Large Pages and Lightweight Memory Management in Virtualized Environments: Can You Have it Both Ways?

Binh Pham[§] Ján Veselý[§] Gabriel H. Loh[†] Abhishek Bhattacharjee[§]

[§] Department of Computer Science
Rutgers University
{binhpham, jan.vesely, abhib}@cs.rutgers.edu

[†] AMD Research
Advanced Micro Devices, Inc.
gabriel.loh@amd.com

ABSTRACT

Large pages have long been used to mitigate address translation overheads on big-memory systems, particularly in virtualized environments where TLB miss overheads are severe. We show, however, that far from being a panacea, large pages are used sparingly by modern virtualization software. This is because large pages often preclude lightweight memory management, which can outweigh their Translation Lookaside Buffer (TLB) benefits. For example, they reduce opportunities to deduplicate memory among virtual machines in overcommitted systems, interfere with lightweight memory monitoring, and hamper the agility of virtual machine (VM) migrations. While many of these problems are particularly severe in overcommitted systems with scarce memory resources, they can (and often do) exist generally in cloud deployments. In response, virtualization software often (though it doesn't have to) splinters guest operating system (OS) large pages into small system physical pages, sacrificing address translation performance for overall system-level benefits. We introduce simple hardware that bridges this fundamental conflict, using speculative techniques to group contiguous, aligned small page translations such that they approach the address translation performance of large pages. Our Generalized Large-page Utilization Enhancements (GLUE) allow system hypervisors to splinter large pages for agile memory management, while retaining almost all of the TLB performance of unsplintered large pages.

Categories and Subject Descriptors

C.1.0 [**Processor Architectures**]: General

Keywords

Virtual Memory, Virtualization, TLB, Speculation

MICRO 2015 Waikiki, Hawaii USA

DOI: http://dx.doi.org/10.1145/2830772.2830773

1. INTRODUCTION

With cloud computing, virtualization technologies (e.g., KVM, Xen, ESX, Docker, HyperV and others) are aggressively employed by companies like Amazon or Rackspace to consolidate diverse workloads (encapsulated in virtual machines or VMs) to ensure high utilization of physical systems while achieving good performance.

The judicious use of large pages [26,38,39] is particularly critical to the performance of cloud environments. Large pages can boost address translation performance in hypervisor-based virtualization and containers [11]. Specifically, in hypervisor-based virtualization, the hypervisor and guests maintain separate page tables, and therefore require high-latency two-dimensional page table walks on Translation Lookaside Buffer (TLB) misses. Past work has shown that this is often the primary contributor to the performance difference between virtualized and bare-metal performance [15, 18]. Large pages (e.g., 2MB pages instead of baseline 4KB pages in x86-64) counter these overheads by dramatically reducing the number of page table entries (PTEs), increasing TLB hit rates and reducing miss latencies. Even containers (which do not require two-dimensional page table walks) are dependent on large pages to improve TLB reach, which is otherwise inadequate to address the hundreds of GB to several TB of memory present on the systems containers are often deployed in [18].

Unfortunately however, the coarser granularity of large pages can curtail lightweight and agile system memory management. For example, large pages can reduce consolidation in real-world cloud deployments where memory resources are over-committed by precluding opportunities for page sharing [20,42]. They can also interfere with a hypervisor's ability to perform lightweight guest memory usage monitoring, its ability to effectively allocate and place data in non-uniform memory access systems [19, 44], and can hamper operations like agile VM migration [41]. As a result, virtualization software often (though it doesn't have) chooses to *splinter* or break large pages into smaller baseline pages. While these decisions may be appropriate for overall performance as they improve consolidation ratios and memory management, they do present a lost opportunity in terms of reducing address translation overheads.

This paper proposes hardware that bridges this fundamental conflict to reclaim the address translation performance opportunity lost by large page splintering.

Specifically, we observe that the act of splintering a large page is usually performed to achieve finer-grained memory management rather than to fundamentally alter virtual or physical address spaces. Therefore, the vast majority of constituent small pages retain the original contiguity and alignment in both virtual and physical address spaces that allowed them to be merged into large pages in the first place. In response, we propose **Generalized Large-page Utilization Enhancements (GLUE)** to identify splintered large page-sized memory regions. GLUE augments standard TLBs to store information that identifies these contiguous, aligned, but splintered regions. GLUE then uses TLB speculation to identify the constituent translations. Small system physical pages are speculated by interpolating around the information stored about a single *speculative large-page* translation in the TLB. Speculations are verified by page table walks, now removed from the processor's critical path of execution, effectively converting the performance of correct speculations into TLB hits. GLUE accurate, software-transparent, readily-implementable, and allows large pages to be compatible, rather than at odds, with lightweight memory management. Specifically, our contributions are:

- We characterize the prevalence of page splintering in virtualized environments. We find that large pages are conflicted with lightweight memory management across a range of hypervisors (e.g., ESX, KVM) across architectures (e.g., ARM, x86-64) and container-based technologies.
- We propose interpolation-based TLB speculation to leverage splintered but well-aligned system physical page allocation to improve performance by an average of 14% across our workloads. This represents almost all the address translation overheads in the virtualized systems we study.
- We investigate design trade-offs and splintering characteristics to explain the benefits of GLUE. We show the robustness of GLUE, which improves performance in every single workload considered.

