Appears in the 16th International Symposium on High-Performance Computer Architecture (HPCA)
Bangalore, India, January, 2010

UNified Instruction/Translation/Data (UNITD) Coherence: One Protocol to Rule Them All

Bogdan F. Romanescu, Alvin R. Lebeck, Daniel J. Sorin
Duke University
bfr2@ee.duke.edu, alvy@cs.duke.edu, sorin@ee.duke.edu

Anne Bracy
Intel Corporation
anne.c.bracy@intel.com

Abstract

We propose UNITD, a unified hardware coherence framework that integrates translation coherence into the existing cache coherence protocol. In UNITD coherence protocols, the TLBs participate in the cache coherence protocol just like the instruction and data caches, without requiring any changes to the existing coherence protocol. UNITD eliminates the need for the software TLB shootdown routine, a procedure known to be performance costly and non-scalable. We evaluate snooping and directory UNITD coherence protocols on multicore processors with 2-16 cores, and we demonstrate that UNITD reduces the performance penalty associated with TLB coherence to almost zero.

1. Introduction

Shared memory multiprocessors, including multicore processors, have many caches, and these caches must be kept coherent. For caches that hold instructions or data, coherence is almost invariably 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.*, translation lookaside buffers), 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 [6].

This dichotomy between using hardware for cache coherence[1] and software for TLB coherence inspires two questions. First, why is cache coherence performed in hardware? Second, why is TLB coherence performed in software? Our answers to these questions lead us to conclude that the time is right to move TLB coherence into hardware.

We begin by exploring why cache coherence is performed in hardware, and we discover two primary reasons: performance and microarchitectural decoupling. Performance-wise, hardware is far faster than software, and 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 [29]. In contrast, activities with global coordination are painfully slow when performed in software. For example, Laudon [23] 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 [22]. Furthermore, hardware coherence enables migrating execution state between cores for performance, thermal, or reliability purposes [10, 19] 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 [6] explains that "the low overhead of maintaining TLB consistency in software on current machines may not justify a complete hardware implementation." As we show in Section 3, this conclusion is likely to change for future many-core chips.

1. We will use "cache coherence" as shorthand for referring to coherence for instruction and data caches.

In this paper, we argue that the time has come for hardware TLB coherence. Hardware TLB coherence provides three primary benefits. First, it drastically reduces the performance impact of TLB coherence. This performance benefit is worthwhile on its own, but it also lowers the threshold for adopting features that incur a significant amount of TLB coherence activity, including: hardware transactional memory (*e.g.*, XTM [13]), user-level memory management for debugging [14], and concurrent garbage collection [12]. 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 [38]. 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 [26, 32].

We propose UNified Instruction/Translation/Data (UNITD) Coherence, a hardware coherence framework that integrates translation coherence into the existing cache coherence protocol. At a high level, the TLBs participate in the cache coherence protocol just like instruction and data caches. UNITD eliminates the need for the software TLB shootdown routine, a procedure known to be performance costly [15, 23, 34]. UNITD is more general than the only prior work in hardware TLB coherence [37], which required specific assumptions about allowable translation caching (*e.g.*, copy-on-write is disallowed). We evaluate snooping and directory UNITD coherence protocols on multicore processors with 2-16 cores using Simics [25]. We 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.

This paper is organized as follows. Section 2 describes the problem of TLB coherence and how TLB shootdown works. Section 3 explores the performance impact of TLB shootdowns. Section 4 describes UNITD, our proposed mechanism for maintaining translation coherence. In Section 5, we discuss implementation issues, including platform-specific issues and optimizations. We experimentally evaluate UNITD in Section 6. We discuss related work in Section 7 and conclude in Section 8.

2. Address Translation

Address translation is a level of indirection that regulates a thread's access to physical memory given a virtual address. Architectures facilitate this indirection by supporting a set of software managed structures called page tables. Each page table entry (PTE) contains a mapping from a virtual memory sub-space to a physical memory sub-space, as well as additional bits representing metadata associated with the mapping (*e.g.*, protection or status bits).

Because accessing a translation can be on the critical path of a memory access, copies of the translations are cached in TLBs. 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.

