hardware Shared Memory Systems. In *Proceedings of the IEEE*, volume 87, pages 445–455, March 1999.

[7] A. Agarwal, R. Bianchini, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, B-H. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 2–13, June 1995.

[8] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. An Evaluation of Directory Schemes for Cache Coherence. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, pages 280–298, May 1988.

[9] N. Agarwal, L. Peh, and N. K. Jha. In-network Coherence Filtering: Snoopy Coherence Without Broadcasts. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 232–243, December 2009.

[10] N. Aggarwal, P. Ranganathan, N. P. Jouppi, and J. E. Smith. Configurable Isolation: Building High Availability Systems with Commodity Multi-Core Processors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, pages 470–481, June 2007.

[11] A. R. Alameldeen, C. J. Mauer, M. Xu, P. J. Harper, M. M.K. Martin, D. J. Sorin, M. D. Hill, and D. A. Wood. Evaluating Non-deterministic Multi-threaded Commercial Workloads. In *Proceedings of the 5th Workshop on Computer Architecture Evaluation Using Commercial Workloads*, pages 30–38, February 2002.

[12] J. Appavoo, D. D. Silva, O. Krieger, M. Auslander, A. Waterland, R. W. Wisniewski, J. Xenidis, M. Stumm, and L. Soares. Experience Distributing Objects in an SMMP OS. *ACM Transactions on Computer Systems*, 25(3):6, 2007.

[13] A. W. Appel and K. Li. Virtual Memory Primitives for User Programs. *SIGPLAN Notices*, 26(4):96–107, 1991.

[14] Arvind and J. Maessen. Memory Model = Instruction Reordering + Store Atomicity. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, pages 29–40, June 2006.

[15] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, December 2006.

[16] T. M. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *Proceedings of the 32nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 196–207, November 1999.

[17] D. Bernick, B. Bruckert, P. D. Vigna, D. Garcia, R. Jardine, J. Klecka, and J. Smullen. NonStop Advanced Architecture. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 12–21, June 2005.

[18] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 72–81, October 2008.

[19] D. L. Black, R. F. Rashid, D. B. Golub, and C. R. Hill. Translation Lookaside Buffer Consistency: A Software Approach. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 113–122, April 1989.

[20] J. Blome, S. Feng, S. Gupta, and S. Mahlke. Self-calibrating Online Wearout Detection. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 109–122, December 2007.

[21] M. Blum and S. Kannan. Designing Programs that Check Their Work. In *ACM Symposium on Theory of Computing*, pages 86–97, May 1989.

[22] R. D. Blumofe and D. P. Papadopoulos. Hood: A User-Level Thread Library for Multiprogramming Multiprocessors. Technical report, University of Texas at Austin, 1998.

[23] H. Boehm and S. V. Adve. Foundations of the C++ Concurrency Memory Model. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 68–78, June 2008.

[24] S. Borkar. Thousand Core Chips: A Technology Perspective. In *Proceedings of the 44th Annual Design Automation Conference*, pages 746–749, 2007.

[25] S. Borkar, N. P. Jouppi, and P. Stenstrom. Microprocessors in the Era of Terascale Integration. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 237–242, 2007.

[26] F. A. Bower, P. G. Shealy, S. Ozev, and D. J. Sorin. Tolerating Hard Faults in Microprocessor Array Structures. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 51–60, June 2004.

[27] Bower, F.A. and Sorin, D.J. and Ozev, S. A Mechanism for Online Diagnosis of Hard Faults in Microprocessors. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 197–208, November 2005.

[28] Cadence Design Systems. Silicon Ensemble PKS datasheet. Online, December 2003. `http://www.cadence.com/datasheets/sepks_ds.pdf`.

[29] H. W. Cain and M. H. Lipasti. Verifying Sequential Consistency Using Vector Clocks. In *Revue in conjunction with Symposium on Parallel Algorithms and Architectures*, pages 153–154, August 2002.

[30] J. F. Cantin, M. H. Lipasti, and J. E. Smith. Dynamic Verification of Cache Coherence Protocols. In *Workshop on Memory Performance Issues*, June 2001.

