Avoiding TLB Shootdowns through Self-invalidating TLB Entries*

Amro Awad*, Arkaprava Basu†, Sergey Blagodurov†, Yan Solihin‡ and Gabriel H. Loh†
*University of Central Florida (UCF), Email: amro.awad@ucf.edu
†Advanced Micro Devices, Inc., Email: {arkaprava.basu,sergey.blagodurov,gabriel.loh}@amd.com
‡North Carolina State University, Email: solihin@ncsu.edu

***Abstract*—Updates to a process's page table entry (PTE) renders any existing copies of that PTE in any of a system's TLBs stale. To prevent a process from making illegal memory accesses using stale TLB entries, the operating system (OS) performs a costly TLB shootdown operation. Rather than explicitly issuing shootdowns, we propose a coordinated TLB and page table management mechanism where an expiration time is associated with each TLB entry. An expired TLB entry is treated as invalid. For each PTE, the OS then tracks the latest expiration time of any TLB entry potentially caching that PTE. No shootdown is issued if the OS modifies a PTE when its corresponding latest expiration time has already passed.**

In this paper, we explain the hardware and OS support required to support *Self-invalidating TLB entries* (SITE). As an emerging use case that needs fast TLB shootdowns, we consider memory systems consisting of different types of memory (e.g., faster DRAM and slower non-volatile memory) where aggressive migrations are desirable to keep frequently-accessed pages in faster memory, but pages cannot migrate too often because each migration requires a PTE update and corresponding TLB shootdown. We demonstrate that such heterogeneous memory systems augmented with SITE can allow an average performance improvement of 45.5% over a similar system with traditional TLB shootdowns by avoiding more than 65% of the shootdowns.

***Keywords*-Heterogeneous Memory; Self-Invalidation; Virtual Memory; TLB; TLB Shootdown; HW/SW Co-design; Non-Volatile Memories;**

I. Introduction

The virtual memory system in modern computers enable a wide range of features, from virtualization of the machine's physical memory and protection, to a wide variety of optimizations such as copy-on-write, memory compression, garbage collection, late binding/allocation of physical memory, memory relocation, and many aspects of processor virtualization. Processors support high-performance virtual memory translations by caching recently accessed page table entries (PTEs) in the low-latency translation lookaside buffers (TLBs).

In the process of supporting the various virtual memory maintenance or optimization operations, however, modifications of the page table entries (e.g., changing an address mapping or page permissions) typically require a TLB shootdown operation. For example, software-transactional memory [14], memory management debugging [16], MapReduce-like frameworks [51], and concurrent garbage collectors [13] can trigger frequent alterations of PTEs and thus associated TLB shootdowns. Importantly, emerging heterogeneous memory systems critically depend upon efficient page migrations across memory backed by disparate technologies. Page migrations alter mappings between virtual addresses and the physical addresses backing them, typically triggering TLB shootdowns [37].

Unfortunately, a TLB shootdown is an expensive operation involving the operating system (OS) and hardware that removes all possible stale copies of the modified PTE from all TLBs that could be caching it. This is necessary to ensure any future accesses make use of the correct, most up-to-date mapping. Depending on the system architecture, number of cores, and number of application threads, the latency for a TLB shootdown can take up to 13.2 μs [37].

In this work, we propose a new hardware-software cooperative approach to manage TLBs that greatly reduces the number of costly TLB shootdown operations. Traditionally, entries are loaded into a TLB on TLB misses and remain valid until they are evicted by the TLB's replacement policy or are explicitly invalidated by a TLB shootdown. Rather than let these TLB entries potentially linger on indefinitely in the processor TLBs until explicitly shot down, we assign each TLB entry an *expiration time*. This acts like a "self-destruct" mechanism, where the processor automatically treats expired entries as invalid. At the same time, for each PTE, the OS keeps track of the *latest* expiration time assigned to any TLB entry potentially caching that PTE. If a PTE is modified any time after its corresponding latest expiration time has passed, then a shootdown is not needed. It is *guaranteed* that *no valid* copy of the PTE being modified is cached in any TLB, by design. This paper details the architectural and OS support required to build an effective virtual memory management system based on this idea of *self-invalidating* TLB entries (SITE).

We demonstrate the potential of this approach with a use case based on emerging heterogeneous memory systems [e.g., a mix of DRAM and non-volatile memory (NVM)]. In such systems, there is a tension between frequent page migrations from the slower NVM to the faster DRAM (to increase the fraction of memory requests serviced by the high-performance DRAM) and less frequent migrations (to

*Most of this work was done when Amro Awad was an intern at AMD Research. He is now an Assistant Professor at UCF.

decrease the performance impact of TLB shootdowns). SITE could present a solution that relaxes this tension to provide both high DRAM service rates and low shootdown overheads, resulting in a substantial improvement in performance over current systems.

In summary, we make the following contributions.

- We introduce the idea of *Self-invalidating* TLB entries (SITE) to eliminate many TLB shootdowns.
- We detail the hardware and OS modifications needed for implementing a SITE-based system.
- We perform detailed evaluation of SITE in reducing the number of TLB shootdowns for a heterogeneous memory system.

II. Background and Motivation

We describe the TLB shootdown process, how it is implemented, and why it does not scale. We discuss sources of TLB shootdowns with a particular focus on heterogeneous memory systems as they serve as our primary working example in this paper. We motivate our work by demonstrating the tension between page migrations and TLB shootdowns in such systems.

A. TLB Shootdown

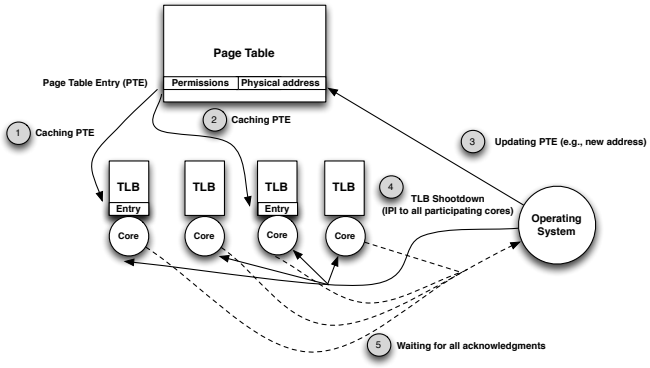


Figure 1. The process of TLB Shootdown.

Modifications to a page table entry (PTE) are typically accompanied by a TLB shootdown. If a processor core continues executing with a stale TLB entry, then erroneous execution may occur due to use of out-of-date address mappings or page permissions. The TLB shootdown process ensures that any cached copies of the modified PTE are discarded before continuing execution of the affected application.

Prevalent commercial processors like those based on x86-64 do not support automatic invalidation of stale TLB entries on a modification to a PTE. The responsibility of invalidating stale TLB entries falls on the OS, which does so via the TLB shootdown process.

As shown in Figure 1, a page table entry (PTE) can be cached in private TLBs of different CPU cores ①②. Later, when the OS updates the PTE ③, the OS issues or initiates an inter-processor interrupt (IPI) across all participating cores ④[1]. Each receiving core executes an interrupt handler routine that invalidates the entry for any cached copies of the PTE in the core's local private TLBs, and then sends an acknowledgment back to the initiating core. In x86-64, invalidating within a local TLB entry is done by either executing an *invlpg* instruction or by writing to the *cr3* register. The OS's TLB shootdown routine running at the initiator core waits for acknowledgment from each receiving core before concluding the overall PTE update process ⑤.

The overhead of interrupting cores and waiting for all of the acknowledgments is very high, and the overhead increases with the number of participating cores. There are multiple sources for the shootdown overheads, including multiple user/kernel mode transitions and the usage of legacy APIC hardware (on x86-based systems) [37], [51]. Rather than reducing the shootdown latency, this work presents a new mechanism to avoid many shootdown operations. We discuss other related mechanisms that could reduce shootdown latency in Section VII, although they are generally orthogonal to our approach.

B. Heterogeneous Memory Systems

Conventional memory systems are based on DRAM technology, which is running up against fundamental physical limitations as the DRAM cell sizes continue to shrink [36]. At the same time, a variety of emerging non-volatile memory (NVMs) technologies are gaining traction, including phase-change memory [30], [28], STT-MRAM [2], memristors [52], and 3DXPoint memory [15]. While the exact characteristics of the different technologies vary, a common challenge is that the performance of these NVMs is typically worse than DRAM. Thus, using NVM as a wholesale replacement for DRAM is unlikely. Instead, many researchers are advocating heterogeneous memory systems consisting of some DRAM for performance coupled with NVM for capacity [7], [38], [42].

