# DiDi: Mitigating The Performance Impact of TLB Shootdowns Using A Shared TLB Directory

Carlos Villavieja[1,2], Vasileios Karakostas[1,2], Lluis Vilanova[1,2], Yoav Etsion[2], Alex Ramirez[1,2], Avi Mendelson[4], Nacho Navarro[1,2], Adrián Cristal[1,3] and Osman S. Unsal[2]

[1]Computer Architecture Department
Universitat Politècnica de Catalunya (UPC)
Barcelona, Spain

[2]Barcelona Supercomputing Center
Barcelona, Spain

[3]IIIA - Artificial Intelligence Research Institute
CSIC - Spanish National Research Council
Spain

[4] Microsoft R&D
Israel

first.lastname@bsc.es, avim@microsoft.com

***Abstract*—*Translation Lookaside Buffers* (TLBs) are ubiquitously used in modern architectures to cache virtual-to-physical mappings and, as they are looked up on every memory access, are paramount to performance scalability. The emergence of chip-multiprocessors (CMPs) with per-core TLBs, has brought the problem of TLB coherence to front stage.**

**TLBs are kept coherent at the software-level by the operating system (OS). Whenever the OS modifies page permissions in a page table, it must initiate a coherency transaction among TLBs, a process known as a *TLB shootdown*. Current CMPs rely on the OS to approximate the set of TLBs caching a mapping and synchronize TLBs using costly Inter-Proceessor Interrupts (IPIs) and software handlers.**

**In this paper, we characterize the impact of TLB shootdowns on multiprocessor performance and scalability, and present the design of a scalable TLB coherency mechanism.**

**First, we show that both TLB shootdown cost and frequency increase with the number of processors and project that software-based TLB shootdowns would thwart the performance of large multiprocessors. We then present a scalable architectural mechanism that couples a shared TLB directory with load/store queue support for lightweight TLB invalidation, and thereby eliminates the need for costly IPIs. Finally, we show that the proposed mechanism reduces the fraction of machine cycles wasted on TLB shootdowns by an order of magnitude.**

## I. Introduction

Virtual memory has long been the standard mechanism that guarantees memory protection between applications. Protection is provided through an abstract linear address space. The decoupling of application-level memory addressing from the processor's physical address space is typically implemented using hierarchical *page tables* that map the virtual page addresses to their corresponding physical pages, as well as store page access permissions. A virtual-to-physical mapping is obtained by traversing the hierarchical page table, commonly referred to as a *page table walk*. This process requires a number of memory accesses that corresponds to the depth of the page table (modern Intel and AMD processors employ page tables with 4 levels). Page tables are managed by the operating system, which controls the physical memory resources.

Fast mapping of virtual addresses to their associated physical addresses is critical to processor performance, as it takes place on every memory operation. Therefore, all processors employ a *Translation Lookaside Buffer* (TLB), which caches address translation information in an on-chip, content-addressable memory, and thereby eliminates the need for a full page-table walk in the common case.

The performance criticality of TLBs mandates that many-core architectures include a TLB on each processing core. This replication requires that all TLBs must be kept consistent with the OS page tables, so that the information cached across all the TLBs preserves a globally consistent view of virtual memory. TLBs are read-only structures, and thus the only way to change any value in the TLB is to invalidate some entries and reload them from the page table. However, due to a lack of proper hardware support, current systems must maintain TLB coherency at the software level using *Inter-Processor Interrupts* (IPIs), in a process known as *TLB shootdown* [10]. The term refers to the coherence transaction initiated by the OS after performing page table modifications. OS code, running on the core making the modifications, sends an IPI to all cores whose TLBs might be caching a mapping affected by the modification, and thereby "shoot down" stale TLB mappings.

The overheads associated with interrupt processing make TLB shootdowns a performance bottleneck that impedes the scalability of multiprocessors. Moreover, as the OS cannot accurately track the contents of TLBs, it must conservatively approximate the set of TLBs that contain stale mappings, potentially resulting in false positives in the form of unnecessarily interrupted cores.

In this paper, we describe *DiDi* [1], a two-level TLB architecture that consists of a per-core TLB and a shared, inclusive, second-level TLB with an associated directory.

In addition, DiDi includes a dedicated per-core mechanism that provides support for invalidating TLB entries on remote

[1]DiDi is an acronym for *Dictionary Directory*, which is the semantic service provided by the second-level TLB of the processing cores. DiDi also means "tell me" in Spanish.

cores without interrupting the instruction stream they execute, thereby eliminating the need to use costly IPIs. We show that this design reduces the performance impact of TLB shootdowns by an order of magnitude.

The main contributions of this paper are the following:

- A detailed characterization of the TLB shootdown problem. We quantify the performance impact of the TLB shootdown on real machines (both Intel and AMD) and its implications on multiprocessor scalability.
- Use of a global TLB directory to eliminate false positives when establishing the set of cores that must be notified on a TLB shootdown. We evaluate the proposed design on a simulated manycore platform using several commercial benchmarks.
- Introduction of a non-intrusive remote TLB invalidation mechanism that does not rely on costly inter-processor interrupts, thereby minimizing the overhead of TLB shootdowns.

The rest of this paper is organized as follows: Section II reviews the implications of DiDi on different TLB-shootdown scenarios. We then discuss related work on TLB coherency management in Section III and describe our evaluation methodology in Section IV. Section V describes and evaluates the costs of a TLB shootdown operation. This includes a characterization and real-world analysis of the costs of all the operations and layers involved in this operation, as well as showing how the cost of this operation increases with the number of cores in a multiprocessor. Section VI describes the proposed TLB architecture, and Section VII presents its performance evaluation. Finally, we conclude in Section VIII.

## II. On The Broad Implications of TLB Shootdowns

TLB shootdowns are initiated by the OS whenever the access permissions of a page are modified. This includes a number of well-known scenarios, such as marking a page as read-only as part of a copy-on-write optimization, reclaiming physical frame when swapping memory out to disk, or simply handling system calls that affect the page-table such as *mprotect* and *munmap*.

The prohibitive cost of a TLB shootdown affects both existing multiprocessor workloads, as well as impedes the development of novel software models such as high-level parallel programming models [20], accelerator-based systems [16], and software transactional memory [1], among others. This section briefly presents several such effects.

Emerging high-level programming models, such as MapReduce [20], use filesystem-based buffering as means of interprocess communication. However, the frequent page-table modifications triggered by mapping and unmapping files trigger TLB shootdowns to synchronize all TLBs on the system. Reducing the overheads associated with TLB shootdowns can boost the scalability of these models.