[31] L. Carter, J. Feo, and A. Snavely. Performance and Programming Experience on the Tera MTA. In *Proceedings of the SIAM Conference on Parallel Processing*, March 1999.

[32] M. Cekleov and M. Dubois. Virtual-Address Caches Part 1: Problems and Solutions in Uniprocessors. *IEEE Micro*, 17(5):64–71, September 1997.

[33] M. Cekleov and M. Dubois. Virtual-Address Caches, Part 2: Multiprocessor Issues. *IEEE Micro*, 17(6):69–74, November 1997.

[34] K. Chakraborty, P. M. Wells, and G. 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*, pages 283–292, October 2006.

[35] M. Chang and K. Koh. Lazy TLB Consistency for Large-Scale Multiprocessors. In *Proceedings of the 2nd Aizu International Symposium on Parallel Algorithms/Architecture Synthesis*, pages 308–315, March 1997.

[36] K. Chen, S. Malik, and P. Patra. Runtime Validation of Memory Ordering Using Constraint Graph Checking. In *Proceedings of the 13th International Symposium on High-Performance Computer Architecture*, pages 415–426, February 2008.

[37] K. Chen, S. Malik, and P. Patra. Runtime Validation of Transactional Memory Systems. In *Proceedings of the International Symposium on Quality Electronic Design*, pages 750–756, March 2008.

[38] Y.S. Chen and M. Dubois. Cache Protocols with Partial Block Invalidations. In *Proceedings of 7th International Parallel Processing Symposium*, pages 16–23, April 1993.

[39] P. Cheng and G. E. Blelloch. A Parallel, Real-time Garbage Collector. *ACM SIGPLAN Notices*, 36(5):125–136, May 2001.

[40] J. Chung, C. C. Minh, A. McDonald, T. Skare, H. Chafi, B. D. Carlstrom, C. Kozyrakis, and K. Olukotun. Tradeoffs in Transactional Memory Virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 371–381, October 2006.

[41] Cisco Systems. Cisco Carrier Router System. Online, October 2006. `http://www.cisco.com/application/pdf/en/us/guest/products/ps5763/c1031/cdccont_0900aecd800f8118.pdf`.

[42] C. Constantinescu. Trends and Challenges in VLSI Circuit Reliability. *IEEE Micro*, 23(4):14–19, 2003.

[43] D. Dhurjati and V. Adve. Efficiently Detecting All Dangling Pointer Uses in Production Servers. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 269–280, 2006.

[44] A. Erlichson, N. Nuckolls, G. Chesson, and J. Hennessy. SoftFLASH: Analyzing the Performance of Clustered Distributed Virtual Shared Memory. *SIGOPS Operating Systems Review*, 30(5), 1996.

[45] B. Gamsa, O. Krieger, and M. Stumm. Tornado: Maximizing Locality and Concurrency in a Shared Memory Multiprocessor Operating System. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, pages 87–100, 1999.

[46] K. Gharachorloo, A. Gupta, and J. Hennessy. Two Techniques to Enhance the Performance of Memory Consistency Models. In *Proceedings of the International Conference on Parallel Processing*, volume I, pages 355–364, August 1991.

[47] M. Gschwind. Optimizing Data Sharing and Address Translation for the Cell BE Heterogeneous Chip Multiprocessor. In *Proceedings of the IEEE International Conference on Computer Design*, pages 478–485, October 2008.

[48] S. Gupta, S. Feng, A. Ansari, J. Blome, and S. Mahlke. The StageNet Fabric for Constructing Resilient Multicore Systems. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, pages 141–151, November 2008.

[49] S. Gupta, S. Feng, J. Blome, and S. Mahlke. StageNetSlice: A Reconfigurable Microarchitecture Building Block for Resilient CMP Systems. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 1–10, October 2008.

[50] D. B. Gustavson. The Scalable Coherent Interface and Related Standards Projects. *IEEE Micro*, 12(1):10–22, 1992.