There are many possible organizations of heterogeneous memory systems. In this paper, we consider a system with a mix of fast DRAM and slower NVM. Other possibilities include a combination of high-bandwidth in-package (3D-stacked) DRAM [1], [4] with either conventional DRAM or NVM outside of the package. While several different approaches have been proposed to manage such heterogeneous memory systems, in this work we consider a system where the OS is responsible for migrating pages between the different memories in a manner that is transparent to the

[1]In a multi-process scenario, when the OS knows that a process does not have any threads running on particular cores, the IPI does not need to be issued to those "non-participating" cores. There are other optimizations that allow filtering of some IPIs, but in general the IPI will still need to be sent to all cores involved in executing threads for the affected process.

application programmer. More details of the exact mechanisms are described later, and other alternative approaches to manage heterogeneous memory systems are discussed in Section VII.

C. Shootdowns vs. Page Migration

Using the heterogeneous memory system described above as a working example, we now examine the tension between aggressive memory management and the corresponding TLB shootdown overheads. We consider a system with a fast, first level of memory (e.g., DRAM), and a slower-but-larger second level of memory (e.g., NVM). The details of our experimental system can be found in Section V-A. Our baseline heterogeneous memory management policy simply keeps track of each page currently mapped to the slow memory. If the page is accessed more than a threshold number of times, then it is migrated to the fast memory. If necessary, a page from the fast memory may also need to be moved back to the slow memory to make room (selected with a clock replacement policy [48]).

For a low migration threshold, a page in the slow memory only needs to be referenced a few times before being migrated to the fast memory. This allows the system to be more responsive, aggressively moving pages to (hopefully) allow more future references to be serviced from the faster memory. A large migration threshold however, requires that a page exhibit high levels of reuse before promoting it to the fast memory. This reduces the likelihood of migrating a page with few future uses, but it also increases the number of requests that must be serviced out of the slower memory.

Figure 2 shows results for two representative applications as we vary the migration threshold. Let us first consider the Lulesh application on the left. The left y-axis shows the fast memory's miss rate (the fraction of memory requests *not* serviced by the fast memory; lower is better). Lulesh demonstrates reasonably good memory locality; when a page is accessed, it is likely to be accessed many times again in the near future. As a result, when we set the migration threshold to 1 (i.e., move a page from slow to fast memory on the very first access), we achieve a very low miss rate from the fast memory. As the threshold is increased, the miss rate steadily climbs because each memory access on our way to reaching the migration threshold results in a miss in the fast memory. However, simply setting a low migration threshold can still be disastrous for performance, as every migration is accompanied by costly TLB shootdowns. The right y-axis shows the number of TLB shootdowns incurred, and as the migration threshold is increased, the number of shootdowns drops.

The right side of Figure 2 shows another application, RSBench, that shows the same macro-scale trends, but the slopes and concavities of the curves differ. RSBench has poorer spatial locality than Lulesh, with many pages that are accessed only a few times. Even with a migration threshold of 1, almost half of the memory accesses miss in the fast memory. When increasing the migration threshold to 10, we observe a large drop in the TLB shootdown rate, indicating that there are a significant number of pages that are accessed fewer than ten times before being evicted out of the fast memory.

Takeaways: These results illustrate the tradeoff between maintaining low fast memory miss rate in a heterogeneous memory system against the number of costly TLB shootdowns. If the overall cost of TLB shootdowns could be reduced, then the heterogeneous memory management system could afford to be more aggressive in its migration decisions, leading to more memory accesses being serviced from the faster memory. In this work, we take the approach of reducing the *frequency* of shootdown operations through self-invalidating TLB entries (described in the next section).

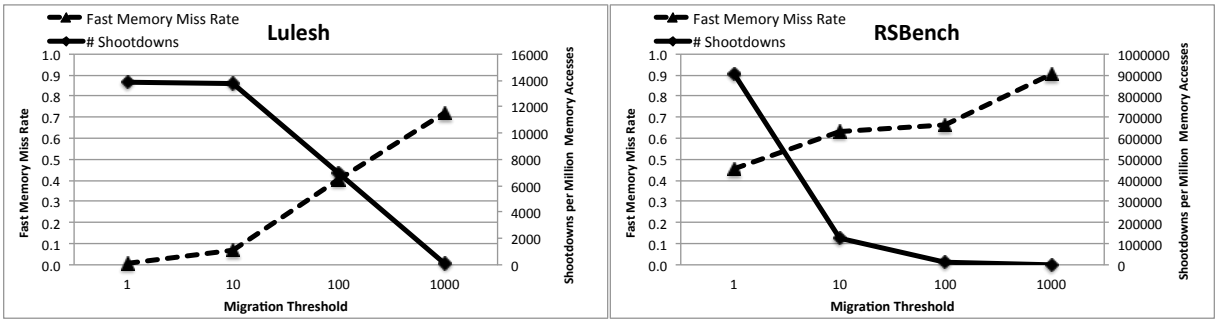
Furthermore, the results also illustrate that different applications react differently in terms of fast-memory miss rates and the corresponding TLB shootdown rates. This suggests that our proposed solution should have some dynamic component to it to adapt to variations between and even within workloads.

Other sources of TLB shootdowns: Beyond heterogeneous memory system, there are several other use cases that could benefit from a technique to avoid shootdowns. For example, software transactional memory dynamically modify page permissions to detect conflicts among concurrent transactions [14]. Concurrent garbage collectors modify page permissions and re-maps pages to reclaim memory without possibility of race conditions [13]. Memory management bugs could be detected by altering page permissions [16]. These all could benefit from reduced overall cost of TLB shootdowns.

III. Self-invalidating TLB Entries (SITE)

Traditionally, address translations (contents of PTEs) are loaded into the TLB by a hardware page table walker (e.g., in ARM and x86-based systems) on a TLB miss and remain valid until they are evicted by the TLB's replacement policy or explicitly invalidated by a TLB shootdown. Instead, we propose self-invalidating TLB Entries or SITE to cache PTEs in TLBs with an expiration time. SITE enforces an invariant that a translation entry in the TLB is valid only if its expiration time is in the future. Thus, an entry in the TLB with an expired "lease" *does not need to be explicitly invalidated*. A shootdown can then be avoided by taking advantage of this invariant if the latest expiration time of a given PTE is known to be in the past.

For a concrete exposition of SITE, consider the scenario shown in Figure 3. The left-hand side of the table shows an example sequence of events and their timestamps in a SITE-based system. The right-hand side depicts these events in a system with two CPU cores. Here, core C_0 accesses virtual address A ①. The corresponding TLB in that core



(a) Lulesh (b) RSBench

Figure 2. The impact of the migration threshold on fast memory miss rate and number of migrations.

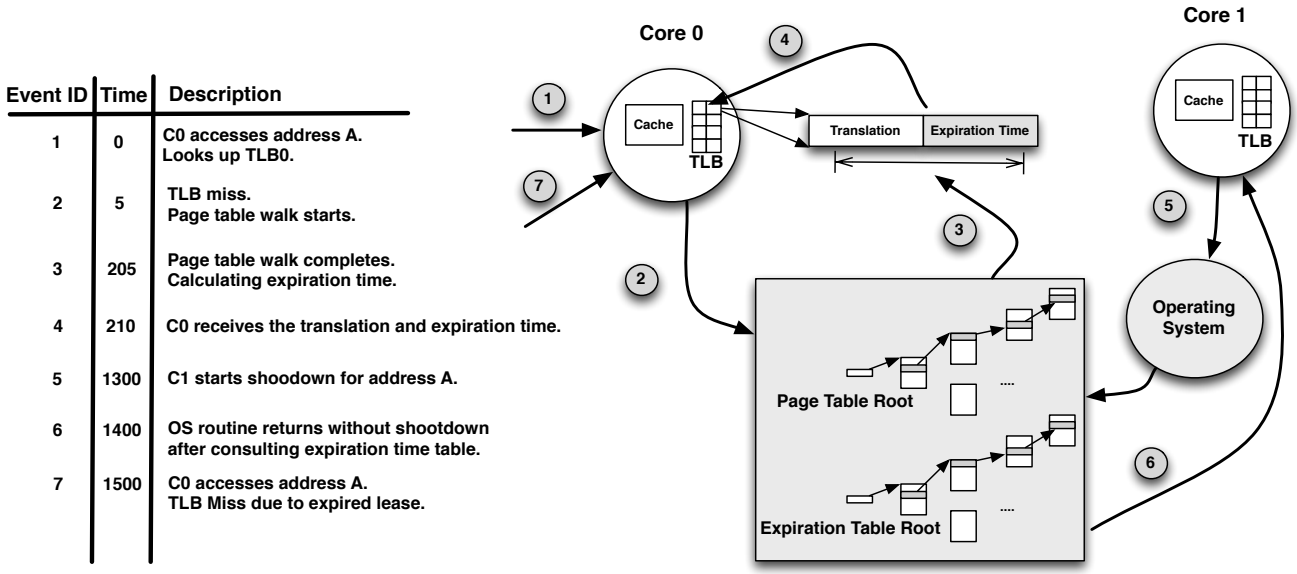


Figure 3. A scenario demonstrating the self-invalidating TLB system.

is searched for a translation for A. Assume that the lookup misses and a page table walk is initiated to find the in-memory PTE that holds the desired translation at time $T = 5$ (2). At time $T = 205$, the hardware page table walker locates the PTE (3). However, different from a traditional system, the walker in a SITE-based system retrieves the translation *and* assigns an expiration time for A. The expiration time is calculated to be 1205 by adding a chosen *lease length* (1,000 in this example) and the current time ($T = 205$). Furthermore, the walker records A's expiration time in a new in-memory data structure called the *Expiration Time Table (ETT)*. At time $T = 210$, Core C_0's TLB receives A's translation and installs it in its TLB along with the expiration time (4). C_0 then initiates its data cache access using the provided physical address. Design details such as lease-length selection are described in the next subsection.