In the high-performance computing (HPC) domain, systems are moving towards accelerator-based CPU/GPGPU designs, which frequently change page permissions and lock physical pages in memory to facilitate data transfers between the CPU and the accelerator [15]. These operations trigger frequent TLB shootdowns and thereby degrade the overall system performance.

Finally, strong atomicity is an important property of transactional memory (TM) systems, which guarantees that data accessed within a transaction cannot be concurrently accessed by any non-transactional code [18]. But while this property is inherent in hardware hardware TM proposals, software TM (STM) systems can only provide strong atomicity by modifying page access permissions inside a transaction. However, the prohibitive cost of TLB shootdowns, triggered by page permission changes, inhibit support for strong atomicity in STM systems [1].

In conclusion, hardware support for low-overhead TLB shootdowns is highly desirable, as its benefits will manifest across multiple software domains. We demonstrate these benefits through our selection of benchmarks (described in Section IV), which span across all the above domains.

## III. Related Work

The need for address translation coherency has motivated a number of studies aiming to reduce its associated overhead.

Rashid et al. [21] showed that consistency between the TLBs and page tables can be relaxed, and presented the concept of *lazy* TLB consistency in cases where page permissions are amplified (a common operation in copy-on-write scenarios). The key idea is that TLB consistency need not be immediately enforced. Instead, if the application wrote to a page that is writeable in the page mapping, but cached as read-only in the TLB, the OS page-fault handler can detect such a scenario, force the eviction of the TLB entry and return control to the application.

Several studies have addressed the costly effects of interrupting the execution of all processors when performing a TLB consistency operation [10], [21], [24], [26]. Black et al. [10] presented an efficient software algorithm to enforce TLB consistency in multiprocessors by avoiding a complete machine-wide TLB invalidation. The authors also argued for high-priority software interrupts and non-blocking MMU operations to make the algorithm more scalable. The Barrelfish OS [5] implemented user-level TLB shootdowns by sending messages between *monitors* — user-space components of the OS. The use of user-level communications thus eliminated the need for costly IPIs.

Jacob et al. [17] compared several virtual memory designs and TLBs, showing the relevance of TLB size to the overhead of TLB shootdowns.

Recent studies have dealt with the TLB performance on CMPs. Bhattacharjee et al. [6], [7] showed that TLB misses are predictable and that inter-core TLB cooperation and prefetching mechanisms can be applied to improve TLB performance. However, this implies that a TLB shootdown must also invalidate mappings in the TLB prefetch buffers.

Srikanraiah et al. [25] presented the concept of *Synergistic TLBs*, where a TLB is able to allocate victim entries on other TLBs during evictions, as well as migrate entries among TLBs in order to improve the overall TLB hit ratio by

replicating translations among multiple TLBs. However, the authors did not address the TLB shootdown problem; there is no information on which TLBs hold a page, so false positives still exist during a TLB shootdown, and the invalidation of remote TLB entries still uses IPIs.

Romanescu et al. [23] proposed *UNITD*, a scalable, hardware-based TLB coherence protocol. UNITD uses a bit on each TLB entry to store sharing information, thereby eliminating the use of IPIs. However, UNITD still resorts to broadcasts for invalidating shared mappings. Furthermore, UNITD adds a costly CAM (Content-Addressable Memory) to each TLB in order to perform reverse address translation when checking if a page translation is present in a specific TLB, thereby greatly increasing the TLB power consumption. Still, in terms of performance, UNITD is equivalent to DiDi, as both eliminate both the use of IPIs as well as false positive remote TLBs invalidations.

Bhattacharjee et al. [8] targetted the effective caching capacity of TLBs. They thus introduced a shared last-level TLB and evaluated the benefit of using a shared last-level TLB compared to a private second-level TLB. However, their design still relies on IPI-based coherency transactions. In addition, their proposed last-level TLB is not fully-inclusive and is thus still susceptible to invalidations with false positives.

In contrast to previous work, we emphasize the characterization and modeling of the performance impact of TLB shootdowns on existing hardware in order to estimate its impact on future large-scale multiprocessors. Moreover, our solution addresses the two main problems in TLB shootdowns: (1) we eliminate the use of IPIs for invalidating remote TLB entries, and (2) we eliminate false positives by providing a completely accurate set of cores caching specific TLB entries.

## IV. Methodology

We characterize the performance impact of TLB shootdowns on existing hardware. Specifically, we evaluate the overhead of processing a TLB shootdown, their frequency, and the number of cores affected by a single TLB shootdown. The characterization is then used to project the performance impact of TLB shootdowns on future, large-scale muticores. Finally, we evaluate DiDi using the TaskSim trace-based simulator [22], which simulates DiDi's multilevel TLB hierarchy. This methodology was selected in favour of cycle-accurate full-system simulations of a CMP system, which may require weeks or months just to evaluate a few seconds of simulated time. To determine how many false positives would happen in real applications runs, we use a PIN tool to simulate the TLB directory.

Our CMP model is summarized in Table I, consisting of 32–256 cores with private L1 caches and a shared L2. Coherence is maintained using a directory-based MSI protocol, embedded in the L2. The interconnect has a two-level ring topology, where each core is connected to a processor ring (8 cores per ring), and a global ring connects the processor rings, L2 banks, and the memory controllers.

*a) Characterization Platform:* The TLB shootdown characterization was performed on both Intel and AMD architectures, described in Table II.

| Parameter | Value |
|---|---|
| Per-core TLB | 128 entries, 4-way set-associative, 2 cycles latency |
| Shared DiDi | 4096 entries, 6 cycles latency |
| L1 | Private, 64KB, 4-way set-associative, 3 cycle latency, split D/I |
| L2 | Shared, 8MB banked design, 8-way set-associative, 22 cycles latency |
| Memory | 4 memory controllers (MC), 2 channels per MC, single 800MHz DDR3 DIMM per ch. |
| Interconnect | Segmented two-level ring, 16 bytes/cycle, 4 concurrent connections per segment |

TABLE I
Simulated system parameters.

| | Intel | AMD |
|---|---|---|
| **Processor** | Xeon E5640 | Opteron 6128 |
| **Frequency** | 2.67GHz | 2.0GHz |
| **RAM** | 8 GB | 32GB |
| **Cores** | 16 (4x 4-core chips) | |
| **Kernel** | Linux 2.6.36 | |

TABLE II
Evaluation platforms.

The Linux kernel was traced using the Linux Trace Toolkit (LTT) [14]. Specifically, we have inserted LTT markers to trace the different steps of a TLB shootdown (see Section V): locking the page table; constructing the set of affected cores; sending IPI to remote cores; executing the interrupt handler on the remote cores; acknowledging invalidations to the initiating core; and releasing the page-table lock.