[51] E. G. Hallnor and S. K. Reinhardt. A Fully Associative Software-Managed Cache Design. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 107–116, June 2000.

[52] T. Heijmen. Soft Error Rates in Deep-Submicron CMOS Technologies. In *Proceedings of the 12th IEEE International Symposium on On-Line Testing*, page 271, 2006.

[53] J. Held, J. Bautista, and S. Koehl. From a Few Cores to Many: A Tera-scale Computing Research Overview. White Paper. Intel Corporation, 2006.

[54] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach, Fourth Edition*. Morgan Kaufmann Publishers Inc., 2006.

[55] S. Heo, K. Barr, and K. Asanovic. Reducing Power Density Through Activity Migration. In *Proceedings of the 2003 International Symposium on Low Power Electronics and Design*, pages 217–222, 2003.

[56] M. D. Hill, A. E. Condon, M. Plakal, and D. J. Sorin. A System-Level Specification Framework for I/O Architectures. In *Proceedings of the 11th ACM Symposium on Parallel Algorithms and Architectures*, pages 138–147, June 1999.

[57] M. D. Hill, J. R. Larus, S. K. Reinhardt, and D. A. Wood. Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessor. *ACM Transactions on Computer Systems*, 11(4):300–318, November 1993.

[58] R. Ho, K.W. Mai, and M.A. Horowitz. The Future of Wires. In *Proceedings of the IEEE*, volume 89, pages 490–504, April 2001.

[59] IBM. IBM PowerPC 750FX and 750FL RISC Microprocessor Errata List DD2.X, version 1.3, February 2006.

[60] Intel Corporation. A Formal Specification of Intel Itanium Processor Family Memory Ordering. Document Number 251429-001, October 2002.

[61] Intel Corporation. Intel Pentium 4 Processor Specification Update. Document Number 249199-065, June 2006.

[62] Intel Corporation. Intel Core Duo Processor and Intel Core Solo Processor on 65nm Process Specification Update. Technical Report 309222-016, February 2007.

[63] Intel Corporation. Intel Core2 Extreme Quad-Core Processor QX6000 Sequence and Intel Core2 Quad Processor Q6000 Sequence Specification Update. Technical Report 315593-021, February 2008.

[64] Intel Corporation. Intel Processor Identification and the CPUID Instruction. Application Note 485, March 2009.

[65] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez. Core Fusion: Accommodating Software Diversity in Chip Multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, pages 186–197, June 2007.

[66] ITRS. The International Technology Roadmap for Semiconductors 2009 - Design. Technical report, ITRS, 2009.

[67] A. Iyer and D. Marculescu. Power Efficiency of Voltage Scaling in Multiple Clock, Multiple Voltage Cores. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 379–386, November 2002.

[68] D. Jewett. Integrity S2: A Fault-Tolerant UNIX Platform. In *Proceedings of the 21st International Symposium on Fault-Tolerant Computing Systems*, pages 512–519, June 1991.

[69] X. Jiang, Y. Solihin, L. Zhao, and R. Iyer. Architecture Support for Improving Bulk Memory Copying and Initialization Performance. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 169–180, September 2009.

[70] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-Way Multithreaded SPARC Processor. *IEEE Micro*, 25(2):21–29, 2005.

[71] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 81–92, December 2003.

[72] L. Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[73] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[74] D. Lampret. OpenRISC 1200 IP Core Specification. Online, Dec. 2006. `http://www.opencores.org`.

[75] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 241–251, June 1997.

[76] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communicatons Systems. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 330–335, December 1997.

[77] H. Lee, S. Cho, and B. R. Childers. Performance of Graceful Degradation for Cache Faults. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI*, pages 409–415, 2007.

[78] J. Levon et al. Oprofile. Online. `http://oprofile.sourceforge.net`.

[79] M. Li, P. Ramachandran, S. K. Sahoo, S. Adve, V. Adve, and Y. 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*, pages 265–276, March 2008.

[80] M. Linklater. Optimizing Cell Core. *Game Developer Magazine*, pages 15–18, April 2007.

[81] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, February 2002.

