TLB Shootdown Mitigation for Low-Power Many-Core Servers with L1 Virtual Caches

Binh Pham, Derek Hower,
Abhishek Bhattacharjee, and Trey Cain

Abstract—Power efficiency has become one of the most important design constraints for high-performance systems. In this paper, we revisit the design of low-power virtually-addressed caches. While virtually-addressed caches enable significant power savings by obviating the need for Translation Lookaside Buffer (TLB) lookups, they suffer from several challenging design issues that curtail their widespread commercial adoption. We focus on one of these challenges–cache flushes due to virtual page remappings. We use detailed studies on an ARM many-core server to show that this problem degrades performance by up to 25 percent for a mix of multi-programmed and multi-threaded workloads. Interestingly, we observe that many of these flushes are spurious, and caused by an indiscriminate invalidation broadcast on ARM architecture. In response, we propose a low-overhead and readily implementable hardware mechanism using bloom filters to reduce spurious invalidations and mitigate their ill effects.

Index Terms—Virtual Cache, virtual memory, TLB, multicores, multiprogramming, multithreading

1 Introduction

Efficient hardware support for virtual memory has been an important topic in the computer systems community because of its impact on performance and power. That hardware support is anchored by a variety of virtually-addressed tructures throughout the system, including Translation Lookaside Buffers (TLBs) and virtually-addressed caches. While much prior work has focused on TLB reach and hit rates [2], [13], [14], [15], this paper sheds light on the increasing importance of another source of overhead-that of TLB coherence or *shootdown* operations.

Virtually-indexed virtually-tagged or VIVT caches use the virtual address for both cache indexing and tag matching, obviating the need for prior TLB lookup. This has several benefits. For example, it eliminates TLB lookup power consumption for every instruction or data access. For the same reason, it reduces cache access latency. One might initially expect virtually-indexed physically-tagged (VIPT) caches to eliminate some of these problems too. While this can be the case (e.g., VIPT caches reduce access latency by overlapping TLB and part of the cache access), VIPT caches introduce problems of their own. Specifically, since VIPT caches overlap TLB lookup with cache indexing, they require the index bits to fit within the page offset. This limits the number of sets that a cache can support and means that caches can be grown only through higher associativity, consuming significantly more power. Consequently, there is a resurgence of interest in studying alternate VIVT caching in low-power server designs [6].

VIVT caches face some key challenges. First, they require special support to correctly manage address synonyms, the situation when multiple virtual addresses map to the same physical address. Synonyms are often used to support, for example, data sharing among multiple processes. VIVT caches use multiple cache lines to store synonyms, creating coherence issues. Second, VIVT caches require special support to correctly manage address homonyms, which occur when a virtual address maps to different physical addresses. This may occur when multiple processes simultaneously use the same virtual address to map different physical addresses, in disjoint address spaces. Synonyms and homonyms have seen significant research in the last few decades [3], [7], [9], [12], [21], with varying degrees of success. Our focus, however, is on a third, unaddressed, problem–page remappings. This is the situation when privileged software (e.g., an OS or hypervisor) modifies the content of a virtual-to-physical page mapping in a page table entry (PTE). PTE changes can occur for many reasons–permission changes, remapping a virtual page to a physical page, etc. The challenge with PTE modifications is that they require all stale VIVT cache lines to be made coherent. Maintaining VIVT cache coherence with page tables is a vexing problem.

PTE modifications are becoming increasingly important. Historically, they have been used for optimizations like copy-on-writes, memory defragmentation to create superpages, and page migration between NUMA nodes [11]. As systems embrace heterogeneous memories, combining high-bandwidth DRAM technologies with lower-cost, higher-capacity DRAM devices [1], operations like page migration are likely to become even more frequent. In any of these cases, a PTE modification means that VIVT cache lines from the virtual page of the PTE are now stale and must be invalidated. The natural mechanism to accomplish this is to use the standard TLB shootdown operation to also perform cache invalidations. This presents a problem, mainly due to the fact that traditional shootdowns operate at the granularity of a page of memory. Because cache lines are smaller than pages, a single page modification and its shootdown may require the invalidation of many VIVT cache lines. Identifying the exact set of lines affected by the page invalidation requires a scan of all cache lines in that page. Such an operation is not practical for several reasons:

- A cache scan takes multiple cycles, proportional to the number of lines in the largest page size. This can potentially require millions of cycles as the largest page size in architecture like ARM aarch64 is 16GB.
- If the time taken to scan the cache becomes long, a large number of incoming scan requests are queued, particularly in modern systems with multiple invalidations.
- The cache scan interferes with demand requests coming from the core.
- The core initiating the shootdown cannot progress until the scan operation completes, increasing the TLB shootdown latency and delaying the execution of subsequent dependent instructions.
- The high power consumed by the cache scan jeopardizes the low-power benefits of VIVT.