*b) Applications:* The applications used for our evaluation are listed in Table III. The set consists of server applications, as well as the set of microbenchmarks used by Romanescu et al. [23]. Moreover, to stress the TLB-shootdown mechanism we implement a simplified version of the strong atomicity for STM systems as described in [1] by modifying the TL2 [12] and using the STAMP [11] applications as workloads. (as all STAMP benchmarks exhibit similar behavior with regards to TLB shootdowns, we only show results for three).

## V. TLB Shootdown Characterization

In order to model the performance impact of TLB shootdowns, we have characterized their behavior. This includes the

| **Server** | |
|---|---|
| Phoenix | MapReduce Engine [20] (*wordcount*) |
| Apache | Web Server and Test Suite [2] |
| **Transactional Memory** | |
| genome | Gene sequencing |
| vacation | Client/server travel reservation system |
| labyrinth | Maze routing |
| **Microbenchmarks** | |
| machtest | Basic cost of a TLB shootdown |
| cowsingle | Copy-on-write (single thread) |
| cowmultiple | Copy-on-write (multi-threaded) |

TABLE III
List of benchmarks used in the paper.

TLB shootdown overhead components, their frequency, and the number of cores they affect.

### A. Anatomy of a TLB Shootdown

A TLB shootdown is effectively a two-phase commit transaction in which the core modifying the page-table verifies that all cores that may be caching the affected mapping evict it from their TLBs. Figure 1 illustrates the time line of a TLB shootdown:

1) One of the cores (top timeline) begins executing an operation that modifies the page-table, triggering the OS to lock the corresponding page table entry. This core is referred to as the *initiator* core.
2) The OS builds a list of cores that requested a translation of the modified page-table entry in the past. These are referred to as *slave* cores. To this end, the OS has to continuously track the usage of memory regions on the different cores. The list of slave cores is approximated, as the translation may have already been evicted from a slave core's TLB (it might contain *false positives*).
3) The initiator sends an IPI to all the slave cores, requesting them to invalidate the TLB entries referring to the modified mapping. Meanwhile, the initiator core invalidates the mapping in the local TLB and waits for the acknowledgments from all the slave cores.
4) All slave cores are interrupted by the IPI and execute the IPI handler for TLB invalidations (switching protection levels and saving/restoring the application's execution context before and after executing the handler). The interrupt handler code invalidates any affected TLB entries and sends an acknowledge message to the initiator core.
5) Once all acknowledge messages are received by the initiator core, it unlocks the page-table entry and continues its execution.

As discussed above, the excessive overhead of a TLB shootdown stems from its usage of IPIs and the associated protection level switches. In addition, due to the lack of a proper directory mechanism, the OS can only approximate the set of slave cores, which results in false positives that end up interrupting the execution of cores that are not actually caching the affected mapping.

### B. TLB Shootdown Overhead

The first step towards understanding the system performance impact of TLB shootdowns is evaluating the direct overhead of a single shootdown on both the initiator and the slave cores. To this end, we have instrumented the Linux kernel using the Linux Tracing Toolkit [14] and measured the different overhead components.

Figure 2 depicts the breakdown of the TLB shootdown overhead for Apache and Wordcount as a function of the number of cores in the system, running on both Intel and AMD machines. As results were consistent for all applications, including the micro-benchmarks, they are only shown for Apache and Wordcount. The figure shows the breakdown of the overhead into three distinct components:

1) **Send IPI** Computing the (approximate) set of slave cores, and sending the IPIs. This component is only incurred by the initiator core.
2) **Execute Handler** Executing the interrupt handler that invalidates the modified mapping from the local TLB. This component is incurred by both the initiator and the slave cores.
3) **Wait for Ack** Waiting for an acknowledgment message from all the slave cores indicating that they have flushed the modified mapping from their TLBs, and that the initiator can release the page-table lock.

The overhead incurred by the initiator core includes all three components shown in the figure, whereas the overhead incurred by slave cores consists only of executing the TLB-invalidating IPI handler.

As expected, the overhead of sending the IPIs and waiting for acknowledgments highly depends on the number of cores in the system: the overhead for computing the set of affected cores and sending the IPIs increases by 4x as the number of cores increases from 2 to 16. Also the time spent waiting for acknowledgments increases by $\sim$10x as the number of cores increase. The reason for this difference lies in the asynchrony of the latter operation, as evident by the difference in the acknowledgment wait time between the two applications shown. This difference is a direct result of the per-core application workload, which affects the timing of executing the interrupt handler on each core.

Another interesting (and somewhat surprising) result is that the overhead incurred by the execution of the interrupt handler increases by $\sim$2x as the number of cores in the system goes from 2 to 16. This is due to memory contention caused when all slave cores read a shared datum containing the invalidation information and, more importantly, write to a bitmask in the shared datum to acknowledge the completion of the operation. The write operation causes thrashing among the coherent L1 caches and thereby increases the time to complete the handler execution. Naturally, the amount of cache thrashing and the resulting increase in memory access latency depends on the number of cores participating in the shared operation, and, as discussed in Section V-D, the number of cores involved in a TLB shootdown increases with the number of cores in the system.

Finally, in Figure 2, we can observe that for all applications and all machine sizes, the cost of a TLB shootdown is around 50% higher in AMD processors. This may be due to (1) Intel machine has a 33.5% faster clock frequency as shown in Table II, and (2) several architectural optimizations on the Intel platform, which reduce the overhead of invalidating a specific entry in the TLB, plus faster IPIs.

### C. TLB Shootdown Frequency

The frequency of TLB shootdowns is detrimental to the characterization of their impact on system performance, and is purely workload-dependent. Workloads that perform memory or file I/O, such as Apache and Wordcount, require large amounts of memory remapping and therefore trigger lots of TLB shootdowns. In contrast, workloads that do not modify