[82] K. Magoutis. Memory Management Support for Multi-Programmed Remote Direct Memory Access (RDMA) Systems. In *Proceedings of the IEEE International Conference on Cluster Computing*, volume 0, pages 1–8, September 2005.

[83] J. Manson, W. Pugh, and S. V. Adve. The Java Memory Model. In *Proceedings of the 32nd Symposium on Principles of Programming Languages*, pages 378–391, January 2005.

[84] M. M. K. Martin, M. D. Hill, and D. A. Wood. Token Coherence: Decoupling Performance and Correctness. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 182–193, June 2003.

[85] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *Computer Architecture News*, 33(4):92–99, September 2005.

[86] A. Meixner, M. E. Bauer, and D. J. Sorin. Argus: Low-Cost, Comprehensive Error Detection in Simple Cores. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 210–222, December 2007.

[87] A. Meixner and D. J. Sorin. Dynamic Verification of Sequential Consistency. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 482–493, June 2005.

[88] A. Meixner and D. J. Sorin. Dynamic Verification of Memory Consistency in Cache-Coherent Multithreaded Computer Architectures. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 73–82, June 2006.

[89] A. Meixner and D. J. Sorin. Error Detection via Online Checking of Cache Coherence with Token Coherence Signatures. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, pages 145–156, February 2007.

[90] MIPS Technologies. The MIPS32 1004K Product Brief. Online, April 2008. `http://www.mips.com/media/files/$$1004k/MIPS32%5F1004K%5Frev1.pdf`.

[91] A. Moshovos, G. Memik, A. Choudhary, and B. Falsafi. JETTY: Filtering Snoops for Reduced Energy Consumption in SMP Servers. In *Proceedings of the 17th IEEE Symposium on High-Performance Computer Architecture*, pages 85–96, January 2001.

[92] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. Architecting Efficient Interconnects for Large Caches with CACTI 6.0. *IEEE Micro*, 28(1):69–79, 2008.

[93] T. Nakura, K. Nose, and M. Mizuno. Fine-Grain Redundant Logic Using Defect-Prediction Flip-Flops. In *Proceedings of the International Solid-State Circuits Conference*, pages 402–611, February 2007.

[94] S. Narayanasamy, B. Carneal, and B. Calder. Patching Processor Design Errors. In *Proceedings of the International Conference on Computer Design*, pages 491–498, October 2006.

[95] U. G. Nawathe, M. Hassan, K. C. Yen, A. Kumar, A. Ramachandran, and D. Greenhill. Implementation of an 8-Core, 64-Thread, Power-Efficient SPARC Server on a Chip. *IEEE Journal of Solid-State Circuits*, 43(1):6–20, 2008.

[96] B. W. O'Krafka and A. R. Newton. An Empirical Evaluation of Two Memory-efficient Directory Methods. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 138–147, May 1990.

[97] M. D. Powell, A. Biswas, S. Gupta, and S. S. Mukherjee. Architectural Core Salvaging in a Multi-core Processor for Hard-error Tolerance. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, pages 93–104, June 2009.

[98] M. Prvulovic, Z. Zhang, and J. Torrellas. ReVive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 111–122, May 2002.

[99] X. Qiu and M. Dubois. Options for Dynamic Address Translation in COMAs. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 214–225, June 1998.

[100] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *Proceedings of the 12th IEEE Symposium on High-Performance Computer Architecture*, pages 13–24, February 2007.

[101] V. K. Reddy and E. Rotenberg. Coverage of a Microarchitecture-level Fault Check Regimen in a Superscalar Processor. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 1–10, June 2008.

[102] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–337, 1994.

[103] Renesas Technologies. Renesas Microcomputers. General Presentation. Online, 2008. `http://documentation.renesas.com/eng/products/mpumcu/rej13b0001_mcu.pdf`.

[104] B. F. Romanescu, A. R. Lebeck, and D. J. Sorin. Specifying and Dynamically Verifying Address Translation-Aware Memory Consistency. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, March 2010.