For these reasons, TLB shootdowns flush VIVT caches entirely, a quicker operation than scans. We characterize the impact of these flushes on the performance of a large-scale ARM systems, which represent an emerging class of server SoCs forthcoming from companies such as Applied Micro, Broadcom, Cavium Semiconductors, and Qualcomm. Our focus is ARM's TLB shootdown mechanism, which is based on several variants of *tlbi* instructions. These instructions include two operands encoded in a single 64 bit register operand: a 48 bit virtual address, and a 16 bit address space identifier (ASID). Upon execution, the *tlbi* instruction invalidate any translations (system-wide) with the following properties:

- Virtual pages matching the page number specified by the *tlbi*.
- Translations either marked as *global* and accessible by any ASID, or marked with the ASID specified by the *tlbi* instruction.

- *B. Pham and A. Bhattacharjee are with the Department of Computer Science, Rutgers University, Piscataway, NJ 08854. E-mail: {binhpham, abhib}@rutgers.edu.*
- *D.Hower is with Qualcomm Technologies, Inc. E-mail: dhower@qti.qualcomm.com.*
- *Trey Cain is with Qualcomm Datacenter Technologies, Inc.
E-mail: tcain@qti.qualcomm.com.*

*Manuscript received 23 Nov. 2016; revised 28 Feb. 2017; accepted 17 Mar. 2017. Date of publication 4 June 2017; date of current version 19 Mar. 2018.
(Corresponding author: Binh Pham.)
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/LCA.2017.2712140*

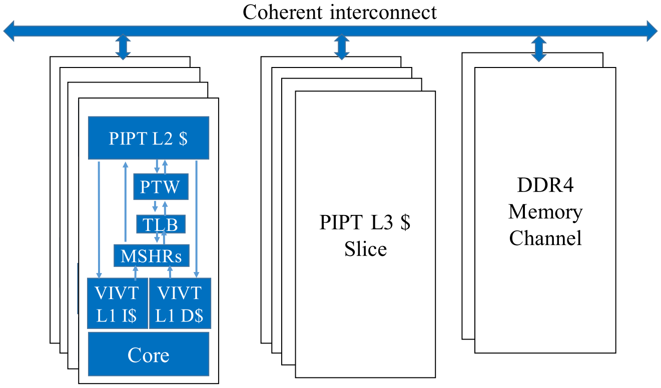

Fig. 1. System configuration: VIVT L1I and L1D caches backed by a PIPT L2 and distributed L3. A TLB access is not on the critical path for L1 cache access, but is looked up on L1 cache miss.

Furthermore, *tlbi* incorporates ”broadcast” semantics-when any processor executes the *tlbi*, its effect must be observed in any translation cache system-wide. This whole-system operation invalidates the TLBs and VIVT caches of all CPUs, though it does so without the expensive inter-processor interrupts of x86 systems. Unfortunately, the indiscriminate nature of the invalidations cannot selectively target remote cores. As a result, cores in a large-scale system observe TLB invalidations more frequently, exacerbating shootdown overheads.

2 Problem and Solution

2.1 Performance Impact of TLB Shootdowns

We quantify the TLB shootdown overheads on a many-core ARM system using a simulator based on the open source QEMU virtual machine. We use a map-reduce multithreaded implementation of word counting (`WordCount` [16]) and the `Graph500` benchmark [10]. We use these workloads as the TLB shootdown initiators and run them concurrently with integer benchmarks in SPEC CPU 2006 [8] as the shootdown victims. We use QEMU as a dynamic memory request generator that feeds a timing approximate performance model, which includes detailed multi-level caches, DDR controllers, and interconnect.The QEMU component runs slightly ahead of the timing model, allowing us to queue enough work to simulate out-of-order execution. Our basic out-of-order CPU model respects true dependencies but lacks important components such as branch predictor. For that reason, we focus primarily on memory-bound workloads whose performance is dominated by the performance of the memory system and interconnect. We simulate a system with 48 ARMv8 cores running unmodified Linux 3.15 kernel as shown in Fig. 1. Each core employs L1 VIVT caches backed by a unified TLB and L2. L1 misses look up the TLB, and then all lower levels of the memory hierarchy are looked up using physical addresses. TLB misses are handled using a hardware page table walker (PTW) that traverses the page table and fills the per-core TLBs. L2 caches are backed by a coherent interconnect connecting a collection of distributed L3 cache instances and DDR4 memory controller.