Later at $T = 1300$, another core C_1 initiates the process of invalidating the translation for address A (e.g., due to a PTE modification) (5). This involves calling a specific routine in the OS. At $T = 1400$, this modified OS routine for SITE alters the PTE for address A in the page table and then consults the corresponding entry in ETT (6). The OS routine finds the expiration time for address A in the ETT is 1205, which is in the past. The OS then immediately returns without performing the typical TLB shootdown and thus avoids the corresponding cost. At $T = 1500$, core C_0 again attempts to access data with virtual address $A_{\bigcirc}$. However, on a TLB lookup this time, the translation is found to have already expired (expiration time 1205) and cannot be used. This ultimately results in a page table walk, similar to what was explained above (2).

In summary, we observe that a SITE-based system can avoid performing TLB shootdowns by utilizing the invariant that a PTE whose latest-known expiration time (according to the ETT) ensures that no valid copies of the PTE can exist in any TLBs. At the same time, using SITE can also

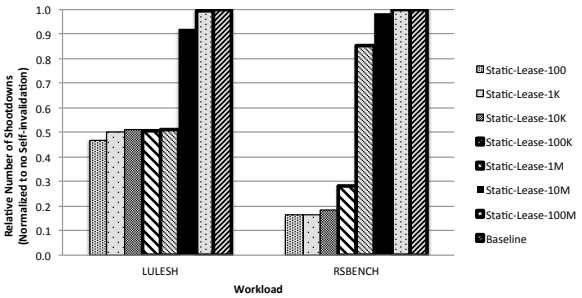


Figure 4. Number of TLB shootdowns normalized to the baseline (no self-invalidation) with varying lease length.

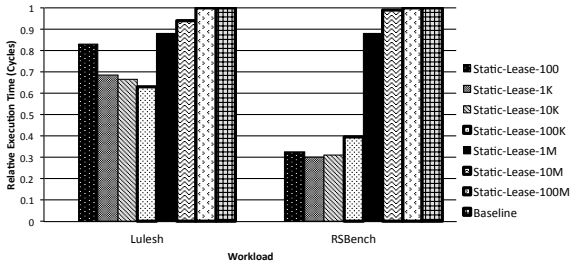


Figure 5. Impact of the lease length on execution time, normalized to the baseline (no self-invalidating TLB).

introduce extra page walks due to expired TLB entries that would have remained valid in a conventional system.

A. Design Considerations for SITE

While the general idea of self-invalidating TLB entries is intuitive and simple, there are several key design considerations to assemble an effective TLB-management solution. Below, we discuss key design dimensions and our rationale for each of our choices.

1) Unit of Lease and Expiration Time: A fundamental need for supporting SITE is a way to mark the passage of time. An obvious option is to use the physical clock tick as the unit for lease and expiration times, as has been previously proposed in the context of hardware cache coherence (e.g., globally synchronized timestamps) [45], [43]; commercial vendors also suggest the possibility of realizing such global timestamps at least within a single chip [19]. While an embodiment of SITE could use such global physical time, we find it is unnecessary. Unlike cache accesses and coherence traffic, TLB shootdowns occur much less frequently, and so a fine-grained timestamp based on clock ticks is not needed for SITE. On the other hand, SITE is more valuable in large systems, possibly with multiple chips/sockets, as TLB shootdown costs increase with system size [51], [40]. Keeping a physical clock synchronized across large systems can be challenging.

We instead use the main memory access (DRAM and NVM) count as the *logical time* unit for lease and expiration times in our proposed SITE implementation. There are at least two reasons behind this choice. First, the main memory access count changes at a much lower rate than clock tick as memory is accessed only after a miss in the last-level cache and is therefore easier to keep synchronized (more in Section VI-B). Second, there are only a handful of memory controllers in a system and therefore the hardware modifications are limited.

2) Choosing the Lease Length: Choosing an effective lease length is crucial. If the lease is too short, then TLB entries will expire quickly, incurring additional TLB misses. This will ultimately increase address translation cost. On the other hand if the lease length is too long, then the TLB entries will not self-invalidate quickly enough to avoid shootdowns. Figure 4 shows how the fraction of incurred TLB shootdowns (lower is better) changes with lease length for two representative benchmarks (the rest are in Section V). As expected, shorter leases reduce the number of shootdowns. On the other hand, however, the address translation cost grows with shorter lease lengths as additional TLB misses are incurred (shown in Figure 11 in Section V).

These two conflicting trends imply that the overall application execution time can be minimized only if the lease length is *neither too short nor too long*.

Dynamic Lease-length Assignment: Figure 5 shows the normalized execution time of two representative workloads with varying statically selected lease lengths (lower is better). Execution times are normalized to a baseline without self-invalidation. We observe that the application LULESH performs best when the lease length is 100K (static-100K, fourth bar from left), and either increasing or decreasing the lease length increases execution time. The same is true for RSBench, but the best-performing lease length is different (1k). Furthermore, even within a single application, different memory regions (e.g., stack vs. heap) experience different access patterns and are thus likely to prefer different lease lengths. These suggest that the desired lease length should be *dynamically* discovered at runtime.

To appreciate what lease length (range) may be desirable for a PTE, Figure 6 shows the timeline of an example sequence of shootdowns and page table walks for a virtual address *A*. In the figure, the page walks PageWalk_{e0} and PageWalk_{e1} occur due to expired entries in the TLB. These page walks could have been avoided if the lease length was long enough to cover the time between the last page walk due a *true* TLB miss (here, PageWalk_{m1}) and the most recent walk (PageWalk_{e1}). We call this the minimum desirable lease length. To avoid the second shootdown (Shootdown_1), the lease length should be shorter than the time between the latest page walk (PageWalk_{m2}) and the shootdown to ensure that the TLB entry has already expired. We call this the maximum desirable lease length. Ideally, one would pick a

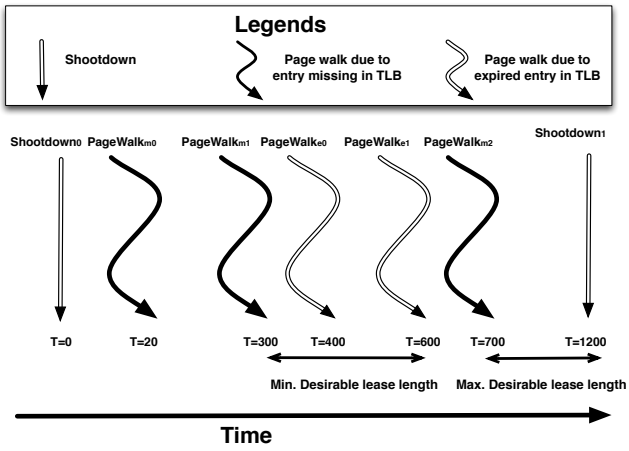


Figure 6. Desirable lease length for a given PTE.

lease length between the minimum and maximum values. Note that the minimum and maximum desirable values may not always overlap, and therefore it may not be possible to simultaneously minimize both additional page walks and shootdowns. Because shootdowns are orders of magnitude slower than page table walks, it is reasonable to avoid shootdowns even at the cost of a few additional TLB misses.

```
On invalidating a PTE for address A:
 IF (ExpirationTime(PTE_A) > CurrentTime) Then
 {   //Shorten the lease, C is constant
       Lease_A = (CurrentTime – LastPTWTS)/C
 }

On a Page Table Walk for address A:
   IF (Page walk due to expiration) Then
   {
      numConsecutiveTLBMiss++;
      IF (numConsecutiveTLBMiss > Th ) Then
      {  //Increase the lease length
            Lease_A = Max { (CurrentTime – LastPTWTS_Miss)*C' , Lease_A*C''}
            numConsecutiveTLBMiss = 0
      }
   }
   else
   {
         LastPTWTS_Miss = CurrentTime  //Updating timing information
   }
  LastPTWTS = CurrentTime
```

LEGEND
ExpirationTime(PTE_A) → Latest expiration time for PTE of address A
CurrentTime → Current *logical* time
$Lease_A$ → Lease length for PTE of address A
numConsecutiveTLBMiss → Num. of consecutive TLB misses due to expiration
$LastPTWTS_{Miss}$ → Timestamp of latest page walk due to missing TLB entry
LastPTWTS → Timestamp of latest page walk for either missing or expired entry

Figure 7. Algorithms for dynamic lease length assignment.