2. BACKGROUND

Virtualization and TLB overheads: In hypervisor-based virtualized systems with two-dimensional page table support, guests maintain page tables mapping guest virtual pages (GVPs) to guest physical pages (GPPs), which are then converted by the hypervisor to system physical pages (SPPs) via a nested or extended page table [11]. TLBs cache frequently used GVP to SPP mappings; on TLB misses, the hardware page table walker performs a two-dimensional traversal of the page tables to identify the SPP. In x86-64 systems, because both guest and hypervisor use four-level radix-tree page tables, accessing each level of the guest's page table requires a corresponding traversal of the nested page table. Therefore, while native page table walks require four memory references, two-dimensional page table walks require twenty-four [11], significantly degrading system performance. Beyond hypervisors, container-based technologies also suffer from address translation overheads

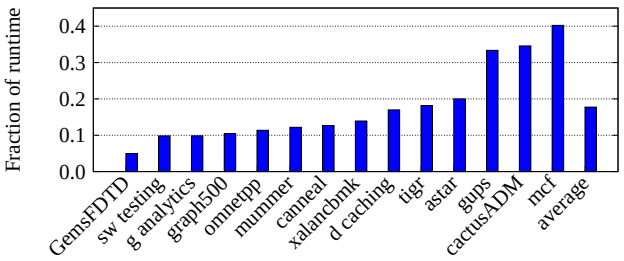

Figure 1: Percent of execution time for address translation, for applications on a Linux VM on VMware's ESX server, running on an x86-64 architecture. Overheads are 18% on average despite the fact that the OS uses both 4KB and 2MB pages.

(even though they use standard one-dimensional page table walks), primarily because TLB reach is unable to match main memory capacities.

Our work focuses on hypervisor-based virtualization as it presents the greater challenge on address translation. However, we have also characterized page splintering on containers and present these results.

Large pages and address translation: To counter increased TLB overheads in virtual servers, virtualization vendors encourage using large pages aggressively [15]. OSes construct large pages by allocating a sufficient number of baseline contiguous virtual page frames to contiguous physical page frames, aligned at boundaries determined by the size of the large page [4]. For example, x86-64 2MB large pages require 512 contiguous 4KB baseline pages, aligned at 2MB boundaries. Large pages replace multiple baseline TLB entries with a single large page entry (increasing capacity), and reduce the number of levels in the page table (reducing miss penalty).

In hypervisor-based virtualization, these benefits are magnified as they are applied to two page tables. While a large page reduces the number of page table walk memory references from four to three in native cases, the reduction is from twenty-four to fifteen for virtual machines. However, because TLBs cache guest virtual to system physical translations directly, a "true" large page is one that is large in both the guest and the hypervisor page table.

3. MOTIVATION AND OUR APPROACH

Real-system virtualization overheads: We begin by profiling address translation overheads on hypervisor-based virtualization technologies. Figure 1 quantifies address translation overheads (normalized to total runtime) when running a Ubuntu 12.04 server (3.8 kernel) as the guest operating system and VMware's ESX 5.5 as the hypervisor on an Intel Sandybridge architecture. Although omitted here for space reasons, we have also assessed these overheads running KVM on an x86-64 system, KVM on an ARM Cortex^{TM}A15 system, and we observed the same trends. We use SPECcpu®, PARSEC [14], and CloudSuite [17] workloads and all measurements use on-chip performance counters.

The data show that two-dimensional page table walks degrade performance, consuming almost 20% of runtime on average. Overheads vary, with `graph analytics` from CloudSuite and `graph500` suffering 10% over-

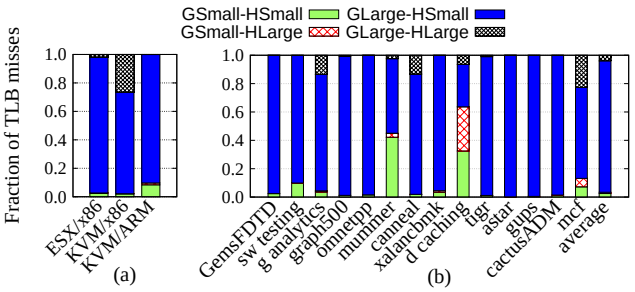

Figure 2: Fraction of TLB misses serviced from SPPs backed by a small page in guest and hypervisor (GSmall-HSmall), small in guest and large in hypervisor (GSmall-HLarge), large in guest and small in hypervisor (GLarge-HSmall), and large in both (GLarge-HLarge).

heads while `mcf` and `cactusADM` suffer well over 30% overheads. Our results corroborate past work [15] that identified TLB overheads as a significant performance bottleneck in virtualized servers.