2.1 Address Translation Coherence

Maintaining coherence between the TLBs and the page tables has historically been named "TLB consistency" [37], but in this paper 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 architectures do not require maintaining TLB coherence for each datum (*i.e.*, TLBs may contain different values). These architectures require TLB coherence only for unsafe changes [36] made to address translations. Unsafe changes include mapping modifications, decreasing the page privileges (*e.g.*, from read-write to read-only) or 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 another 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 can be lazily updated when the core attempts to execute a store instruction; an access violation will occur and the page fault handler can load the updated translation.

2.2 TLB Shootdown

TLB shootdown [6, 11, 33] is a software routine for enforcing TLB coherence using inter-processor interrupts. TLB shootdown is the most common technique for maintaining TLB coherence; we defer a discussion of alternative techniques to Section 7. Because managing the virtual memory 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 3.

TLB position. TLBs can be placed at different levels between the core and the memory [30]. Most microar-

Initiator	Victim
• disable pre-emption 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 translations 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 pre-emption	

Figure 1. TLB shootdown routines for initiator and victim processors

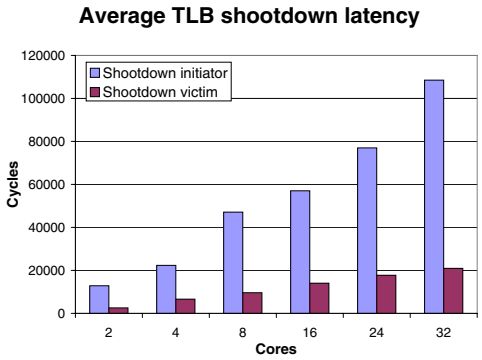


Figure 2. Per-shootdown latency

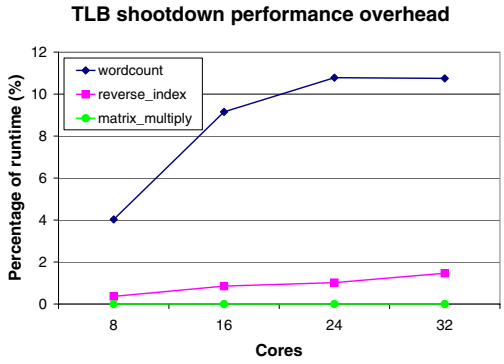


Figure 3. Shootdown performance overhead on Phoenix benchmarks

chitectures implement per-core TLBs associated with virtually-indexed physically-tagged caches. These designs pose scalability problems for many-core systems, because the performance penalty for the shootdown initiator increases with the number of victim processors. Because this solution is most common, throughout the paper we also assume per-core TLBs unless otherwise mentioned. Another option is to position the TLB at the memory [37], 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 [8, 9].

Shootdown algorithm. The TLB shootdown procedure can be implemented using various algorithms that trade complexity for performance. Teller's study [36] is an excellent description of various shootdown algorithms. In this paper, we assume the TLB shootdown procedure implemented in Linux kernel 2.6.15 and described in Figure 1. The shootdown is triggered by one processor (*i.e.*, initiator) that programs an inter-processor interrupt to determine all other processors sharing the same address space (*i.e.*, victims) to invalidate the translation(s). 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) [33]. The algorithm thus reduces the time spent by victims in the shootdown interrupt.

3. Performance Impact of TLB Coherence

In this section, we analyze the penalty associated with TLB shootdown routines and its impact on the performance of real applications. We perform these experiments on a real (not simulated) 32-Xeon processor system 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.

Figure 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 measured the latency by reading the timestamp counter at the beginning and end of the shootdown. As described in Section 2, the latency of a shootdown is application-independent and depends on microarchitectural characteristics, the number of processors involved, and the OS. Figure 2 shows that the latency of a shootdown increases roughly linearly with the number of proces-

sors involved. This latency does not capture the side effects of TLB shootdowns such as the TLB invalidations that result in extra cycles spent in repopulating the TLB with translations after the shootdown. This additional cost depends on the thread's memory footprint, as well as the position of the corresponding cache blocks in the memory hierarchy. For an Intel64 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.

Our second experiment analyzes the impact of TLB shootdowns on real applications. For this study we chose three benchmarks from the Phoenix suite [31] that cover a wide range in terms of the number of TLB shootdowns incurred within a given amount of application code. We use Oprofile [24] to estimate the percent of total runtime (*i.e.*, fraction of Oprofile's samples) spent by the applications in TLB shootdowns. Figure 3 shows the increased runtime associated with the TLB shootdowns, which becomes significant for applications that use the routine more often (*e.g.*, wordcount).