[105] B. F. Romanescu, A. R. Lebeck, D. J. Sorin, and A. Bracy. UNified Instruction/Translation/Data (UNITD) Coherence: One Protocol to Rule Them All. In *Proceedings of the 15th International Symposium on High-Performance Computer Architecture*, pages 199–210, January 2010.

[106] B. F. Romanescu and D. J. Sorin. Core Cannibalization Architecture: Improving Lifetime Chip Performance for Multicore Processors in the Presence of Hard Faults. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 43–51, October 2008.

[107] B. Rosenburg. Low-synchronization Translation Lookaside Buffer Consistency in Large-scale Shared-memory Multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 137–146, December 1989.

[108] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end Arguments in System Design. *ACM Transactions on Computer Systems*, 2(4):277–288, 1984.

[109] S. Sarangi, A. Tiwari, and J. Torrellas. Phoenix: Detecting and Recovering from Permanent Processor Design Bugs with Programmable Hardware. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, December 2006.

[110] S. K. Sastry Hari, M. Li, P. Ramachandran, B. Choi, and S. V. Adve. mSWAT: Low-cost Hardware Fault Detection and Diagnosis for Multicore Systems. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 122–132, December 2009.

[111] E. Schuchman and T.N. Vijaykumar. Rescue: A Microarchitecture for Testability and Defect Tolerance. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 160–171, June 2005.

[112] M. Shah, J. Barreh, J. Brooks, R. Golla, G. Grohoski, N. Gura, R. Hetherington, P. Jordan, M. Luttrell, C. Olson, B. Saha, D. Sheahan, L. Spracklen, and A. Wynn. UltraSPARC T2: A Highly-Threaded, Power-Efficient, SPARC SoC. In *Proceedings of the IEEE Asian Solid-State Circuirts Conference*, November 2007.

[113] P. Shivakumar, S. W. Keckler, C. R. Moore, and D. Burger. Exploiting Microarchitectural Redundancy For Defect Tolerance. In *Proceedings of the 21st International Conference on Computer Design*, pages 481–488, October 2003.

[114] S. Shyam, K. Constantinides, S.Phadke, V. Bertacco, and T. M. 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*, pages 73–82, October 2006.

[115] J. Smith and R. Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann Publishers Inc., 2005.

[116] D. J. Sorin, M. M.K. Martin, M. D. Hill, and D. 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*, pages 123–134, May 2002.

[117] L. Spainhower and T. A. Gregg. IBM S/390 Parallel Enterprise Server G5 Fault Tolerance: A Historical Perspective. *IBM Journal of Research and Development*, 43(5/6), September/November 1999.

[118] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. The Case for Lifetime Reliability-Aware Microprocessors. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 276–287, June 2004.

[119] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. Exploiting Structural Duplication for Lifetime Reliability Enhancement. *SIGARCH Computer Architecture News*, 33(2):520–531, 2005.

[120] J. Srinivasan, S.V. Adve, P. Bose, and J.A. Rivers. The Impact of Technology Scaling on Lifetime Reliability. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 177–186, June 2004.

[121] R. Stets, H. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott. Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 170–183, 1997.

[122] Y. Sugure, T. Seiji, A. Yuichi, Y. Hiromichi, H. Kazuya, T. Akihiko, H. Kesami, K. Takeshi, and S. Takanori. Low-Latency Superscalar and Small-Code-Size Microcontroller Core for Automotive, Industrial, and PC-Peripheral Applications. *IEICE Transactions on Electronics*, E89-C(6), June 2006.

[123] Synopsys Inc. Design Compiler Technology Backgrounder. Online, April 2006. `http://www.synopsys.com/products/logic/design_comp_tb.pdf`.

[124] M. Talluri and M. D. Hill. Surpassing the TLB Performance of Superpages With Less Operating System Support. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 171–182, October 1994.

[125] P. J. Teller. Translation-Lookaside Buffer Consistency. *IEEE Computer*, 23(6):26–36, June 1990.