Based on the above observation that the lease length should be maintained within the minimum and maximum desirable range, the algorithms described in Figure 7 dynamically adjust lease lengths based on a program's observed behaviors. On a PTE invalidation, if the OS finds that the copies of the PTE in TLBs may not have expired, it lowers the lease length following the insight from the Figure 6 (typically, constant C=2). When the hardware page table walker observes several consecutive page walks due to expired entries, it correspondingly increases the lease length. We empirically found that a threshold Th=16 is effective to ensure that the algorithm favors the reduction of shootdowns even at the cost of a few extra page table walks.

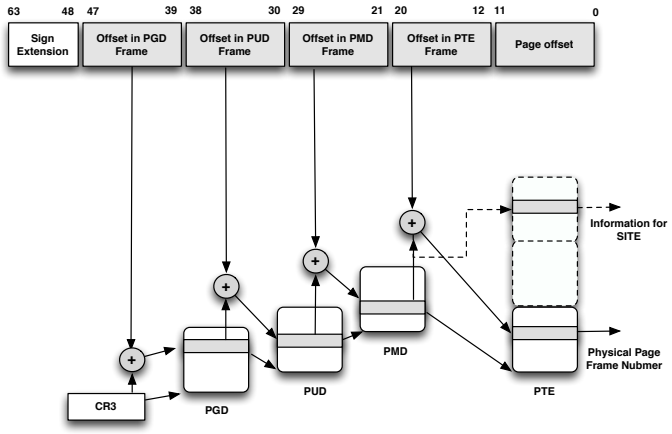


Figure 8. Lookup of Expiration Time Table (ETT) as an extension to x86-64 page table. Entry for an ETT is outlined in dotted lines. In x86-64, CR3 keeps the pointer to the root of the page table.

3) Storing Expiration Times and Lease: Our SITE-based system needs to store the latest expiration time of TLB entries corresponding to a given PTE. Note that because cache latencies are higher than TLB lookup times, the extra latency of book-keeping for the expiration times is completely hidden. Furthermore, for dynamic lease length assignment, additional information (Figure 7) needs to be kept at a per-page granularity. This information is comprised of the latest expiration time [2], current lease length, and timestamps for the latest *true* page table walks, and that for any page walk ($LastPTWTS_{Miss}$ and LastPTWTS in Figure 7, respectively). At 32 bits per timestamp, this amounts to 16 bytes of storage per page, which is less than 0.31% overhead for 4KB pages (and lower still for large pages).

The PTE format in modern page tables barely has any free bits available [6], and so we propose to store the SITE information in an auxiliary in-memory data structure called the Expiration Time Table (ETT). To avoid constructing a whole new page-table-like structure, the ETT reuses the non-leaf page-table nodes as shown in Figure 8 (based on the x86-64 page-table structure). The sharing of top-level/non-leaf nodes reduces the number of memory accesses to look up the ETT. Each 8-byte PTE entry is paired with 16 bytes of SITE information, and so for each page of PTE information in the original page table, we interleave two additional pages to store the ETT entries[3]. With this layout, we can simply

[2]Section VI discusses how overflow can be handled

[3]At first glance, the overhead of the ETT may appear unattractive because it amounts to an effective page-table size increase by $\sim 2\times$ (less due to the sharing of the non-leaf nodes). This can be easily reduced by re-encoding the various ETT timestamps because their upper bits will almost always be identical, and so there are well-known encoding optimizations that can be applied. However, even with an unoptimized additional 16 bytes per ETT entry, page tables are very small compared to the corresponding application's memory footprint to begin with, and so we do not bother optimizing the ETT size here (less than 0.31% overhead).

reuse the calculation of the final PTE entry with a modified offset to retrieve the desired ETT entry. A 16-byte ETT-entry can be atomically updated by page table walker.

As shown in Figure 8, a page table walk proceeds as unchanged for any non-leaf nodes of the page table. If SITE is enabled, however, then on reaching the last non-leaf node it looks up the corresponding entry in the ETT, *in parallel* to looking up the leaf-level of the page table. Note that, a PTE is 8 bytes long and thus, OSes pack 512 PTEs in a 4KB page at the leaf level of a page table. Because each ETT entry is 16 bytes, the OS could allocate two more contiguous pages next to the page holding the PTE[4].

IV. Implementation: Putting it All Together

We now describe the hardware and software modifications necessary to implement one possible embodiment of a SITE-based system.

A. Hardware Modifications

TLB Structure and Lookup: SITE extends TLB entries to hold their expiration times (32 bits). This adds state overhead less than the TLB tag overhead. However, a TLB's tag array does not contribute much to a processor's area budget and thus, this extension barely impacts a processor's overall die area. The TLB lookup process in SITE needs to compare the expiration time of a given TLB entry with the current logical time (here, number of DRAM accesses). On a tag match in a TLB, a hit is signaled only if its expiration time is greater than the current logical time. The tag and expiration time lookups occur in parallel and do not significantly impact the latency of a TLB access.

Also note that the overheads of SITE are minor compared to previously-proposed temporal cache coherence works that use the idea of self-invalidation for caches as they need to extend each cache block with a timestamp and alter the hardware cache coherence protocol [45], [43].

Hardware Page Table Walker: The hardware page table walker (PTW) for SITE has three additional responsibilities. ① The PTW assigns an expiration time to each address translation it returns to TLBs. The PTW does so by adding the current logical time and the lease value for the PTE (found in the ETT) as described in Section III-A. ② Next, the PTW needs to store the expiration time in the corresponding entry in the ETT. This becomes the latest expiration time for that given PTE. At the same time, the PTW updates the statistics needed for the dynamic lease assignment as described in Figure 7, which may cause ③ the PTW to increase the lease length for later accesses.

All of the above steps need the PTW to access the ETT. As depicted in the Figure 8, the page table and the ETT share the top levels or non-leaf nodes, and thus nothing extra needs to be done there (effectively re-using the existing PTW operations). While accessing the leaf level of the page table, the corresponding ETT entry is concurrently accessed. Thus, we do not expect page table walk latencies to increase significantly due to this extension, but we have accounted for the possibility of longer page walks for SITE by assuming a very conservative 33% latency increase in our evaluations.

Per-core Logical Time: We use the main memory access count as the logical clock. Memory controllers can easily keep count of memory accesses. However, on every load or store, a core cannot query the memory controller for the current count. Therefore in our implementation, we add a register to each core to hold the current logical time (i.e., memory access count here). On each DRAM access, the memory controller broadcasts a special message over the cache coherence network (similar to a coherence control message) to update the per-core registers. In systems with multiple memory controllers, each controller independently sends such messages to the cores. This is correct because the per-core logical timestamp (register value) is what is used in SITE, not the counts in individual memory controllers (more on scalability in Section VI-B). Also note that these update messages are sent to cores (not caches) at a negligible rate compared to cache coherence messages that keep the private cache hierarchies coherent in multi-core systems. We empirically measured that this adds approximately only 3% overhead on coherence bandwidth, which can be further reduced, if necessary (Section VI-B).

B. OS Modifications

SITE is an OS-hardware co-designed system. Below, we describe the OS enhancements needed to support SITE.

PTE-invalidation Routine: The OS's *TLB shootdown routine* is invoked to invalidate a PTE (Section II). This routine is enhanced to avoid actually performing costly shootdowns, when possible. Specifically, it looks up the entry in the ETT corresponding to the PTE to be invalidated. If the expiration time in that entry is less than the current logical time (memory access count), then the shootdown is avoided and the routine returns immediately. If not, the default path of TLB shootdown is followed as shown in Figure 1.

This routine also needs to decrease the lease length as per the dynamic lease assignment policy in Figure 7 and update the corresponding ETT entry. This does not add any observable overhead as PTE invalidation routines are already several thousands of cycles long.

Allocating ETTs and Page Fault Handler: At the time of allocating a PTE to map a newly allocated memory page, the corresponding entry in the ETT is also created and initialized. The page fault handler is easily extended for this purpose. This adds an insignificant overhead as the latency of page faults is already in the order of several micro-seconds [42].

[4]Note that PTEs can be overlapped with SITE information in such that one cache line access is sufficient to obtain the PTE and SITE information. We leave such optimizations as future work.