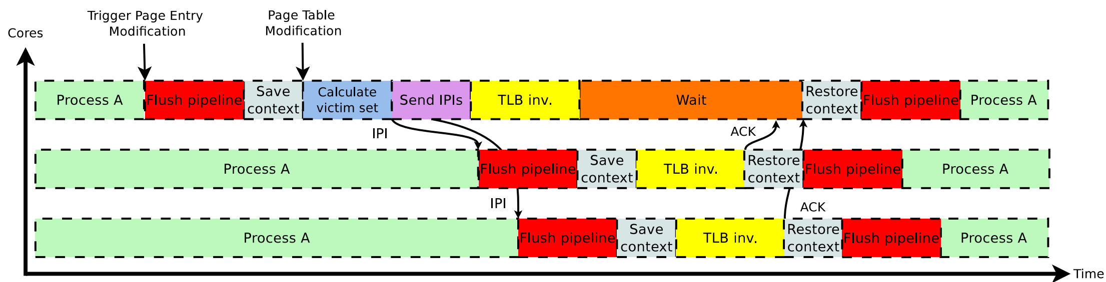
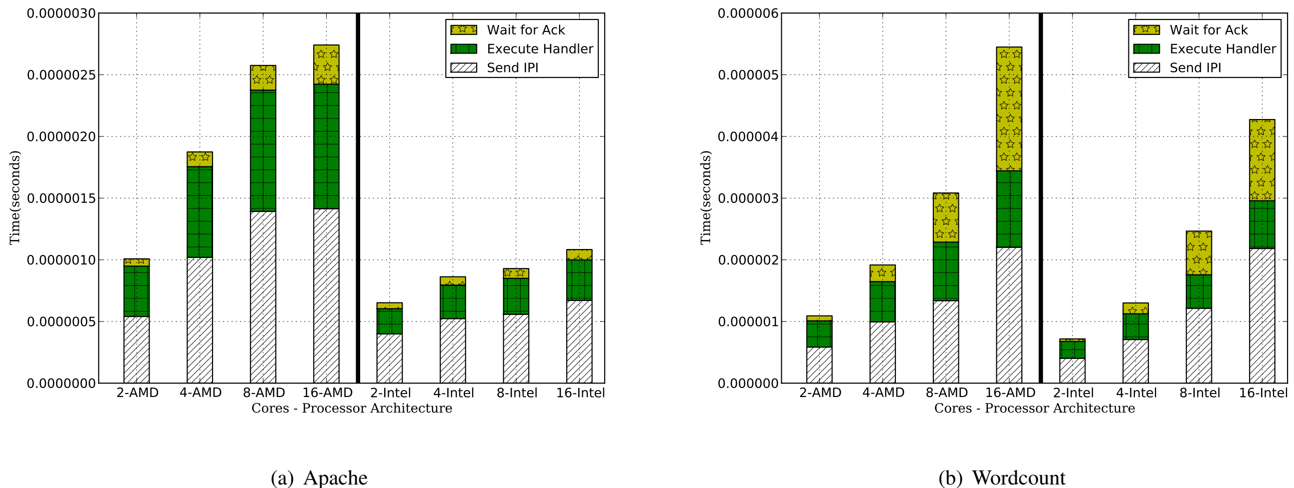



Fig. 1. Timeline of a TLB shootdown, illustrating how the timeline of the *initiator* core (top) relates to the timelines of the *slave* cores (bottom).




(a) Apache

(b) Wordcount

Fig. 2. Decomposition of the TLB shootdown overheads for Apache and Wordcount. The initiator core incurs the full overhead (shown in figure), whereas slave cores only incur the *Execute Handler* overhead.

their memory mappings, such as the PARSEC and NAS benchmarks, do not generate TLB shootdowns and are therefore hardly affected by them.

Figure 3 shows the frequency of TLB shootdowns as a function of the number of system cores for Apache and Wordcount running on Intel and AMD platforms, respectively (frequency is shown in KHz). Measurements are shown for 2, 4, 8, and 16 cores. As expected, increasing the number of system cores results in an increased frequency of TLB shootdowns for both applications, and in both cases the frequency increases between 5–10x as the number of system cores increases from 2 to 16.

The figure also shows that the frequency measurements for both applications (on both architectures) fit well on a linear regression curve. This suggests that the TLB shootdown frequency can be modeled as a linear relation, and it increases 1KHz/core for Wordcount and 200Hz/core for Apache. The reason for this linear relation stems from the throughput-oriented nature of both applications, which partition a fixed amount of work on an increasing number of cores. Moreover, the linear model can also be used to project the TLB shootdown frequency for larger multiprocessors, as depicted in the figure.

Finally, the figure also demonstrates that the TLB shootdown frequency is application-dependent rather than architecture dependent, as the differences between the Intel and AMD systems are small.

### *D. Number of Slave Cores*

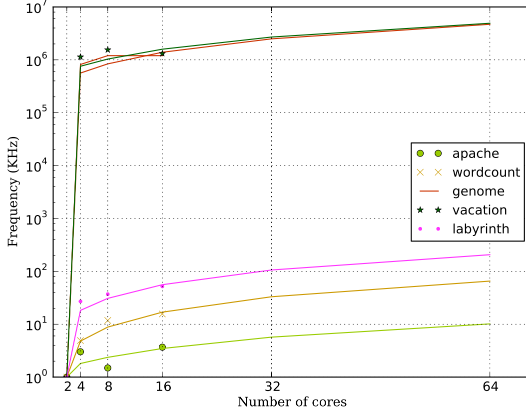
Figure 4 depicts the average number of slave cores in a TLB shootdown as a function of the number of cores in the system, for both Wordcount and Apache, running on both Intel and AMD architectures. The figure clearly shows that the number of cores affected by a TLB shootdown increases with the number of cores in the system and that, once again, the linear regression model is a good fit for the resulting curve.

Furthermore, the figures also show the (in)effectiveness of an OS-level minimization of the set of slave cores in a TLB shootdown. The OS only manages to exclude ∼30% of the cores in the system for Apache-triggered TLB shootdowns, while Wordcount-triggered TLB shootdowns still interrupt all the cores in the system. Even more, as we will see in Section VII, ∼80% of the interrupted slave cores are selected due to false positives, and in fact do not cache the affected mapping.

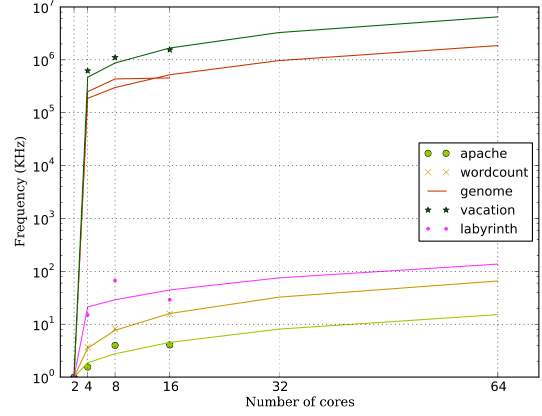
This result indicates that on the event of a TLB shootdown, current systems typically stop the execution of most, if not all, cores in the system.

### *E. Modeling The Impact of TLB Shootdowns On System Performance*

The characterization of the different parameters affecting the overhead incurred by TLB shootdowns allows us to develop a model to estimate their effect on overall system performance.