[126] P. J. Teller, R. Kenner, and M. Snir. TLB Consistency on Highly-Parallel Shared-Memory Multiprocessors. In *Proceedings of the 21st Annual Hawaii International Conference on Architecture Track*, pages 184–193, 1988.

[127] I. Wagner, V. Bertacco, and T. Austin. Shielding Against Design Flaws with Field Repairable Control Logic. In *Proceedings of the Design Automation Conference*, pages 344–347, July 2006.

[128] D. L. Weaver and T. Germond, editors. *SPARC Architecture Manual (Version 9)*. PTR Prentice Hall, 1994.

[129] N. H. E. Weste and K. Eshraghian. *Principles of CMOS VLSI Design: A Systems Perspective*. Addison-Wesley Longman Publishing Co. Inc., 1985.

[130] S. B. Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: An Operating System for Many Cores. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, December 2008.

[131] A. Wolfe. AMD's Quad-Core Barcelona Bug Revealed. *InformationWeek*, December 11, 2007.

[132] D. A. Wood, S. J. Eggers, G. Gibson, M. D. Hill, and J. M. Pendleton. An In-Cache Address Translation Mechanism. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 358–365, June 1986.

[133] D. A. Wood and R. H. Katz. Supporting Reference and Dirty Bits in SPUR's Virtual Address Cache. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 122–130, May 1989.

[134] H. Zhong, S.A. Lieberman, and S.A. Mahlke. Extending Multicore Architectures to Exploit Hybrid Parallelism in Single-thread Applications. In *Proceedings of the 13th IEEE International Symposium on High Performance Computer Architecture*, pages 25–36, February 2007.

Biography

Bogdan Florin Romanescu was born on October 9th, 1980 in Iaşi, Romania. He received his B. Eng. summa cum laude, Valedictorian, in automatic control and computer engineering from "Gh. Asachi" Technical University of Iaşi in 2005. He earned a M. Sc. degree in electrical and computer engineering from Duke University in 2007. He received his Ph.D. in electrical and computer engineering from Duke University in 2010. He is the recipient of an Excellence Fellowship in 2004 and 2005.

Selected Publications

- B. F. Romanescu, A. R. Lebeck, and D. J. Sorin. Specifying and Dynamically Verifying Address Translation-Aware Memory Consistency. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, March 2010.
- B. F. Romanescu, A. R. Lebeck, D. J. Sorin, A. Bracy. UNified Instruction/Translation/Data (UNITD) Coherence: One Protocol to Rule Them All, In *Proceedings of the 15th International Symposium on High-Performance Computer Architecture*, pages 199-210, January 2010.
- B. F. Romanescu and D. J. Sorin. Core Cannibalization Architecture: Improving Lifetime Chip Performance for Multicore Processors in the Presence of Hard Faults. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 43-51, October 2008.
- B. F. Romanescu, M. E. Bauer, D. J. Sorin, S. Ozev. Reducing the Impact of Intra-Core Process Variability with Criticality-Based Resource Allocation and Prefetching, In *Proceedings of the 5th ACM International Conference on Computing Frontiers*, pages 129-138, May 2008.
- B. F. Romanescu, M. E. Bauer, S. Ozev, D. J. Sorin. VariaSim: Simulating Circuits and Systems in the Presence of Process Variability. *Computer Architecture News*, 35(5):45-48, December 2007.
- B. F. Romanescu, M. E. Bauer, D. J. Sorin, S. Ozev. Reducing the Impact of Process Variability with Prefetching and Criticality-Based Resource Allocation. Poster and extended abstract in *Proceedings of the 16th International Conference on Parallel Architectures and Compilation Techniques*, page 424, September 2007.
- B. F. Romanescu, M. E. Bauer, D. J. Sorin, S. Ozev. A Case for Computer Architecture Performance Metrics that Reflect Process Variability. Duke University, Dept. of Electrical and Computer Engineering, Technical Report #2007-2, May 2007.
- B. F. Romanescu, S. Ozev, D. J. Sorin. Quantifying the Impact of Process Variability on Microprocessor Behavior. In *Proceedings of the 2nd Workshop on Architectural Reliability*, December 2006.