Table I
CONFIGURATION OF THE SIMULATED SYSTEM.

Processor and Caches	
CPU	8 in-order cores, single hardware thread per-core
L1 TLB	32-entry fully associative per core (1 cycle hit latency)
L2 TLB	256-entry 8-way associative per core (10 cycles hit latency)
Page Walk Latency	150 cycles page-walk (L2 TLB miss), 50 extra cycles for SITE
L1 Cache	private, 1 cycles, 32KB, 4-way, 64B block
L2 Cache	private, 10 cycles, 256KB, 8-way, 64B block
L3 Cache	shared, 25 cycles, 8MB, 16-way, 64B block
Coherence	MOESI protocol
Main Memory	
Fast to Slow Memory Capacity Ratio	1:8
Fast Memory Latency	150 cycles
Fast Memory Replacement	Clock replacement policy [48]
Slow Memory Latency	600 cycles
Page Migration Overhead	
TLB Shootdown Latency	Issuing core 20,000 cycles, Receiving core 5,000 cycles
Page Copy Latency	5000 cycles
Page Migration Size	4KB

Table II
APPLICATIONS

BT	Block tri-diagonal solver (NAS benchmark).
CG	Conjugate gradient kernel (NAS benchmark).
FT	Discrete 3D fast Fourier transform (NAS benchmark).
LU	Lower-Upper Gauss-Seidel solver (NAS benchmark).
LULESH	Solve a simple Sedov blast problem for hydrodynamic (ProxyApp)
MINIFE	Finite Element mini-application (ProxyApp)
RSBENCH	A multi-pole resonance representation lookup cross section algorithm (ProxyApp)
XSBENCH	A computational kernel of the Monte Carlo neutronics application OpenMC (ProxyApp).

V. EVALUATION AND ANALYSIS

In this section, we evaluate the ability of SITE to reduce the cost of TLB shootdowns in a heterogeneous memory system. However, SITE is more broadly applicable to other use cases like copy-on-write and garbage collection that critically depends upon TLB shootdowns (discussed in Section II).

A. Simulation Model

Similar to prior related work [51], [37], [33], to evaluate SITE, we used trace-based simulation using the PIN toolset [39]. The simulator models detailed three levels of caches, two levels of TLBs, and keeps the caches coherent using a MOESI coherence protocol. We model heterogeneous memory comprising of fast (DRAM) and slow (PCM) memory. The ratio of capacity between these two types of memory is 1:8. We then modeled our threshold-based page migration policy where a page is migrated to the fast memory only after a given number of accesses (threshold) to a page residing in slow memory. Unless mentioned otherwise, we use a page-migration threshold of 10 for the rest of the evaluation. We also model detailed page migration and TLB shootdown costs. Table I details the configuration of the system we modeled.

Using such a PIN-/trace-based simulation allowed us to simulate workloads with large memory footprints (more than 1 GB) and run the applications for long run times. Both of these are important to reliably characterize a system with heterogeneous memory. We then use the statistics from this simulation infrastructure to feed a detailed performance model. At a high level, it models three types of latency. ① Non-load/store instructions are assumed to execute in one cycle. ② Data access costs for load/store instructions are calculated using hit/miss counts at the three levels of caches and their corresponding hit/miss latencies as listed in Table I. Accesses that miss in the last-level cache incur a latency to either fast or slow memory depending upon the current memory mapping. ③ Address translation costs for each load/store instruction are calculated by combining L1 and L2 TLB hit/miss rates with their corresponding latencies and the latency of page table walks (Table I). For the baseline without self-invalidating TLB entries, we assumed a page table walk latency of 150 cycles. However, in case of SITE, we conservatively increase this by 33% to 200 cycles to model any extra latencies associated with ETT accesses. This is very conservative because it is possible for the page table walker to access the page table leaf node and ETT entry in parallel. ④ Finally, we calculate the cost of migrating a page from slow memory to fast memory by taking into account the latency of copying the page between memories and the cost of the TLB shootdown. For TLB shootdown cost, we model it accurately assuming a larger latency incurred at the initiator (issuer) core of the shootdown and $\frac{1}{4}^{\text{th}}$ that latency at the receiving cores. The issuer core needs to wait for all other cores to acknowledge the shootdown while receivers do not (Section II and Figure 1), hence this asymmetry. The latencies and ratio of asymmetry are similar to those reported in previous works [40], [51].

B. Workloads

We evaluate SITE using eight multi-threaded applications as listed in Table II. We choose these workloads from the NAS parallel benchmark suite [10] and from publicly released HPC proxy applications from the U.S. Department of Energy [18], [3], [50], [49]. The applications are chosen such that they are memory intensive, a requirement to exercise systems with heterogeneous memory in any meaningful way. We use input set size of *C* for NAS workloads, and the memory footprint of each proxy application is around 1.25 GB.

C. Performance Evaluation

In the evaluation, we seek to answer the following questions: ① What performance improvement can SITE achieve by avoiding TLB shootdowns? ② What are the sources of improvements (degradations)? ③ What is the sensitivity of the evaluation against varying parameters such as the

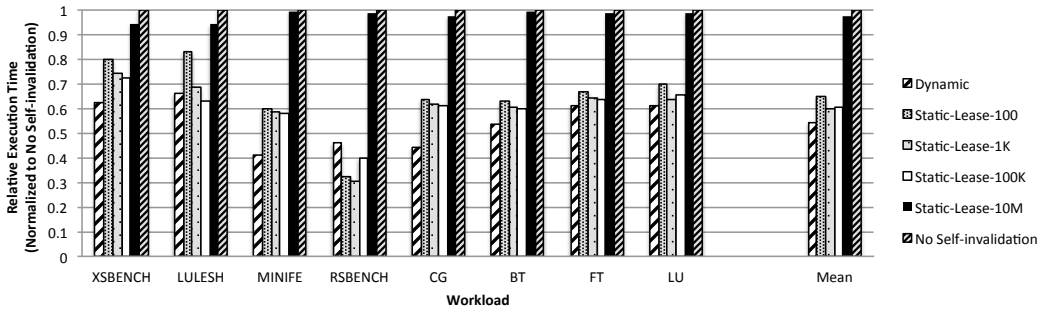


Figure 9. The impact of SITE on performance of a system with heterogeneous memory (page migration threshold = 10).

migration threshold, shootdown latency, page table walk latency, and slow memory access latency?

Figure 9 shows how execution time of different applications under page migration (migration threshold = 10) improves with SITE. The y-axis shows the execution time normalized to a baseline with no self-invalidating TLB (lower is better). There is a cluster of bars for each application listed along the x-axis. The right-most bar in each cluster represents the baseline (no self-invalidation). The left-most bar shows the normalized execution time with SITE employing the dynamic lease assignment algorithm (*Dynamic lease*) described in Figure 7. The bars in the middle of each cluster represent SITE with static lease length assignment. For example, *Static-Lease-1K* represents SITE employing a constant lease length of 1K units of logical time. We make two observations from Figure 9. ① SITE greatly helps performance, with dynamic lease assignment (*Dynamic*) reducing the execution time by *45.5%* on average across all workloads. This is expected though, as it is well known that the majority of overhead from page migration is due to TLB shootdowns [37], [31], [7], and the benefit of migration can be negated by such overheads. Because SITE avoids many shootdowns, it helps performance significantly. ② The dynamic lease algorithm helps across all workloads. Except RSBench, the dynamic algorithm outperforms the best of all static lease policies evaluated by adjusting lease on a per-page basis [5]. Even if a particular static lease length performs better for a given workload, it is not known a priori what this preferred lease length is. Therefore, dynamic lease assignment is crucial for an effective SITE-based solution.

Next, we analyze the above-mentioned performance numbers. Figure 10 shows the number of TLB shootdowns normalized to the baseline for each configuration in the figure 9. The y-axis of Figure 10 shows the number of shootdowns normalized to that in the baseline (lower is better). The x-axis is same as the previous figure for execution times. We see that a large fraction of TLB shootdowns is avoided by

[5]RSBench prefers a very short lease length, but our dynamic algorithm was less aggressive and did not reduce the fraction of shootdowns by as much compared to the best static lease values.

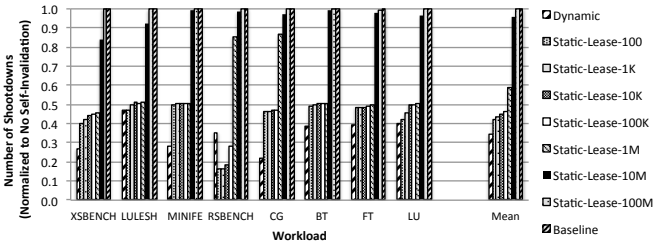


Figure 10. Normalized TLB shootdown count with varying lease assignment policy of SITE.