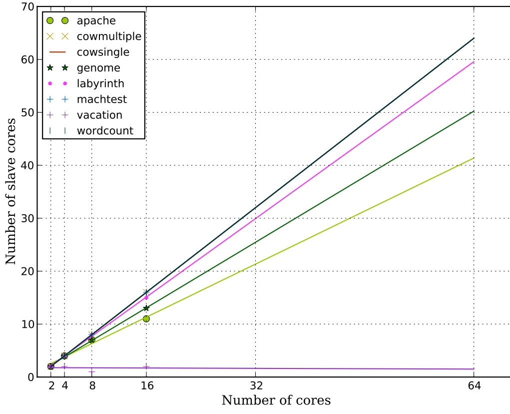



(a) Intel

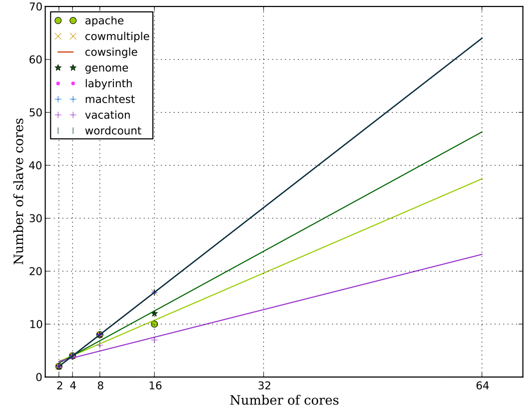



(b) AMD

Fig. 3. Shootdown frequency as a function of the number of system cores.




(a) Intel




(b) AMD

Fig. 4. The number of cores affected by a TLB shootdown as a function of the number of system cores.

Furthermore, as the different parameters can be characterized using a linear regression, the model can be used to both project the performance impact on larger multiprocessors, as well as to evaluate alternative TLB synchronization mechanisms.

Specifically, for a system with $p$ processors, the total number of system cycles spent executing a TLB shootdown is:

$$Cycles_{shootdown} = S(p) \times T_{slave}(p) + T_{initiator}(p)$$

Where $S(p)$ is the linear model for the number of slave cores participating in a TLB shootdown, $T_{slave}(p)$ is the local cycles executed by each slave, and $T_{initiator}(p)$ is the local cycles executed by the initiator core.

Furthermore, as the frequency $F(p)$ of TLB shootdown can also be represented as a linear model, the overall *fraction* of compute cycles lost on executing TLB shootdowns is:

$$\frac{Cycles_{shootdown} \times F(p)}{Processor\ frequency}$$

Figure 5 illustrates the percentage of machine cycles lost on TLB shootdowns, based on the performance impact model, as a function of the number of cores in the system. The figure depicts the average curve for Apache and Wordcount, based on the parameters for the Intel platform.

This demonstrates the importance of the TLB shootdown problem for future parallel machines. The percentage of machine cycles spent on TLB shootdowns increases linearly from ∼4% for the 16-core test machine, and up to 10% and 24% for future machines consisting of 64 and 128 processors, respectively.

Therefore, Figure 5 motivates the need for an efficient inter-TLB synchronization mechanism that will reduce the overheads associated with invalidating TLB entries, as well as maintain the accurate set of cores that cache a tainted page mapping.

## VI. DiDi - a TLB Directory

This section describes the design of the proposed shared and inclusive last-level TLB, as well as how this is extended to include a non-intrusive and highly-efficient mechanism for remote TLB invalidation.

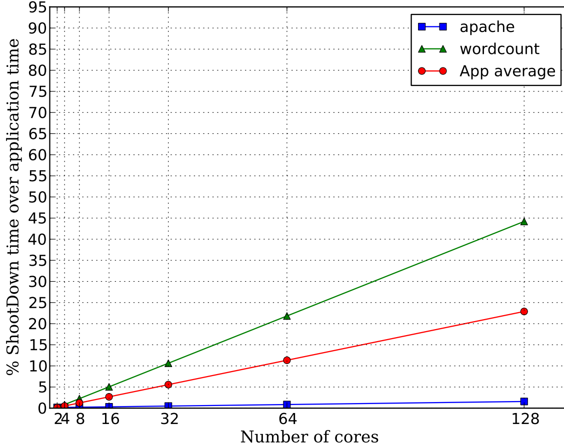



Fig. 5. The percentage of compute cycles lost on executing TLB shootdowns, as a function of number of cores in the system (based on the Intel model)

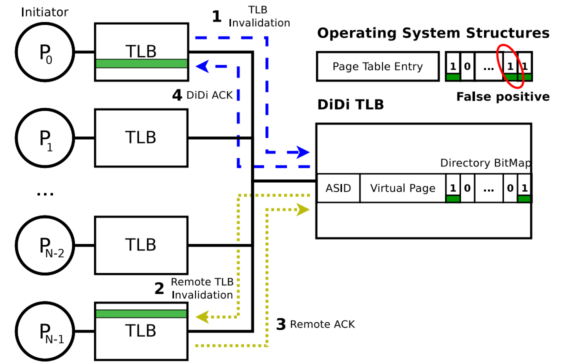



Fig. 6. DiDi architecture components (per-core private first-level TLB plus DiDi directory of system-wide TLB state), and non-intrusive remote invalidation mechanism.

## A. Shared Second-Level TLB Directory

As described in Section I, TLB shootdowns introduce a high performance penalty. As a synchronization process, the first step in a TLB shootdown and, also, the first problem for the performance of an application, is to establish which TLBs hold the page translation that is being invalidated. To this purpose, our design includes a shared second-level TLB. This second-level TLB is an associative cache that acts as a *Dictionary Directory* (*DiDi*). Its goal is to track the location of every address translation stored on the first-level TLBs of the whole system. Basically, all insertions and deletions on the first-level TLBs are intercepted by DiDi, such that it can always have up-to-date information on the system's TLBs state. The DiDi cache is located between the TLBs in the cores and the off-chip main memory. It acts as an intermediate proxy with no latency or delay on the message interception. All TLB updates are captured within this structure without affecting the performance.

Figure 6 depicts the proposed DiDi TLB architecture. As it can be observed, the OS page table is not always synchronized with the contents of the TLBs in the cores. The oval in the figure shows the page table containing information of a false positive on a page-table entry, as processor N-2 does not actually contain the page-table entry in its TLB.

On a TLB shootdown, a core TLB sends the invalidation to DiDi. Therefore, only cores containing the entry are notified, without the possibility of false positives. Core TLB invalidations to the second-level TLB are performed independently of the execution of the cores, as described in Section VI-B.

Whenever the first-level TLB performs an entry insertion or eviction, a message is sent to DiDi to maintain its property of being inclusive and always up-to-date. As the frequency of first-level TLB insertions and deletions is expected to be low, the fact that DiDi is centralized should not add noticeable contention to the interconnection network. TLB updates happen on a ratio of 1% or less. In a larger architectures, DiDi could be distributed, and it would only require some extra synchronization between several DiDi caches.