Fig. 2a and Fig. 2b show the frequency of TLB shootdowns during the execution of `WordCount` and `Graph500` respectively. Each point on the x-axis represents an interval of one million cycles. The y-axis plots the number of TLB shootdowns. As shown, these benchmarks frequently experience shootdowns, often in a bursty manner. This is because there are multiple threads in the two programs, parsing a common memory-mapped file and writing to the mapped pages subsequently. This triggers the kernel’s copy-on-write policy if the file is mapped with the corresponding flags set.

As explained in Section 1, each of these TLB shootdowns is broadcast to every all CPUs and flushes all L1 VIVT caches. Because of the indiscriminate nature of those broadcast invalidations, even the cores running “victim” SPEC benchmarks flush their L1 caches. We are interested in measuring the performance impact of those spurious shootdowns on our simulated system. Our baseline is an “ideal” case where we do not flush L1 instruction and data cache state.

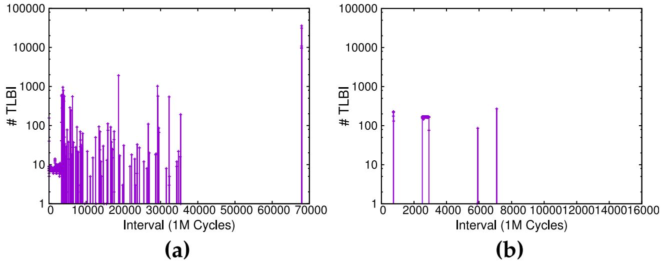

Fig. 2. Variation of TLBI frequency in (a) `WordCount` and (b) `Graph500`.

We first compare the ideal CPI of our “victim” benchmarks with the regular case where we do flush the L1 caches upon TLB shootdown. In this experiment, we allocate forty CPUs to run either `WordCount` or `Graph500` (our shootdown initiators), and the other eight CPUs to run copies of a SPEC application, which has very low or zero TLB shootdowns. We plot the CPI increase with respect to the “ideal” case in Fig. 3. As shown, the performance degradation measured of all victim applications remains consistently high, up to 25 percent when we pair `WordCount` with `sjeng`. On average, running SPEC applications concurrently with `WordCount` degrades performance by 12 percent while pairing with `Graph500` is 6 percent.

Due to space constraints, we omit the performance degradation results of the shootdown “initiators”. However, we find that both `WordCount` and `Graph500` suffer 1-2.5 percent CPI increase when running on a single CPU system, and this increases up to 5 percent for a 48 CPU system. We expect higher core counts to further exacerbate this problem. Because the “victim” applications experience much higher overheads from cache flushes under TLB shootdowns, the rest of this paper focuses on describing a technique to reclaim this lost performance.

2.2 Filtering Spurious Invalidations using Approximate History of Virtual Page Accesses

A seemingly straightforward solution is to employ a Bloom filter [4] to approximately track a set of addresses, similar to its use in other contexts [18]. In this context, the Bloom filter tracks the set of pages whose translations have been cached since the last time that the VIVT cache was invalidated, and incoming *tlbi* operations are checked against the Bloom filter. In the absence of a match, the *tlbi* operation can be safely dropped, otherwise one must conservatively assume that the cache *may* contain a cached translation entry, and the cache is invalidated.

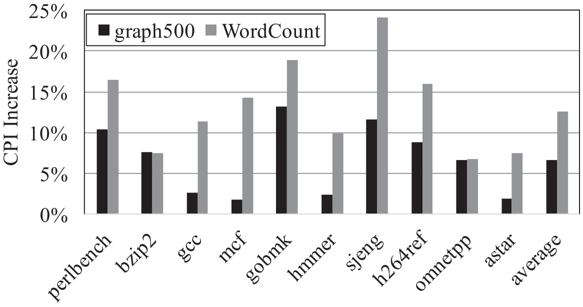

Fig. 3. Percent increase in the *victim app* CPI due to VIVT cache flushes that are induced by frequent TLB shootdown from initiator applications `graph500` and `WordCount`.