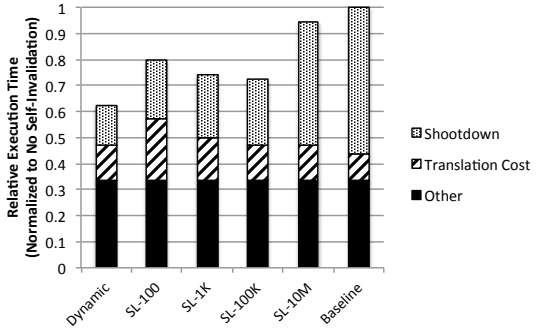


Figure 11. XSBench's execution time breakdown.

SITE with both dynamic and static lease assignment policies. For example, on average across all workloads, 65.2% of the shootdowns are avoided by SITE with dynamic lease assignment. While SITE can avoid TLB shootdowns, it can increase address translation costs by incurring additional TLB misses due to expired leases. To understand how these two opposing trends impact the overall execution time, we dig into one of the representative workload XSBench in Figure 11. The y-axis shows the execution time normalized to the baseline. There are stacked bars for different lease assignment policies of SITE and for the baseline. Each bar is broken into three parts (stacks) showing the fraction of execution time spent on TLB shootdown (*Shootdown*), the

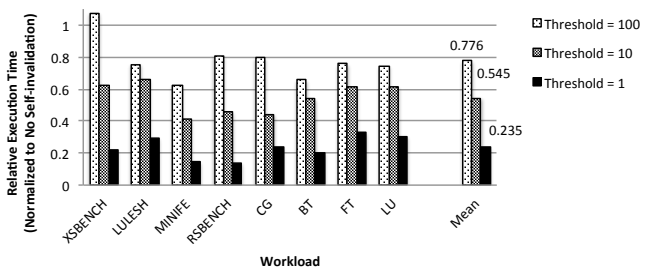


Figure 12. The impact of migration threshold on SITE's performance.

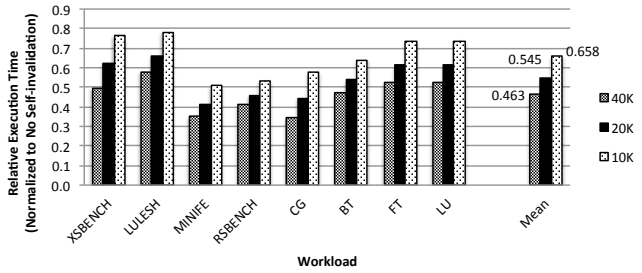


Figure 13. The impact of TLB shootdown latency on SITE's performance.

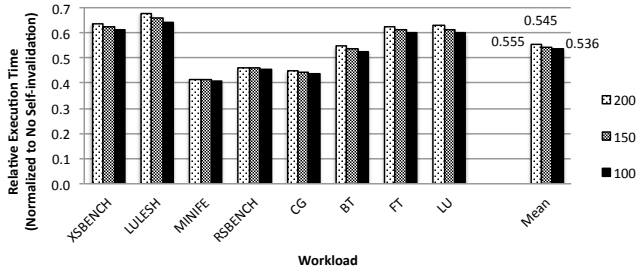


Figure 14. Impact of page table walk latency on SITE's performance.

fraction of time spent on address translation (*Translation cost*), and the rest that includes data access, page copy costs, and computation cost (*Others*). From the figure, we observe that a large fraction of the execution time in the baseline is attributed to the overhead of TLB shootdowns due to page migrations. This is in line with the findings in previous works [37], [31], [7]. This fraction, however, decreases significantly with SITE. Reduction is more significant with shorter leases and best with dynamic lease assignment. The fraction for address translation is much smaller compared to that for TLB shootdowns in the baseline because the cost of shootdowns are orders of magnitude larger than the cost of a page table walk. However, this fraction grows as the lease length decreases, but not enough to negate the advantage of the lower shootdown costs until a very small static lease length of *1K*. On the other hand, the dynamic lease assignment policy for SITE does a good job of balancing the lease length to achieve significant overall performance improvement. We omit such breakdown for other workloads due to lack of space and because they show the same overall trends.

D. Sensitivity Studies

Next, we evaluate the resiliency of our SITE-based solution against several key design parameters.

Impact of Migration Threshold: In a system with heterogeneous memory, lowering the threshold for the number of accesses to a page in slow memory before it is migrated to fast memory (migration threshold) can increase the hit rate in fast memory, but it also increases the number of shootdowns. Figure 12 shows how SITE (with dynamic lease assignment) fares with migration thresholds of 1 (first touch), 10, and 100 (Threshold=1, 10, 100, respectively, in the figure). Lowering the threshold increases the number of shootdowns in the baseline due to aggressive page migration and thus offers more opportunity for SITE to avoid shootdowns. Consequently, the efficacy of SITE increases significantly with lowering migration threshold as shown in the figure.

Impact of TLB Shootdown Latency: SITE avoids TLB shootdowns and thus changes in shootdown latency affect its efficacy. Previously, we assumed a shootdown cost of 20K cycles to the initiator (issuer) core and $\frac{1}{4}$ th (i.e., 5K cycles) at receiving cores (Table I). In Figure 13, we varied the shootdown latency to 10K cycles and to 40K cycles for the initiator core while maintaining the same ratio to the latency for the receiving cores. As expected, the efficacy of SITE increases or decreases with an increase or decrease in the shootdown latency. However, even with a relatively small shootdown latency of 10K, SITE still reduces the average execution time of applications by 34.2%.

Impact of Page Table Walk Latency: SITE can increase the number of page walks, and thus we vary page walk latencies to see how that impacts SITE's performance. Previously, we assumed a page walk latency of 150 cycles and 200 cycles (33% more) for SITE. We vary the baseline page walk latency to 100 and 200 cycles, while also proportionally increasing the page walk latency for SITE. In Figure 14, we present execution times of SITE (with dynamic lease length), normalized to the baseline (lower is better) with these variations. As expected, higher page walk latency decreases efficacy of SITE, but it still remains potent even at a page walk latency of 200 cycles with about 45% reduction in execution time on average.

Impact of Slow Memory Latency: Previously, we assumed the slow memory in our heterogeneous memory system is 4× slower than the fast memory (Table I). Figure 15 shows the impact of varying the latency ratio between fast and slow memory (2× and 8×). We observe that the efficacy of SITE does not alter with varying the latency ratio of fast and slow

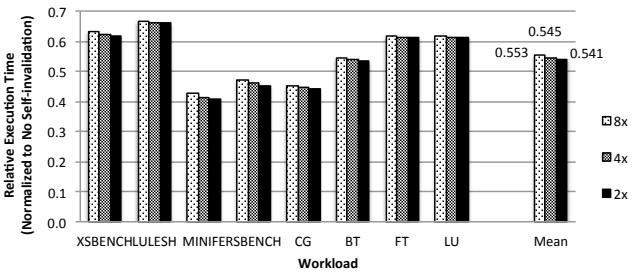


Figure 15. Impact of the latency ratio of slow to fast memory on SITE.

memory.

VI. Discussion

In this section, we discuss a few topics related to the idea of SITE and its proposed implementation.

A. Energy Efficiency of SITE

SITE extends each TLB entry with a 32-bit expiration time. On every TLB lookup, SITE also needs to compare this expiration time with the current logical time to confirm that the entry has not yet expired. Because the expiration time can be looked up in parallel with the TLB tag, this does not significantly impact latency, but it adds to the dynamic energy consumption. However, industry projections show that the TLB contributes about 6% of a chip's power [46]. Thus, SITE is unlikely to add much to the overall energy consumption of the chip.

More importantly however, SITE is likely to significantly reduce static energy. Static energy (leakage) consumption is proportional to the execution time, and SITE can reduce execution times significantly (e.g., by up to 65%). As the technology feature size for processors shrinks, the transistors tend to leak more current, and therefore the static energy can become a larger challenge [24]; static energy already contributes more than 30-40% of some chips' energy even at the 22nm technology node [8]. Thus, SITE is likely to help reduce the net energy consumption of a system.

B. Scalability of SITE

SITE needs a global logical clock to ascertain when TLB entries have expired, and so SITE's overall scalability is determined by the scalability of this clock. If the clock progresses too fast, then keeping the clock synchronized is challenging. Thus, as discussed in Section III-A1, we avoided using physical clocks and used the main memory access count as the logical clock.