Note that the same address translation might be held by more than one TLB. Thus, each DiDi entry contains a *Directory Bitmap*, which identifies which cores are actually holding that specific translation on their private TLBs. As shown on Figure 6, this eliminates all cases of false-positives during a TLB shootdown.

The inclusiveness of the shared directory implies that directory evictions, following a miss on one of the cores' TLBs, may trigger an eviction in another core's TLB. Nevertheless, an evaluation of the inclusive property using the set of reference application, as well as the NAS [3] and PARSEC [9] parallel benchmark suites, has shown negligible effect on performance, and is therefore not shown.

In our evaluation, DiDi is implemented as a 2-way set associative cache of 4096 entries, which requires a total size around 4KB. Based on the CACTI [19] model, we estimate the proposed hardware structure would require a die area of 0.145 $mm^2$ and a power consumption of 13.5mW.

*1) Virtual Address Homonyms:* As virtual address spaces are independent address namespaces, there exist *homonyms* between them. In the case of the Intel and AMD architectures, this is solved by performing a complete TLB flush [2] whenever the OS changes the current address space. Instead, some other architectures like MIPS tag the TLB entries with some extra bits, which are usually known as the *Virtual Process Identifier* (VPID) or *Address-Space Identifier* (ASID). Unfortunately, as the TLBs have very tight delay constraints, the ASID stored in the TLB must have very few bits. This incurs on very frequent ASID reuse and, thus, the need to flush all the entries that were associated to the ASID being reused prior to using it for a different address space.

As DiDi contains entries from all cores, which might be running different address-spaces, we also need to tag its entries with an ASID. Fortunately, the optimal number of bits needed for the ASID in DiDi are bound by the number of hardware execution contexts in the system (the number of cores in a non-SMT architecture).

[2] In order to boost the OS performance, there is an extra page table entry bit which identifies pages that are never flushed during a context switch. This is typically used by the OS to map its own pages, which are common to all address spaces.

*2) Other Considerations:* Second-level TLBs have already been integrated into existing processors such as Intel Nehalem [4]. The common goal, as for second-level caches, is to provide additional cache capacity beyond what is provided by first-level caches.

Although DiDi is originally targeted to improve TLB coherency management, it could also be used as a second-level TLB to improve the system's address translation capacity, with the benefit of letting a TLB act as a "prefetcher" of page table walks for the other TLBs [7]. In such a case, a physical address field and page permission bits should be added to the page in the DiDi line.

In the case of an SMT system, the TLBs can be shared by the hardware contexts in a core, in which case entries are tagged. Thus, there is also a need to map the DiDi ASID to the execution context in each core.

Finally, in the case of *large pages*, these must be taken into account when an invalidation takes place. In such a case, the invalidation or eviction of a page must trigger the "reset" of all DiDi bits corresponding to all the entries covered by the large page in the *Directory Bitmap*.

### *B. Non-Intrusive Remote TLB Invalidation*

As seen in Figure 2, much of the performance overhead incurred by a TLB shootdown is related to interrupt processing. In order to eliminate this cost, we add a per-core *Pending TLB Invalidation* (PTLBI) buffer that DiDi is able to control in order to perform remote TLB invalidations. Note that this is the very same mechanism used to perform the remote TLB invalidations triggered by capacity constraints in DiDi, with the only addition that the OS is also able to issue an instruction to explicitly invalidate a TLB entry on all the cores.

In Figure 6 we can observe the steps to perform a system-wide TLB entry invalidation, which is conceptually a copy-cat hardware implementation of today's TLB shootdown software transaction:

1) An initiator core requests the invalidation of a TLB entry by sending a message to DiDi, which contains the virtual address and the currently executing ASID on the initiator core.
2) Using the received virtual address and ASID, DiDi looks up the target entry and retrieves the *Directory Bitmap* for the selected page. Using that bitmap, a TLB entry invalidation operation is sent to the affected cores (slaves) using a multicast message that contains the target virtual address.
3) When a slave finishes the invalidation of the selected TLB line, replies to DiDi with an acknowledgement message.
4) Once all requested cores have replied to DiDi, it clears the selected page from the directory and delivers a last acknowledgement message to the initiator core (the invalidation of the initiator core can be performed either at the beginning or the end of the transaction).

Figure 7 shows a high-level organization of the per-core PTLBI and the interactions between the PTLBI buffer and the core's components.

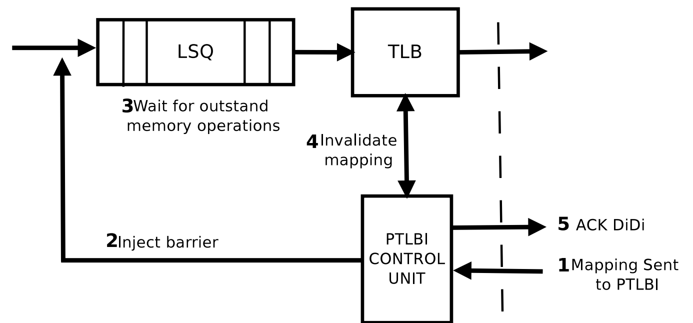



Fig. 7. Interactions between the PTLBI buffer and the core architectural components.

Upon receiving an invalidation request from DiDi, the PTLBI buffer injects a memory barrier into the core's *Load/Store Queue* (LSQ) in order to allow any outstanding memory operations to complete. Once the barrier completes, the PTLBI instructs the invalidation of the TLB entry identified by DiDi and sends an acknowledgement message back to DiDi. The purpose of the memory barrier injection is to ensure that no new memory instructions, which could conflict with the page being invalidated, will be executed. Thus, we estimate the cost of this operation to have an upper bound latency equal to a round-trip access to off-chip memory.

Figure 8 depicts the new time-line of a TLB shootdown in more detail. Using DiDi, the OS sends the invalidation request to the directory, which forwards the request to the PTLBI buffers on the cores that store the affected mapping (if any). It is important to note here that DiDi does not interrupt the execution of the remote cores, avoiding the costly pipeline flush and context saving and restoring that the OS would perform when receiving an IPI. The overhead of this operation will not take more than a few hundred cycles, compared to the typical interrupt overhead that can be as much as ten times higher [13].

## VII. Evaluation

### *A. Overhead of DiDi TLB Shootdown*

As we have described in the previous sections, our architectural support mitigates the impact of the TLB shootdown in two dimensions: first, it minimizes the number of affected cores by maintaining the information in the DiDi directory; and second, it minimizes the overhead in the slave cores with the PTLBI non-intrusive mechanism.