These experiments allow us to make two important observations. First, 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 paper is the latter, by maintaining TLB coherence in hardware. The second observation supported by Figure 3 is that there can be large variations in the usage patterns of TLB shootdowns across applications. As such, we will evaluate UNITD across a wide range of shootdown frequencies.

4. UNITD Coherence

In this section we present UNITD, a framework for unifying TLB coherence with cache coherence in one hardware protocol. At a high level, UNITD integrates the TLBs into the existing cache coherence protocol that uses a subset of the typical MOSI coherence states. TLBs are simply additional caches that participate in the coherence protocol like coherent, read-only instruction caches. UNITD has no impact on the cache coherence protocol and thus does not increase its complexity.

With respect to the coherence protocol, TLB entries are read-only—translations are never modified in the TLBs themselves—and thus only two coherence states are possible: Shared (read-only) and Invalid. Because there are only two possible states, UNITD uses the existing Valid bit to maintain an entry's coherence state. 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 the TLB receives a coherence message invalidating 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.

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 blocks, but a translation cached in a TLB 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). 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. 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.
- 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.

4.1 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 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 4 illustrates the translation, on an IA32 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

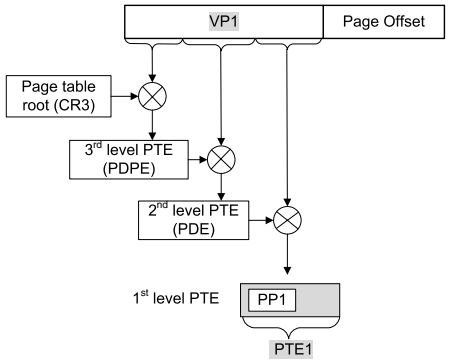


Figure 4. 3-level page table walk on IA32. UNITD associates PTE1 with the VP1->PP1 translation

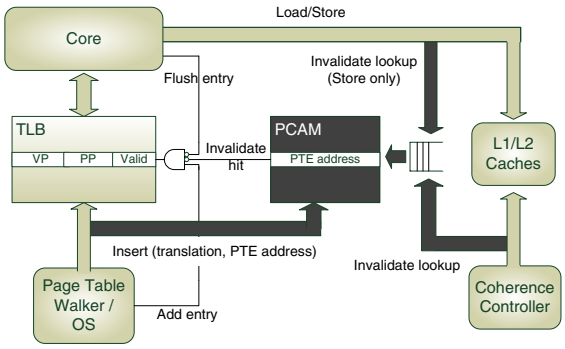


Figure 5. PCAM integration with core and coherence controller

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. 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.

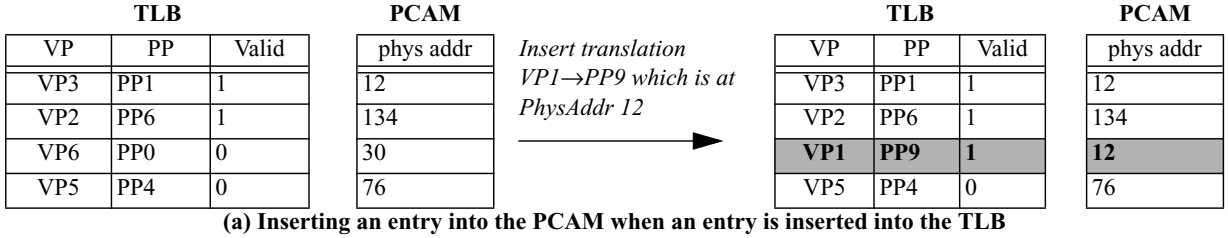
Implementation. How the last-level PTE's physical address is identified depends on whether the architecture assumes hardware or software management of TLB fills and evictions. Designs with hardware-managed TLBs rely on dedicated hardware ("page table walker") that walks iteratively through the page table 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 tables) 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.

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 the rest of the paper we assume a system with hardware-managed TLBs, but UNITD is equally applicable to systems with software-managed TLBs.