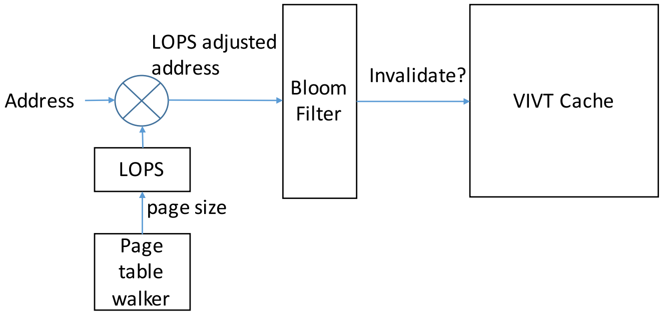

Fig. 4. Bloom filter with page size adjustment.

In order for the Bloom filter to track the pages correctly, the page size of the translation needs to be known. Bloom filter insertion is done at L1 VIVT cache fill time, and the page size information can be provided by the address translation unit after the L1 VIVT miss. Bloom filter check is performed anytime a *tlbi* operation is received on a core. Here a complication arises since the *tlbi* operations do not include the page size of the translation being invalidated. Without knowing the page size, extremely conservative assumptions need to be made about its potential size. For example, ARMv8 supports up to a 16 GB page size, so one might have to assume that a *tlbi* operation conflicts with any number of cache lines that fall in the referenced 16 GB region. Even though page size can be added to the *tlbi* operations in later ARM generations, here we propose an alternative solution that has very low overhead and can be implemented easily in hardware.

We couple the Bloom filter with a "largest observed page size" (LOPS) register that designates the largest page size for which a translation entry is cached. By tracking the largest observed page size, and conservatively assuming that any incoming TLB shootdown is of this page size, we can avoid having to assume that the incoming TLB shootdown must target the largest possible page size, but can instead assume a much smaller typical page. A block diagram of the components is shown in Fig. 4. Based on the contents of the LOPS register, we treat all addresses that update or check the Bloom filter as if they are based on that page size. Initially, the register is set to 4 KB. As lines are inserted into the cache, the register is used to determine the number of lower-order bits that are masked off of the address before it is used to update the Bloom filter. On the receipt of a *tlbi* operation, we similarly use the register to truncate the low order bits of the incoming *tlbi*'s address that is checked against the Bloom filter.

Since the addresses used to set the Bloom filter depend on the contents of the LOPS register, if a larger page size is observed therefore resulting in a register update, the Bloom filter must be invalidated, and along with it the VIVT cache. For example, as increasing page sizes are observed, some number of spurious cache invalidations may occur as the LOPS register homes in on the actual largest page size used in practice. We then retain the LOPS register, and only periodically reset it to 4 KB, allowing it to retain its value for long periods of time. The success of this scheme depends on the fact that it is rare for more that a small handful of page sizes to be used. In our study, we observe that the largest of which is still sufficiently small (e.g., 2 MB) to serve as a useful filter of the address space (as opposed to 16 GB).

For the purposes of this work, we assume that a Bloom filter is employed per cache, although it is also possible to share this unit across a group of virtually tagged structures. We use one of the $H3$ hash functions by Carter and Wegman [5] to compute the hashed value of a page number to index in our Bloom filters.

3 Performance Results

Fig. 5 shows the performance improvement of using bloom filters, comparing compared to the baseline. Our baseline, as before, assumes that a core's L1D and L1I caches are flushed on the receipt of a TLB invalidation command.

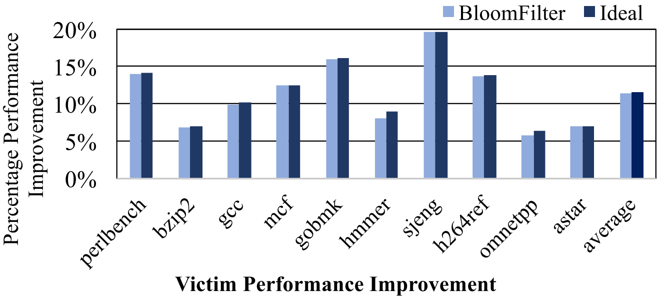

Fig. 5. Performance improvements of victim applications when using Bloom filters to reduce spurious cache flushes induced by TLB shootdowns.

Fig. 5 shows that bloom filters greatly improve performance by shielding the CPUs that run victim applications (and don't share the address space of the multi-threaded applications initiating shootdowns) from unnecessary shootdown activity. For all the benchmarks, we also include the results from a run in which the caches are never flushed, approximating the ideal case where the cache is oblivious to *tlbi* operations (i.e., as in a physically tagged cache). Bloom filters improve performance by an average of 12 percent, and achieve close to ideal performance in in every single case. Omitted for space reason, but we also show that this technique improves the performance of the application initiating the shootdowns. The primary source of this benefit is better instruction cache performance because our bloom filters help protect instruction caches from shootdowns that target pages in the data address space (rather than the instruction address space), and vice versa. We see a performance boost of 3 percent on average for shootdown initiators.