In Figure 9 we show a comparison of the overhead cycles caused by TLB shootdown in the baseline case, and using our architecture support (DiDi + PTLBI). The frequency of shootdown is the same in both cases, as measured in Apache and Wordcount in Section V, although it is possible that DiDi would filter out the false-positives from the broadcast. The overhead of a TLB invalidate using PTLBI is one order of magnitude smaller.

Our results show that the percentage overhead of TLB shootdown still increases with the number of processors, however, it stays within perfectly reasonable limits even at high processor counts, enabling scalability that was not possible under the baseline conditions.

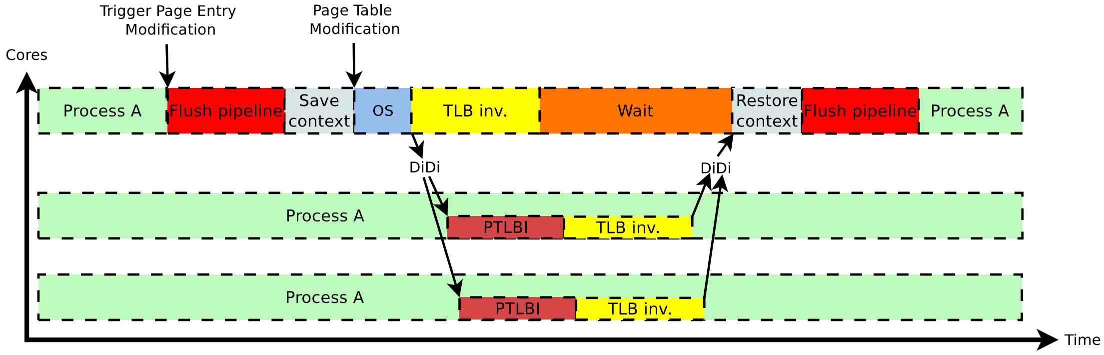



Fig. 8. Timeline of a DiDi TLB shootdown.

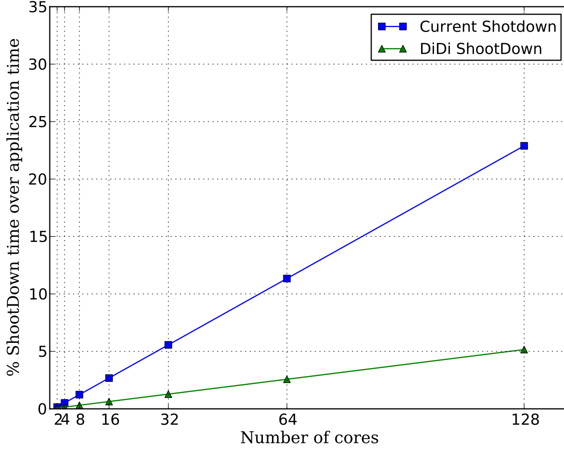



Fig. 9. Projection of the overhead of TLB shootdown over application time using current TLB Shootdown cost vs DiDi TLB Shootdown.

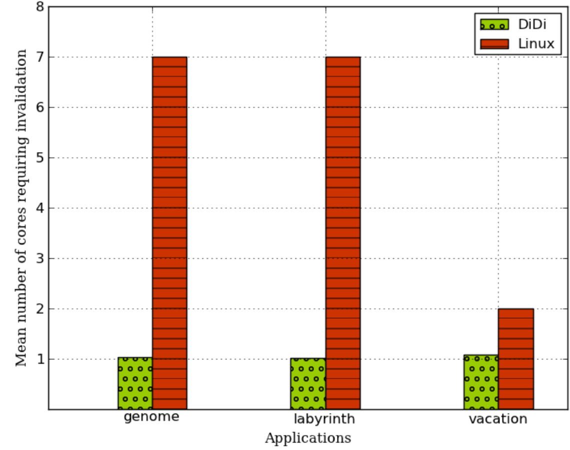



Fig. 10. Number of slave cores containing the page affected on the TLB Shootdown. Comparison on a 8-core simulation of DiDi vs real execution of the benchmarks on a real Linux machine with a 8 cores configuration.

### *B. False positives detection with DiDi*

The primary goal of DiDi's shared directory is to provide accurate information about which TLBs contain a stale mapping, and reduce the number of cores affected during a TLB shootdown by eliminating false-positives.

Figure 10 depicts the mean number of cores affected by a TLB coherence transaction, both for DiDi and a standard Linux TLB shootdown, for the set of the transactional memory benchmarks running with 8 threads, under the execution scenario explained in Section IV.

The figure shows that while typically only a single TLB caches an affected mapping, as reported by DiDi, the Linux kernel interrupts 2-7 cores due to false-positives, resulting from the need to approximate the affected set of core. Together with Figure 4, which shows that the Linux kernel typically interrupts most of the machine cores during a TLB Shootdown, these results highlight the detrimental effect of inaccurate approximation of the affected set of cores and the importance of DiDi's shared TLB directory.

### *C. Performance evaluation*

Finally, we evaluate the performance results of DiDi showing the evaluated benchmarks using the proposed directory and the less intrusive remote TLB invalidation mechanism.

Figure 11 shows the speedup obtained for the performance cost of the TLB Shootdown using DiDi vs the conventional Linux TLB Shootdown. The X-axis shows the number of cores for each configuration. Configurations for 8 and 16 cores are experimental results, while 32 and 64 are projections based on the previous configurations results. The Y-axis shows the percentage of speedup over the results obtained in the Intel machine. We compare to Intel instead of the AMD machine since the TLB Shootdown cost in this architecture is shown to be less. As we can observe in the Figure, all benchmarks obtain some speedup and the speedup increases as the number of cores in the configuration increases. Furthermore, *Genome* and *Vacation* obtain a speedup over 2.5 for the larger configurations. As it can be observed from Figures 3, 4, this is because both benchmarks have the greatest number of slave cores and the highest TLB Shootdown frequency.