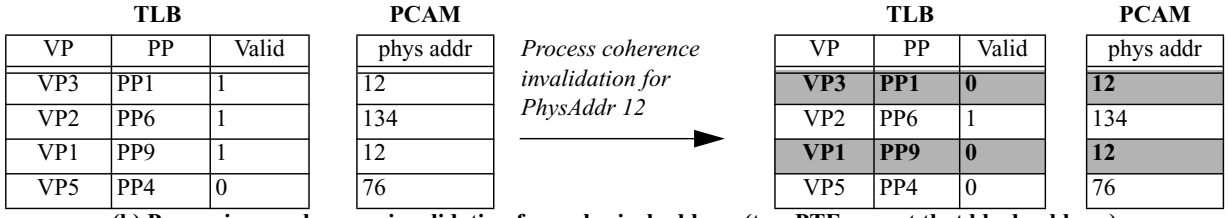
4.2 Issue #2: Augmenting the TLBs to Enable Access Using a PTE's Physical Address

Concept. To perform coherence lookups in the TLBs, UNITD needs to be able to (a) access the TLBs with physical addresses and (b) invalidate the translations associated with the PTEs that reside at those physical addresses, if any. Assuming the one-to-one correspondence between translations and PTEs discussed in Section 4.1, a TLB translation moves to Invalid whenever the PTE defining the translation is modified.

Implementation. To record the physical addresses of PTEs associated with the translations cached by 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 determined by the PTE's physical address). Figure 5 shows how the PCAM is integrated into the system, with interfaces 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 5.2). The PCAM is off the critical path of a memory access; it is not accessed during regular TLB lookups for obtaining translations.



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



(b) Processing a coherence invalidation for a physical address (two PTEs are at that block address)

Figure 6. PCAM operations

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 [18], for example. Henceforth, we assume a physical CAM implementation, without loss of generality.

Maintaining coherence on physical addresses of PTEs requires book-keeping at a fine granularity (*e.g.*, double-word for a 32-bit architecture). In order to integrate 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. Because multiple PTEs can be placed in the same cache block, the PCAM can hold multiple copies of the same datum. For simplicity, throughout the rest of the paper we refer to PCAM entries simply as PTE addresses.

Figure 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 6a). Note that there can be multiple PCAM entries with the same physical address, as in Figure 6a; 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 6b illustrates a coherence invalidation of physical address 12 that hits in two PCAM entries.

5. Platform-Specific Issues, Implementation Issues, and Optimizations

In this section, we analyze several implementation issues, including: UNITD's integration with speculative execution in superscalar cores (Section 5.1), UNITD's handling of translations that are currently in both the TLB and data cache of a given core (Section 5.2), how UNITD is compatible with a wide range of system models and features (Section 5.3), and a method of reducing the number of TLB coherence lookups (Section 5.4).

5.1 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, these situations require that the load (and its consumers) be squashed.[1] To detect these mis-speculations, cores adopt one of two solutions [17]: 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 event. 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.

5.2 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. We present two 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 to 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. Because Solution #1 requires 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 can ensure that the TLB always contains coherent data by probing the TLB on stores by the local core.[2] This situation is the only case in which UNITD allows a combination of seemingly incompatible coherence states. Because processors already provide mechanisms for self-snoops on stores for supporting self-modifying code [21], UNITD can take advantage of existing resources, which is why we have chosen Solution #2 over Solution #1.

5.3 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.

5.3.1 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 5.2, 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.

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.

5.3.2 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.

5.3.3 Virtual Address Synonyms

UNITD is not affected by synonyms (virtual address aliases) because it operates on PTEs that define unique 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 in the synonym set.

1. We consider a sequentially consistent system in this paper.

2. In Section 5.4, we discuss how to minimize the number of TLB snoops for this purpose.

5.3.4 Superpages

Superpages rely on "coalescing neighboring PTEs into superpage mappings if they are compatible" [35]. 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).

5.3.5 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.

5.4 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 [28].

6. Experimental Evaluation

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

6.1 Methodology

We use Virtutech Simics [25] to simulate an x86 multicore processor. For the memory system timing simulations we use GEMS [27]. We extend the infrastructure to accurately model page table walks and TLB accesses. We do not model the time to deliver interrupts, an approximation that aids systems with shootdowns but not UNITD. The parameters of our simulated system are given in Table 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. We report results averaged across twenty simulated executions, with each simulation having a randomly perturbed main memory latency as described by Alameldeen *et al.* [2].