Here, we make three additional observations that could further enhance the scalability of SITE for larger systems. (1) We observed that the best-performing lease lengths across the applications are in 100s or 1000s of main memory accesses (Figure 9) because the shootdowns happen relatively infrequently. Therefore, SITE's performance is unlikely to be impacted if the logical clock advances only after every 10 or even 100 main memory accesses. Advancing the clock only in 10 or 100 memory accesses reduces the need to synchronize the clock by 10 or 100 times. (2) Further, we observe that SITE can relax the need for keeping all local, per-core logical clocks *completely synchronized*, as long as the maximum drift among them is bounded. SITE then avoids a shootdown only if the expiration time of the corresponding PTE is in the past by more than the maximum allowed drift. Drift of n logical ticks can be bounded by ensuring that the logical clocks are not advanced if the number of yet-to-be-acknowledged updates to the per-core clock is limited to the maximum of n. (3) Finally, SITE does not break even if logical clocks temporarily stop advancing. The logical clock needs to progress fast enough for the TLB entries to expire but *not* necessarily on exactly every main memory access, or on every 10 or 100 memory accesses.

In summary, we find that SITE can be scaled well to larger systems using one or a combination of above-mentioned techniques.

C. Timestamp Overflow

Our proposed implementation of SITE uses 32-bit logical timestamps that would overflow at a very slow rate. We estimate that even with a sustained high memory bandwidth demand of 20 GB/sec, the timestamp counter would overflow in 13 seconds, assuming the timestamp is incremented on every memory access. In case of the rare overflow, we propose to simply flush all TLBs and invalidate all ETTs.

VII. Related Work

In this section, we summarize the most related work.

Self-Invalidation: The concept of self-invalidation has been explored in other contexts, such as in cache coherence protocols [35], [43], [25], [27], [26], [47], [45], in cache replacement [23], [32], and in cache power management [5], [17], [22]. Min et al. proposed a timestamp-based software-assisted cache coherence protocol [35]. Lebeck et al. proposed to proactively invalidate shared cache blocks through dynamic self-invalidation (DSI) to eliminate invalidation messages [27]. Shim et al. improved DSI by delaying writes until all shared copies are expired [43]. Lai et al. proposed a Last-Touch Predictor to improve the accuracy of predicting the shared blocks to be proactively invalidated [26]. Somogyi et al. proposed predicting the last store to cache blocks via PC addresses [47]. Employing self-invalidation to reduce cache coherence overhead has also been extended for GPUs and accelerators [25], [45]. In the context of cache replacement, various studies have looked into identifying dead blocks as potential replacement victims, using counters events [23] or cache bursts [32]. Other work proposed using self-invalidation to reduce leakage power by turning off dead cache blocks [5], [17], [22].

To the best of our knowledge, this is the first work to propose using self-invalidation for TLB entries. Not only is

the context new, but the challenges are unique because TLBs, page table walkers, and TLB shootdowns are not entirely managed by the hardware, and the OS plays an important role in these. For example, there is no hardware-enforced TLB coherence in commercial processors and the OS needs to get involved. Similarly, the OS allocates entries in the page table, which is then looked up by the hardware page table walker. This results in a significantly different design space compared to past work on conventional caches.

Reducing TLB Shootdown Cost: This category of work is the most related to our work, and includes the following studies. Romanescu et al. [40] proposed hardware coherence support for TLBs [40]. While feasible, hardware coherence for the TLBs may be an overkill because PTEs are updated much less frequently compared to data caches, and hardware coherence may add complexity to the core TLB structures and circuitry. Cache coherence is already tedious to validate due to its complex state machine [9]; TLB coherence is likely to add to that. Villavieja et al. [51] proposed a shared hardware TLB directory (DiDi) that is inclusive of all TLB entries in the system. DiDi helps filter extraneous shootdowns to cores without cached copies of the PTE. The centralized nature of this hardware-based solution, and the fact that the proposed shared TLB needs to be inclusive to all other TLBs, makes it difficult to scale to a larger number of cores and to large per-core TLBs that are becoming more common. It is not clear how it can be adapted to handle multi-socket (SMP) systems, either. Oskin and Loh [37] proposed hardware-assisted TLB shootdowns to reduce inter-processor interrupt latency, but the overheads are still large, i.e., in the microsecond range. Our proposed self-invalidating TLB scheme is distributed, and scalable to implement, without relying on any centralized hardware structures. Recent versions from ARM processors started to support an instruction called *TLBi*, which submits a request to globally invalidate a specific TLB entry; however, it must be followed by expensive global barriers, *DSB*. The costs of such global barrier are expected to be extremely expensive for large numbers of cores. SITE aims to eliminate the need for TLB shootdown, whereas TLBi tries to minimize the cost of TLB shootdown. Accordingly, SITE can be combined with shootdown overhead optimizations. Furthermore, x86 systems are the most dominant for server market, where IPI-based shootdown is still required.

Heterogeneous Memory Systems Management: There are many prior studies exploring the management of page placement in heterogeneous memory systems. Some studies relied on a hardware-only management approach [20], [21], [29], [11], [41], [44]. While transparent to system software, hardware caching is not without its costs. First, the hardware cache typically requires non-trivial resources for bookkeeping (i.e., the tags) that either require the construction of large on-chip structures (SRAM) that increases chip costs, and/or cannibalizes part of the fast memory capacity to store the bookkeeping state, thereby reducing the effective size of the faster memory resource. Furthermore, by making the fast memory software-transparent, the total system memory capacity is effectively reduced. An alternative to hardware-only management is a software management approach. Software management can be categorized into OS-based and application-explicit management. OS-based management approaches focus on migration policies and improving performance via prefetching [37], [7], [33]. In application-level management work, the application has the flexibility to choose where to allocate memory objects and how to migrate pages across different memory technologies. An example of that is Memkind [12], which is a user-level heap manager that provides programmers with different memory services. Similarly, Meswani et al. proposed explicit management of two-level memory through explicit calls from applications [34].

Our work investigates a complementary and crucial question of reducing the cost of TLB shootdowns. Lowering the number and cost of TLB shootdowns is a crucial enabling technology for the heterogeneous memory management discussed above.

VIII. Conclusion

We introduce the concept of self-invalidating TLB entries (SITE) to avoid TLB shootdowns. While previous works in this domain attempt to make TLB shootdowns faster, this work avoids shootdowns at the risk of additional TLB misses. Since shootdowns are typically two order of magnitudes or more slower than TLB misses, this trade-off often comes out beneficial. We further propose to dynamically adjust lease length of TLB entries to ensure this trade-off remains beneficial for overall application performance even when shootdowns are not common. However, future work could unlock more performance by adjusting lease lengths more aggressively.

The scalability of SITE to higher number of cores is limited by the scalability of the logical clock it employs. Our proposal to use memory access counts as logical clock works well for moderately sized systems and for our use case of systems with heterogeneous memory. However, for larger systems with several sockets, an alternative design that employs logical clock may work better. Future research can explore how hierarchical clocks can be used by SITE to scale to large number of sockets.

IX. Acknowledgment

AMD®, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

REFERENCES

[1] HMC Consortium, http://www.hybridmemorycube.org/technology.html.

[2] Huai, Yiming, et al. Observation of spin-transfer switching in deep submicron-sized and low-resistance magnetic tunnel junctions. Applied Physics Letters 84.16: 3118-3120, 2004.

[3] Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory. Technical Report LLNL-TR-490254.

[4] Jedec specifications for hbm, jesd235a, http://www.jedec.org/standards-documents/docs/jesd235a.

[5] Jaume Abella, Antonio González, Xavier Vera, and Michael FP O'Boyle. Iatac: a smart predictor to turn-off l2 cache lines. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2(1):55–77, 2005.

[6] AMD. *AMD64 Architecture Programmer's Manual, Volume 2, Revision 3.23*.

[7] Amro Awad, Sergey Blagodurov, and Yan Solihin. Write-aware management of nvm-based memory extensions. In *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, pages 9:1–9:12, New York, NY, USA, 2016. ACM.

[8] Y. Bai, Y. Song, M. N. Bojnordi, A. Shapiro, E. G. Friedman, and E. Ipek. Back to the future: Current-mode processor in the era of deeply scaled cmos. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(4):1266–1279, April 2016.

[9] Y. Bai, Y. Song, M. N. Bojnordi, A. Shapiro, E. G. Friedman, and E. Ipek. Back to the future: Current-mode processor in the era of deeply scaled cmos. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(4):1266–1279, April 2016.

[10] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Robert L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. The nas parallel benchmarkssummary and preliminary results. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 158–165. ACM, 1991.

[11] Evgeny Bolotin, David Nellans, Oreste Villa, Mike O'Connor, Alex Ramirez, and Stephen W Keckler. Designing efficient heterogeneous memory architectures. *IEEE Micro*, 35(4):60–68, 2015.

[12] Christopher Cantalupo, Vishwanath Venkatesan, and Jeff R Hammond. User extensible heap manager for heterogeneous memory platforms and mixed memory policies. 2015.

[13] Perry Cheng and Guy E. Blelloch. A parallel, real-time garbage collector. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 125–136, New York, NY, USA, 2001. ACM.