## VIII. Conclusions

In this paper, we have characterized the performance impact of TLB shootdowns, and have shown they pose a serious

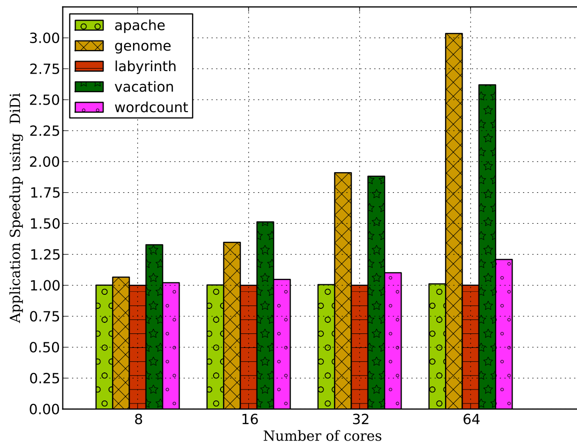



Fig. 11. Speedup of DiDi on the evaluated benchmarks. DiDi is evaluated on a 8,16 core CMP. For larger configurations, 32 and 64 core, we show a projection of the speedup

impediment to the scalability of future multiprocessors. Our analysis shows that the TLB shootdown problem stems both from the use of costly IPIs for TLB synchronization, as well as the inability of the OS to accurately identify the subset of cores that caches a tainted mapping.

To mitigate the harmful performance impact of TLB shootdowns, we have proposed DiDi, a TLB coherence mechanism based on a shared directory and a non-intrusive mechanism for remote TLB invalidation that does not interrupt the execution of the invalidated core. The proposed design therefore replaces the complex OS implementation of TLB coherence transactions with an accurate, lightweight hardware mechanism.

Finally, we have demonstrated that DiDi can mitigate the TLB shootdown problem and decrease the fraction of machine cycles wasted on executing TLB shootdowns by an order of magnitude.

## Acknowledgments

This research is supported by the Consolider program (contract No. TIN2007-60625) from the Ministry of Science and Innovation of Spain, the TERAFLUX project (ICT-FP7-248647), and the European Network of Excellence HIPEAC-2 (ICT-FP7-249013). This work was also supported by the cooperation agreement between the Barcelona Supercomputing Center National Supercomputer Facility and Microsoft Research. Y. Etsion is supported by a Juan de la Cierva Fellowship from Ministry of Science and Innovation of Spain. Special thanks to the members of the Heterogeneous Architectures group at BSC and the anonymous reviewers for their comments and suggestions.

## References

[1] M. Abadi, T. Harris, and M. Mehrara. Transactional Memory With Strong Atomicity Using Off-The-Shelf Memory Protection Hardware. *SIGPLAN Not.*, 44:185–196, February 2009.

[2] Apache Software Foundation. The Apache Test Project. http://httpd.apache.org/test/, 2011.

[3] D. H. Bailey, L. Dagum, E. Barszcz, and H. D. Simon. NAS Parallel Benchmark Results. In *Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, Supercomputing, pages 386–393, 1992.

[4] K. Barker, K. Davis, A. Hoisie, D. J. Kerbyson, M. Lang, S. Pakin, and J. C. Sancho. A Performance Evaluation of the Nehalem Quad-core Processor for Scientific Computing. *Parallel Processing Letters*, December 2008.

[5] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Symp. on Operating Systems Principles (SOSP)*, pages 29–44, 2009.

[6] A. Bhattacharjee and M. Martonosi. Characterizing the TLB Behavior of Emerging Parallel Workloads on Chip Multiprocessors. In *Intl. Conf. on Parallel Arch. and Compilation Techniques*, pages 29–40, 2009.

[7] A. Bhattacharjee and M. Martonosi. Inter-core Cooperative TLB For Chip MultiProcessors. In *Intl. Conf. on Arch. Support for Programming Languages & Operating Systems*, pages 359–370, 2010.

[8] B. A. Bhattacharjee, D. Lustig, and M. Martonosi. Shared Last-Level TLBs for Chip Multiprocessors. In *Symp. on High-Performance Computer Architecture (HPCA)*, 2011.

[9] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Intl. Conf. on Parallel Arch. and Compilation Techniques*, October 2008.

[10] D. L. Black, R. F. Rashid, D. B. Golub, and C. R. Hill. Translation Lookaside Buffer Consistency: A Software Approach. *Computer Architecture News*, 17(2):113–122, 1989.

[11] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[12] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *DISC*, pages 194–208, 2006.

[13] Y. Etsion, D. Tsafrir, and D. G. Feitelson. Effects of Clock Resolution on the Scheduling of Interactive and Soft Real-Time Processes. In *Intl. Conf. on Measurement & Modeling of Computer Systems (SIGMETRICS)*, pages 172–183, Jun 2003.

[14] P.-M. Fournier, M. Desnoyers, and M. R. Dagenais. Combined Tracing of the Kernel and Applications with LTTng. In *Linux Symposium*, Jul 2009.

[15] I. Gelado, J. H. Kelm, S. Ryoo, S. S. Lumetta, N. Navarro, and W. mei W. Hwu. CUBA: An Architecture For Efficient CPU/Co-Processor Data Communication. In *ACM Intl. Conf. on Supercomputing*, pages 299–308, 2008.

[16] H. P. Hofstee. Power Efficient Processor Architecture and The Cell Processor. In *Symp. on High-Performance Computer Architecture (HPCA)*, pages 258–262, 2005.

[17] B. L. Jacob and T. N. Mudge. A Look At Several Memory Management Units, TLB-refill Mechanisms, And Page Table Organizations. *Operating Systems Review*, 32(5):295–306, 1998.

[18] M. Martin, C. Blundell, and E. Lewis. Subtleties of Transactional Memory Atomicity Semantics. *IEEE Computer Architecture Letters*, 5:17–, July 2006.

[19] 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.

[20] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *Symp. on High-Performance Computer Architecture (HPCA)*, pages 13–24, 2007.

[21] R. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew. Machine-Independent Virtual Memory Management For Paged Uniprocessor And Multiprocessor Architectures. *Computer Architecture News*, 15(5):31–39, 1987.

[22] A. Rico, F. Cabarcas, A. Quesada, M. Pavlovic, A. J. Vega, C. Villavieja, Y. Etsion, and A. Ramirez. Scalable Simulation of Decoupled Accelerator Architectures. Technical Report UPC-DAC-RR-2010-14, Universitat Politècnica de Catalunya, Jun 2010.

[23] 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 *Symp. on High-Performance Computer Architecture (HPCA)*, 2010.

[24] B. Rosenburg. Low-Synchronization Translation Lookaside Buffer Consistency In Large-Scale Shared-Memory Multiprocessors. In *Symp. on Operating Systems Principles (SOSP)*, pages 137–146, 1989.

[25] S. Srikantaiah and M. Kandemir. Synergistic TLBs for High Performance Address Translation in Chip Multiprocessors. In *Intl. Symp. on Microarchitecture*, 2010.

[26] P. J. Teller, R. Kenner, and M. Snir. TLB Consistency On Highly-Parallel Shared-Memory Multiprocessors. In *Annual Hawaii Intl. Conf. On System Sciences (Architecture Track)*, pages 184–193, 1988.