Table 1. Target System Parameters

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 2. Microbenchmarks

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

6.2 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. With the exception of the wordcount benchmark from the Phoenix suite [31], we are unaware of existing benchmarks that exercise TLB shootdown mechanisms.[1] 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 mmapped with corresponding flags set) or unmapping pages. 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 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 effi-

1. Few benchmarks are designed to use TLB shootdowns. Rather, most benchmarks are designed to stress cores and caches.

ciency 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 3. In our experiments, we vary the number of shootdowns between 0 and 12000. Varying the number of TLB shootdowns reveals the benefits of UNITD as well as creating a correspondence between the possible benefits and the original time spent in shootdowns.

In addition to these microbenchmarks, we study UNITD's performance on applications that exhibit no shootdowns, including swaptions from the Parsec suite [5] and pca, string-match, and wordcount (with a much smaller input file than the one used in Figure 3, due to simulator limitations, leading to a negligible number of shootdowns) from the Phoenix suite [31]. We perform these experiments to confirm that UNITD does not degrade common-case performance.

6.3 Performance

In all performance experiments, we compare UNITD to two systems. The first comparison is 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).

Single_unmap. Figure 7 shows UNITD's performance on a directory system on the single_unmap benchmark[1] as a function of the number of shootdowns and number of cores. 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. Second, UNITD speedups increase with the number of TLB shootdowns and with the number of cores. If the shootdown count is large, the

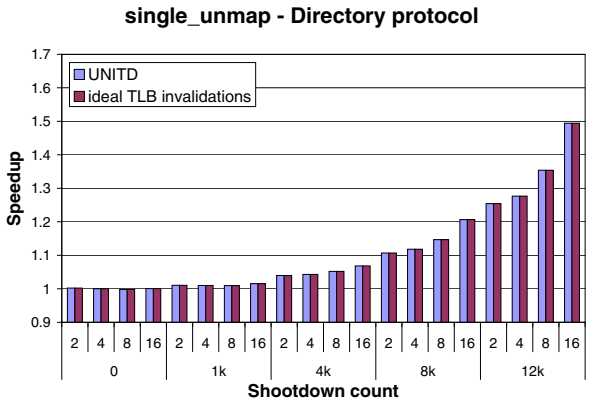


Figure 7. single_unmap benchmark. UNITD speedup over baseline system for directory

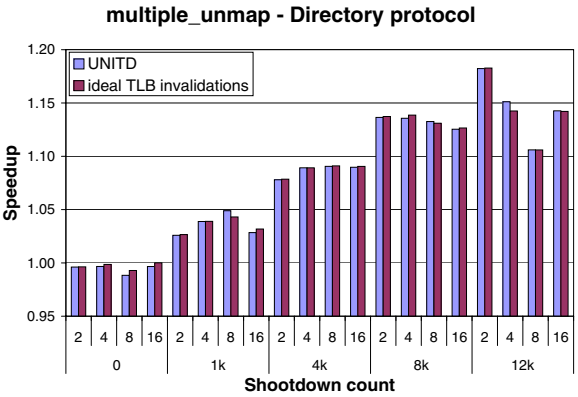


Figure 8. multiple_unmap benchmark. UNITD speedup over baseline system for directory

performance benefits scale accordingly, up to 68% speedup for the 16 cores configuration. In addition, even for the same number of shootdowns, UNITD's improvements increase with the increasing number of cores. For 4000 shootdowns, UNITD 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.

Multiple_unmap. Figure 8 shows the performance when there are multiple threads unmapping the pages for directory systems. 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. We observe small speedup/slowdowns for the executions

1. Due to space constraints, we discuss the results for the snooping system, but we omit graphing the results, which are qualitatively similar to those for the directory system.