To help us understand why using bloom filters helps improve the performance of both victim and initiator applications, Fig. 6 compares the number of spurious cache flushes per kilo instructions observed between the baseline, where we always flush caches on receipt of a *tlbi* command, and the bloom filter design. As can be seen from Fig. 6, using bloom filters reduces the number of unnecessary d-cache flushes significantly across all workloads, and more than 80 percent on average. This reduction is even better for

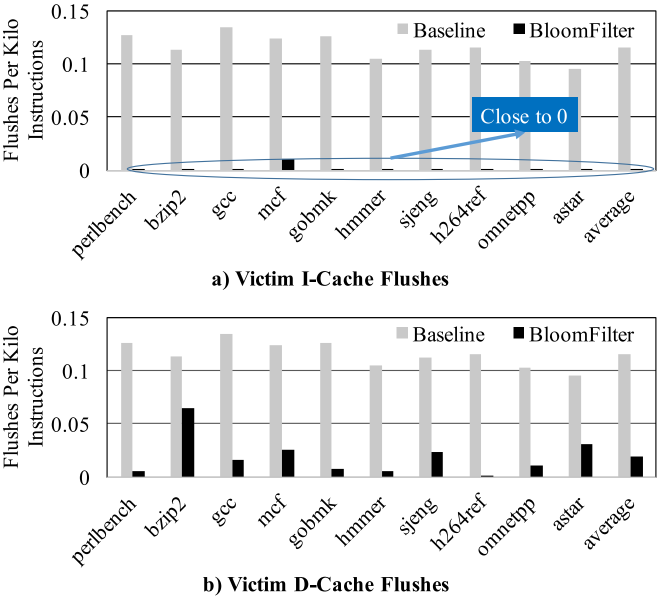

Fig. 6. Number of spurious cache flushes observed in *victim* applications in the baseline versus using bloom filters for a) instruction caches, b) data caches.

the instruction caches. The combination of this large cache flush reduction from both types of caches explains why our victim application speedup is close to ideal.

4 Related Work

The bulk of recent work to improve TLB shootdown performance has focused on IPI-based mechanisms typical of x86 [11], [17], [19]. While operating systems, e.g., Linux, already has mechanisms such as tracking thread migration to shield TLB shootdowns from cores that never execute a particular process, Villavieja et al. [19] showed that this results in a very high false positive rate, which happens when the target translation has been evicted from the core TLB but the OS has no knowledge about this. Together with the costly IPI handling, TLB shootdown presents a serious problem to multicore system scalability. Instead, they proposed a shared directory to precisely track location of all address translations present in the first-level TLBs of the whole-system in order to know exactly which core to forward TLB shootdowns to. However, their proposal does not protect icaches from shootdowns targeting dcaches and vice versa . They also rely on updating a centralized structure for every insertion or deletion from all first level TLBs, which potentially hampers the scalability of the system. Our design does not suffer from these limitations.

Yoon and Sohi [21] recently proposes a hardware mechanism to group all synonymous pages of the same physical page to a single leading virtual page. Their design involves using the active synonym detection table (ASDT) to check whether lines from a physical page are present in the virtual cache, and they discussed the possibility of using this structure to filter out TLB shootdown events. However, as TLB shootdowns target virtual pages, while ASDT is indexed by physical pages, this would require scanning ASDT for matching virtual pages. Our bloom filter design can be integrated with their design to speed up this operation, and we leave it for future work.

Several other studies look at engineering non-intrusive microcoded versions of the shootdown code on recipient cores [11], and mechanisms to replace IPIs entirely by leveraging existing cache coherence protocols to also perform TLB coherence [17], [20].

Overall, most of past work focuses on multi-threaded workloads, where threads running on different cores do indeed share the same address space and hence may genuinely require the TLB shootdowns. This however, ignores the more common multi-programmed case, where multi-threaded workloads may share the system with simultaneously running single- or multi-threaded workloads. TLB shootdowns (and their harmful cache effects when using virtually-tagged caches) are often spurious in this scenario, injuring “victim” applications needlessly.