[14] JaeWoong Chung, Chi Cao Minh, Austen McDonald, Travis Skare, Hassan Chafi, Brian D. Carlstrom, Christos Kozyrakis, and Kunle Olukotun. Tradeoffs in transactional memory virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 371–381, New York, NY, USA, 2006. ACM.

[15] Rob Crooke and Al Fazio. Intel Non-Volatile Memory Inside. The Speed of Possibility Outside. In *Intel Developer Forum (IDF), 2015*.

[16] D. Dhurjati and V. Adve. Efficiently detecting all dangling pointer uses in production servers. In *International Conference on Dependable Systems and Networks (DSN'06)*, pages 269–280, June 2006.

[17] Krisztián Flautner, Nam Sung Kim, Steve Martin, David Blaauw, and Trevor Mudge. Drowsy caches: simple techniques for reducing leakage power. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA), 2002*, pages 148–157. IEEE, 2002.

[18] Michael Heroux, Douglas Doerfler, Paul Crozier, James Willenbring, Carter Edwards, Alan Williams, Mahesh Rajan, Eric Keiter, Heidi K Thornquist, and Robert W Numrich. Improving performance via mini-applications. In *Sandia Report 2009*.

[19] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual 2012*.

[20] Hakbeom Jang, Yongjun Lee, Jongwon Kim, Youngsok Kim, Jangwoo Kim, Jinkyu Jeong, and Jae W Lee. Efficient footprint caching for tagless dram caches. In *Proceedings of IEEE International Symposium on High Performance Computer Architecture (HPCA), 2016*, pages 237–248. IEEE, 2016.

[21] Djordje Jevdjic, Gabriel H Loh, Cansu Kaynak, and Babak Falsafi. Unison cache: A scalable and effective die-stacked dram cache. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 25–37. IEEE, 2014.

[22] Stefanos Kaxiras, Zhigang Hu, and Margaret Martonosi. Cache decay: exploiting generational behavior to reduce cache leakage power. *ACM SIGARCH Computer Architecture News*, 29(2):240–251, 2001.

[23] Mazen Kharbutli and Yan Solihin. Counter-based cache replacement algorithms. In *Proceedings of the 2005 IEEE International Conference on Computer Design (ICCD), 2005*, pages 61–68. IEEE, 2005.

[24] Nam Sung Kim, Todd Austin, David Blaauw, Trevor Mudge, Krisztián Flautner, Jie S. Hu, Mary Jane Irwin, Mahmut Kandemir, and Vijaykrishnan Narayanan. Leakage current: Moore's law meets static power. *Computer*, 36(12):68–75, December 2003.

[25] Snehasish Kumar, Arrvindh Shriraman, and Naveen Vedula. Fusion: design tradeoffs in coherent cache hierarchies for accelerators. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 733–745. ACM, 2015.

[26] An-Chow Lai and Babak Falsafi. Selective, accurate, and timely self-invalidation using last-touch prediction. In *Proceedings of the 27th International Symposium on Computer Architecture (ISCA), 2000*, pages 139–148. IEEE, 2000.

[27] Alvin R Lebeck and David A Wood. Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors. In *ACM SIGARCH Computer Architecture News*, volume 23, pages 48–59. ACM, 1995.

[28] Benjamin Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. In *International Symposium on Computer Architecture*, 2009.

[29] Yongjun Lee, Jongwon Kim, Hakbeom Jang, Hyunggyun Yang, Jangwoo Kim, Jinkyu Jeong, and Jae W Lee. A fully associative, tagless dram cache. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 211–222. ACM, 2015.

[30] Z. Li, R. Zhou, and T. Li. Exploring high-performance and energy proportional interface for phase change memory systems. In *IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 210–221, Feb 2013.

[31] Felix Xiaozhu Lin and Xu Liu. Memif: Towards programming heterogeneous memory asynchronously. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 369–383. ACM, 2016.

[32] Haiming Liu, Michael Ferdman, Jaehyuk Huh, and Doug Burger. Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, pages 222–233, 2008.

[33] Mitesh R Meswani, Sergey Blagodurov, David Roberts, John Slice, Mike Ignatowski, and Gabriel H Loh. Heterogeneous memory architectures: A hw/sw approach for mixing die-stacked and off-package memories. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 126–136. IEEE, 2015.

[34] Mitesh R Meswani, Gabriel H Loh, Sergey Blagodurov, David Roberts, John Slice, and Mike Ignatowski. Toward efficient programmer-managed two-level memory hierarchies in exascale computers. In *Hardware-Software Co-Design for High Performance Computing (Co-HPC), 2014*, pages 9–16. IEEE, 2014.

[35] Sang Lyul Min and J-L Baer. Design and analysis of a scalable cache coherence scheme based on clocks and timestamps. *IEEE Transactions on Parallel and Distributed Systems*, 3(1):25–44, 1992.

[36] Prashant J. Nair, Dae-Hyun Kim, and Moinuddin K. Qureshi. Archshield: Architectural framework for assisting dram scaling by tolerating high error rates. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 72–83, New York, NY, USA, 2013. ACM.

[37] Mark Oskin and Gabriel H. Loh. A software-managed approach to die-stacked dram. In *PACT*, pages 188–200. IEEE, 2015.

[38] Luiz E Ramos, Eugene Gorbatov, and Ricardo Bianchini. Page Placement in Hybrid Memory Systems. In *Proceedings of the International Conference on Supercomputing (ICS)*, pages 85–95. ACM, 2011.

[39] Vijay Janapa Reddi, Alex Settle, Daniel A Connors, and Robert S Cohn. Pin: a binary instrumentation tool for computer architecture research and education. In *Proceedings of the 2004 workshop on Computer architecture education: held in conjunction with the 31st International Symposium on Computer Architecture*, page 22. ACM, 2004.

[40] Bogdan F Romanescu, Alvin R Lebeck, Daniel J Sorin, and Anne Bracy. Unified instruction/translation/data (unitd) coherence: One protocol to rule them all. In *Proceedings of IEEE 16th International Symposium on High Performance Computer Architecture (HPCA), 2010*, pages 1–12. IEEE, 2010.

[41] Jee Ho Ryoo, Karthik Ganesan, Yao-Min Chen, and Lizy Kurian John. i-mirror: A software managed die-stacked dram-based memory subsystem. In *Proceedings of the 27th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), 2015*, pages 82–89. IEEE, 2015.

[42] Mohit Saxena and Michael M Swift. FlashVM: virtual memory management on flash. In *Proceedings of the 2010 USENIX conference on USENIX Annual Technical Conference, 2010*, pages 14–14. USENIX Association, 2010.

[43] Keun Sup Shim, Myong Hyon Cho, Mieszko Lis, Omer Khan, and Srinivas Devadas. Library cache coherence. 2011.

[44] Jaewoong Sim, Gabriel H Loh, Vilas Sridharan, and Mike O'Connor. Resilient die-stacked dram caches. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 416–427. ACM, 2013.

[45] Inderpreet Singh, Arrvindh Shriraman, Wilson WL Fung, Mike O'Connor, and Tor M Aamodt. Cache coherence for gpu architectures. In *Proceedings of IEEE 19th International Symposium on High Performance Computer Architecture (HPCA2013), 2013*, pages 578–590. IEEE, 2013.

[46] Avinash Sodani. Race to exascale: Opportunities and Challenges. In *Keynote at the Annual IEEE/ACM 44th Annual International Symposium on Microarchitecture.*

[47] Stephen Somogyi, Thomas F Wenisch, Nikolaos Hardavellas, Jangwoo Kim, Anastassia Ailamaki, and Babak Falsafi. Memory coherence activity prediction in commercial workloads. In *Proceedings of the 3rd workshop on Memory performance issues: in conjunction with the 31st international symposium on computer architecture*, pages 37–45. ACM, 2004.

[48] Andrew S Tanenbaum. Modern operating systems. 2009.

[49] John R Tramm, Andrew R Siegel, Benoit Forget, and Colin Josey. Performance analysis of a reduced data movement algorithm for neutron cross section data in monte carlo simulations. In *Solving Software Challenges for Exascale*, pages 39–56. Springer, 2014.

[50] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. XSBench - The Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis. In *PHYSOR 2014 - The Role of Reactor Physics toward a Sustainable Future*, Kyoto.

[51] Carlos Villavieja, Vasileios Karakostas, Lluis Vilanova, Yoav Etsion, Alex Ramirez, Avi Mendelson, Nacho Navarro, Adrian Cristal, and Osman S. Unsal. Didi: Mitigating the performance impact of tlb shootdowns using a shared tlb directory. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 340–349, Washington, DC, USA, 2011. IEEE Computer Society.

[52] J Joshua Yang, Dmitri B Strukov, and Duncan R Stewart. Memristive devices for computing. *Nature nanotechnology*, 8(1):13–24, 2013.