The prevalence of page splintering: Figure 2(a) quantifies the prevalence of splintering across different hypervisors and architectures, showing the generality of this problem. We profile splintering for ESX and KVM on x86-64 architectures, and KVM on ARM architectures. In all cases, we run four VMs, each with the workload shown. For each configuration's TLB misses, we plot the fraction eventually serviced from splintered large pages (large pages in the guest and small pages in the host), small pages in both dimensions, large pages in both dimensions, and small pages in the guest and large pages in the host (which is typically rare). Our results show that guest VMs construct and use large pages aggressively (on average, almost all TLB misses are to regions with large guest pages). However, the vast majority of these references are to guest large pages that are splintered (GLarge-HSmall), regardless of the specific hypervisor or architecture used.

Figure 2(b) sheds light on the per-benchmark characteristics of page splintering. We find that guest OSes, when running workloads like `graph analytics`, `canneal`, and `mcf`, are able to more aggressively use large pages (usually because they allocate large data structures at once) but the hypervisor still chooses to splinter many pages for overall lightweight memory management. Going beyond hypervisor-based virtualization, we have also profiled page splintering in containers, using Linux containers with kernel same-page merging (KSM) [5] extensions to encourage high consolidation ratios. Similar to several real-world deployments [23] and we see that containers also suffer from large page splintering. This occurs because KSM shares pages among containers to consolidate as many workloads as possible with over 85% of TLB misses to splintered regions. The tension between large page address translation benefits and fine-grained memory management is regrettable because modern OSes work hard to create large pages.

Our high-level approach with GLUE: We illustrate GLUE's operation using hypervisor-based virtualization though it is also applicable to containers. We observe that page splintering usually splits large page

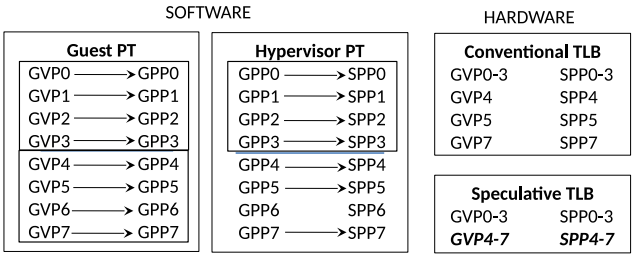

Figure 3: Guest large page GVP4-7 is splintered, but SPPs are conducive to interpolation. A speculative TLB maps the page table in two entries (using a speculative entry for GVP4-7) instead of four entries (like a conventional TLB).

sized regions into smaller pages for finer-grained management and monitoring, *without relocating most small pages*. Therefore, the contiguous memory alignment required to generate large pages likely remains; GVPs and SPPs are aligned within 2MB memory regions corresponding to the alignment they would have had in an unsplintered page. GLUE exploits these cases where GVPs and SPPs share their 9 least significant bits (for 4KB baseline pages and 2MB large pages). We have profiled page tables in many real-world deployments and in every case, we have found that over 82% (average of 97%) of the 4KB pages have GVPs and SPPs sharing their bottom 9 bits. These numbers foreshadow the potential for careful interpolation-based TLB speculation.

Figure 3 shows GLUE's operation. For illustrative purposes we assume that four contiguous PTEs make a large page, hence the guest page table can combine PTEs for GVP 0-3 into a single large page (the same for PTEs for GVP 4-7). The hypervisor does indeed back GVP 0-3 with its own large page (corresponding to the PTEs for GPP 0-3). Unfortunately, it splinters the guest large page for GVP 4-7 because GPP 6 (and SPP 6) are unallocated. A conventional TLB requires one large page entry and three baseline entries to fully cover this splintered large page. GLUE, on the other hand, observes that the SPPs corresponding to GVP 4-7 are still aligned in the way they would be had they actually been allocated a large page. Consequently, GLUE requires just two TLB entries to cover the page table, with one entry devoted to a speculative 2MB region (italicized and in bold). On requests to GVP 4-7, this entry can be used to interpolate the desired SPP; thus, speculative TLBs achieve higher capacity.

Overall, GLUE achieves performance benefits by leveraging *one-dimensional* large pages in the guest to approach the benefits of true, unsplintered, two-dimensional large pages. Speculation removes two-dimensional page table walks from a program's critical path, boosting performance. All enhancements are hardware-only so that any software and hypervisor may exploit them transparently, making it robust across the range of real-world splintering scenarios.