5 Conclusion

In this work, we have explored the challenge of VIVT translation coherence in the the context of the emerging class of low-power/high-core count ARM-compatible servers. We have demonstrated the tlbi-shootdown scaling problem both for applications that include significant shootdown activity, as well as applications that suffer without any shootdown activity of their own. In response, we propose a filtering technique that augments traditional ISA-based shootdown operations to lower their performance overheads. We show that this can help reduce VIVT translation shootdown overheads to nearly nil. Given the simplicity of our implementation, we expect our technique to be readily deployed in the emerging class of low-power/high core-count servers.

References

[1] N. Agarwal, D. Nellans, M. O’Connor, S. Keckler, and T. Wenisch, “Unlocking bandwidth for GPUs in CC-NUMA systems,” in *Proc. Int. Symp. High Performance Comput. Archit.*, 2015, pp. 354–365.

[2] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, “Efficient virtual memory for big memory servers,” in *Proc. 40th Annu. Int. Symp. Comput. Archit.*, 2013, pp. 237–248.

[3] A. Basu, M. D. Hill, and M. M. Swift, “Reducing memory reference energy with opportunistic virtual caching,” in *Proc. 39th Annu. Int. Symp. Comput. Archit.*, 2012, pp. 297–308.

[4] B. H. Bloom, “Space/time trade-offs in hash coding with allowable errors,” *Commun. ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970.

[5] J. L. Carter and M. N. Wegman, “Universal classes of hash functions (extended abstract),” in *Proc. 9th Annu. ACM Symp. Theory Comput.*, 1977, pp. 106–112.

[6] Cavium, *ThunderX Family of Workload Optimized Processors*. 2015.

[7] M. Cekleov and M. Dubois, “Virtual-address caches, part 2: Multiprocessor issues” *IEEE Micro*, vol. 17, no. 6, pp. 69–74, Nov. 1997.

[8] J. L. Henning, “SPEC CPU2006 benchmark descriptions,” *SIGARCH Comput. Archit. News*, 2006.

[9] S. Kaxiras and A. Ros, “A new perspective for efficient virtual-cache coherence,” in *Proc. 40th Annu. Int. Symp. Comput. Archit.*, 2013, pp. 353–546.

[10] R. C. Murphy, K. B. Wheele, B. W. Barrett, and J. A. Ang, “Introducing the graph 500,” 2010.

[11] M. Oskin and G. H. Loh, “A software managed approach to die-stacked DRAM,” in *Proc. Int. Conf. Parallel Archit. Compilation*, 2015, pp. 188–200.

[12] C. H. Park, T. Heo, and J. Huh, “Efficient synonym filtering and scalable delayed translation for hybrid virtual caching,” in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit.*, 2016, pp. 90–102.

[13] B. Pham, A. Bhattacharjee, Y. Eckert, and G. Loh, “Increasing TLB reach by exploiting clustering in page translations,” in *Proc. IEEE 20th Int. Symp. High Performance Comput. Archit.*, 2014, pp. 558–567.

[14] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee, “CoLT: Coalesced large-reach TLBs,” in *Proc. 45th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2012, pp. 258–269.

[15] B. Pham, J. Vesely, G. H. Loh, and A. Bhattacharjee, “Large pages and lightweight memory management in virtualized environments: Can you have it both ways?” in *Proc. 48th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2015, pp. 1–12.

[16] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, “Evaluating MapReduce for multi-core and multiprocessor systems,” in *Proc. IEEE 13th Int. Symp. High Performance Comput. Archit.*, 2007, pp. 13–24.

[17] B. Romanescu, A. Lebeck, D. Sorin, and A. Bracy, “Unified instruction/translation/data (UNITD) coherence: One protocol to rule them all,” in *Proc. 16th Int. Symp. High-Performance Comput. Archit.*, 2010, pp. 1–12.

[18] D. Sanchez, L. Yen, M. D. Hill, and K. Sankaralingam, “Implementing signatures for transactional memory,” in *Proc. 40th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2007, pp. 123–133.

[19] C. Villavieja, et al., “Didi: Mitigating the performance impact of TLB shootdowns using a shared TLB directory,” in *Proc. Int. Conf. Parallel Archit. Compilation Techn.*, 2011, pp. 340–349.

[20] Z. Yan, J. Vesely, G. Cox, and A. Bhattacharjee, “Hardware translation coherence for virtualized systems,” in *Proc. 40th Annu. Int. Symp. Comput. Archit*, 2017.

[21] H. Yoon and G. S. Sohi, “Revisiting virtual l1 caches: A practical design using dynamic synonym remapping” in *Proc. IEEE Int. Symp. High Performance Comput. Archit.*, 2016, pp. 212–224.