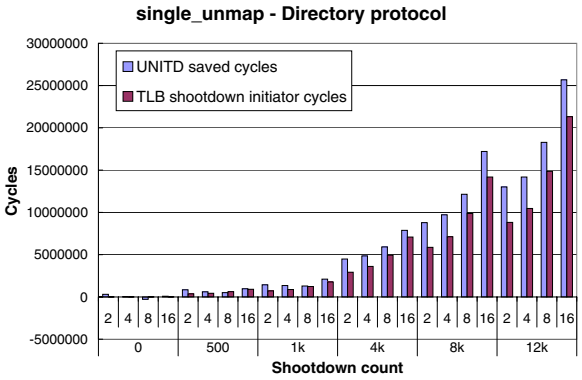


Figure 9. Runtime cycles eliminated by UNITD for single_unmap with directory protocol.

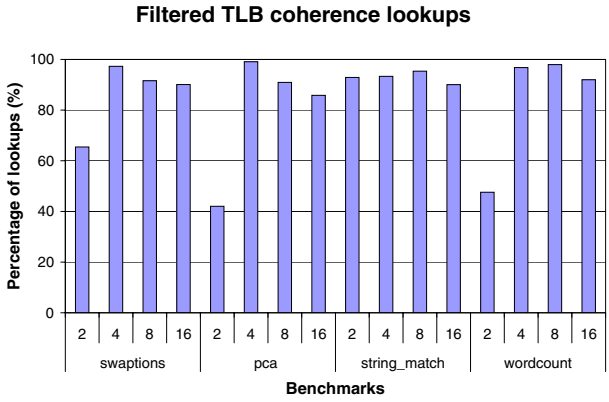


Figure 10. Percentage of TLB coherence lookups filtered with a simple Jetty filter

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.

Single_cow[1]. 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. 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 the worker thread spends in TLB shootdown initiation, as in single_cow, but, for multiple_cow, performance is also influenced by the time to service TLB shootdown interrupts triggered by other threads. The interrupt servicing routine has a significant impact on performance that increases with the number of cores as shown in Section 3. For multiple_cow, performance is greatly affected by TLB shootdowns, especially for 16 cores. In this case, UNITD outperforms the base case by up to 20% for the snooping protocol.

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. 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.

1. Due to space constraints, we discuss results for the COW microbenchmark but omit graphing the results.

Understanding UNITD's Performance Benefit. To better understand the performance benefits of UNITD, Figure 9 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 3, 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.

6.4 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 5.4, 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 [28]. 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 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 10 shows.

6.5 Hardware and Power Costs

The hardware and power costs associated with UNITD are 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 mentioned in Section 4.2.

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. 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.

7. Related Work

Section 2.1 described the software TLB shootdown routine as the most common technique of maintaining TLB coherence. Previous research 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 TLB 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 [16] and K42 [3] introduce the concept of clustered objects that are associated with each thread, thus reducing the contention of the kernel managed resources. Corey [7] follows the same concept by giving applications the power to decide which PTEs will be processor-private and thus eliminate shootdowns for these PTEs.

Alternative TLB coherence mechanisms. Teller has proposed several hardware-based mechanisms for handling TLB coherence [37], but they restrict the system model in significant ways, such as prohibiting the copy-on-write policy. Wood *et al.* [39] 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 [40]. The design also complicates the handling of virtual memory based optimizations such as concurrent garbage collection or copy-on-write [4].

8. Conclusions

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.

Acknowledgments

This work has been supported in part by the Semiconductor Research Corporation under contract 2009-HJ-1881 and the National Science Foundation under grant CCR-0444516. We thank Milo Martin for his helpful feedback on this work.

References

[1] A. Agarwal *et al.* The MIT Alewife Machine: Architecture and Performance. In *Proc. of the 22nd Annual Int'l Symposium on Computer Architecture*, June 1995.

[2] A. R. Alameldeen *et al.* Evaluating Non-deterministic Multi-threaded Commercial Workloads. In *Proc. of the Fifth Workshop on Computer Architecture Evaluation Using Commercial Workloads*, Feb. 2002.

[3] J. Appavoo *et al*. Experience Distributing Objects in an SMMP OS. *ACM Trans. Computer Systems*, 25(3), 2007.

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

[5] C. Bienia et al. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proc. of the International Conference on Parallel Architectures and Compilation Techniques*, Oct. 2008.

[6] D. L. Black *et al*. Translation Lookaside Buffer Consistency: A Software Approach. In *Proc. Third Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, Apr. 1989.

[7] S. Boyd-Wickizer *et al*. Corey: An Operating System for Many Cores. In *Proc. of the 8th USENIX Symposium on*

Operating Systems Design and Implementation, Dec. 2008.

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

[9] M. Cekleov and M. Dubois. Virtual-Address Caches, Part 2: Multiprocessor Issues. *IEEE Micro*, 17(6), Nov. 1997.

[10] K. Chakraborty, P. M. Wells, and G. S. Sohi. Computation Spreading: Employing Hardware Migration to Specialize CMP Cores On-the-Fly. In *Proc. of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2006.

[11] M.-S. Chang and K. Koh. Lazy TLB Consistency for Large-Scale Multiprocessors. In *Proc. of the 2nd Aizu International Symposium on Parallel Algorithms / Architecture Synthesis*, Mar. 1997.

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

[13] J. Chung *et al*. Tradeoffs in Transactional Memory Virtualization. In *Proc. of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2006.

[14] D. Dhurjati and V. Adve. Efficiently Detecting All Dangling Pointer Uses in Production Servers. In *Proc. of the International Conference on Dependable Systems and Networks*, June 2006.

[15] 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.

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

[17] K. Gharachorloo, A. Gupta, and J. Hennessy. Two Techniques to Enhance the Performance of Memory Consistency Models. In *Proc. of the International Conference on Parallel Processing*, volume I, Aug. 1991.

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

[19] S. Heo, K. Barr, and K. Asanovic. Reducing Power Density Through Activity Migration. In *Proc. of the 2003 Int'l Symp. on Low Power Electronics and Design*, 2003.

[20] M. D. Hill *et al.* Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessor. *ACM Trans. on Computer Systems*, 11(4), Nov. 1993.

[21] Intel Corporation. *Intel Processor Identification and the CPUID Instruction*, Mar. 2009.

[22] R. Kumar *et al.* Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *Proc. of the 36th Annual International Symposium on Microarchitecture*, Dec. 2003.

[23] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proc. of the 24th Annual Int'l Symposium on Computer Architecture*, June 1997.

[24] J. Levon *et al.* Oprofile. http://oprofile.sourceforge.net.

[25] P. S. Magnusson *et al.* Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2), Feb. 2002.

[26] K. Magoutis. Memory Management Support for Multi-Programmed Remote Direct Memory Access (RDMA) Systems. In *Proc. of the IEEE International Conference on Cluster Computing*, Sept. 2005.

[27] M. M. Martin *et al.* Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *Computer Architecture News*, 33(4), Sept. 2005.

[28] A. Moshovos, G. Memik, A. Choudhary, and B. Falsafi. JETTY: Filtering Snoops for Reduced Energy Consumption in SMP Servers. In *Proc. of the Seventh IEEE Symposium on High-Performance Computer Architecture*, Jan. 2001.

[29] U. G. Nawathe *et al*. Implementation of an 8-Core, 64-Thread, Power-Efficient SPARC Server on a Chip. *IEEE Journal of Solid-State Circuits*, 43(1), 2008.

[30] X. Qiu and M. Dubois. Options for Dynamic Address Translation in COMAs. In *Proc. of the 25th Annual International Symposium on Computer Architecture*, June 1998.

[31] C. Ranger *et al.* Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *Proc. of the Twelfth International Symposium on High-Performance Computer Architecture*, Feb. 2007.

[32] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proc. of the 21st Annual International Symposium on Computer Architecture*, Apr. 1994.

[33] B. Rosenburg. Low-synchronization Translation Lookaside Buffer Consistency in Large-scale Shared-memory Multiprocessors. In *Proc. of the Twelfth ACM Symposium on Operating Systems Principles*, 1989.

[34] R. Stets *et al.* Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network. In *Proc. of the 16th ACM Symposium on Operating Systems Principles*, 1997.

[35] M. Talluri and M. D. Hill. Surpassing the TLB Performance of Superpages With Less Operating System Support. In *Proc. of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1994.

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

[37] P. J. Teller, R. Kenner, and M. Snir. TLB Consistency on Highly-Parallel Shared-Memory Multiprocessors. In *Proc. of the Twenty-First Annual Hawaii International Conference on Architecture Track*, 1988.

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

[39] D. A. Wood *et al.* An In-Cache Address Translation Mechanism. In *Proc. of the 13th Annual International Symposium on Computer Architecture*, June 1986.

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