GLUE's relationship with prior work: GLUE's mechanisms are partly inspired by SpecTLB [9], which speculatively interpolates physical pages for operating systems using reservation-based large pages. Our work differs in a few key ways. First, the SpecTLB work

discusses the possibility of using TLB speculation to improve TLB miss latencies when guests and hosts use large pages, and to let hypervisors use very large (e.g., 1GB) pages while allowing guests to trap to special regions of memory for I/O. We are the first, however, to identify page splintering as a serious problem – and the first to propose and implement interpolation-based TLB speculation hardware mitigate page splintering. Second, we do not require separate, dedicated hardware structures for TLB speculation, unlike SpecTLB (although we propose some small modifications to existing structures). GLUE distinguishes baseline PTE entries, regular large page PTE entries, and speculative large page PTE entries. Third, we go beyond SpecTLB to propose enhancements to basic TLB speculation that mitigate performance degradation from incorrect speculation (e.g., pipeline flush and refetch) via intelligent PTE prefetching, which reduce the need to perform page table walks to verify a subset of the speculations. In addition, GLUE is also related to recent work on gap-tolerant sequential mappings (GTSM) which promotes superpages in non-contiguous memory [16]. Though there are some similarities with our work, GTSM is applicable to each dimension separately, unlike GLUE. Furthermore, GTSM requires complex software changes for new page table structures; whereas GLUE supports existing software. Further, GTSM's new page table format is unsuitable for one of the most important sources of page splintering – page sharing: Shared pages can be located anywhere in system memory, but GTSM only handles holes within 4MB memory regions.

4. SOURCES OF PAGE SPLINTERING

Despite their ability to lower address translation overheads, large pages impede other lightweight memory management operations. We discuss some of these reasons below and their influence on page splintering.

Page sharing: Memory deduplication or page sharing, an important source of page splintering, is implemented in commercial hypervisors (e.g., ESX, Xen, KVM) and is used with containers (LXC, Docker) to consolidate as many virtual machines as possible on the same physical resources [7, 20, 43]. Memory deduplication requires software to scan physical memory to identify memory pages with the same content and eliminate redundant copies. While effective at increasing consolidation ratios, our experiments and past work has found that page sharing leads to aggressive page splintering for two reasons [20]. First, baseline pages are much more likely to have equivalent content than large pages. We ran experiments to compare deduplication opportunities when considering baseline 4KB versus large 2MB pages. We found that using small 4KB pages allowed us to deduplicate 4-10× more physical memory than when using large pages, corroborating past results [20]. Second, the overheads of performing a word-by-word comparison for a 2MB page is much higher than that for a smaller, 4KB page [7]. Hence, when consolidation is targeted, large pages are rapidly splintered.

Hypervisors like ESX and KVM use page sharing aggressively both between and within VMs. This is particularly useful in real-world cloud deployments like Amazon's EC2, Oracle Cloud, and IBM Softlayer, which provide technical support for a limited number of OSes (e.g., Linux, Windows, and Solaris) commonly running VMs from one "template" [34]. In these environments, the workloads have great scope for memory deduplication. In fact, recent industry research has noted this trend and is advocating proactive and aggressive splintering of large pages in anticipation of the need for page sharing [20]. While effective at boosting consolidation, address translation performance is sacrificed.

Non-uniform memory: Past work has shown that large pages may fail to deliver benefits, and can actually degrade performance, on today's multi-socket, non-uniform memory access systems (NUMA) [19]. Because NUMA memory is spread across several physical nodes, large pages may contribute to imbalance in the distribution of memory controller requests, reduce locality of accesses, and increase memory latencies. The OS may therefore splinter large pages, with 4KB chunks migrated among the memory nodes to increase locality [19]. This problem will become even more prevalent with the impending adoption of even more complex, non-uniform heterogeneous memory architectures that balance the access latency, bandwidth, and power needs of heterogeneous processing elements consisting of CPUs, GPUs, and other accelerators [29]. Recent advances in technologies like phase-change, ferroelectric, magnetic, and memristor based RAM, allied with die stacking [6, 25, 44] suggest that intelligent page placement and movement among multiple memories will determine system performance and energy; page splintering will likely increase in these scenarios considerably.

Working set sampling: Hypervisors typically require some mechanisms to estimate the working set sizes of guest OSs. For example, VMware's ESX uses a statistical sampling approach to estimate virtual machine working set size without guest involvement [41]. For each sampling period, the hypervisor intentionally invalidates several randomly selected guest physical pages and monitors guest accesses to them. After a sampling period (usually a few minutes), the fraction of invalidated pages re-accessed by the guest is checked. This fraction is used to infer a VM's working set size. If a randomly chosen 4KB region falls within a large 2MB page, the hypervisor splinters the large page.

Initially, one may consider "repairing" working set estimation in software. This, however, is difficult; for example, one might use dirty and access bits in both guest and hypervisor page tables to detect page accesses instead of invalidating whole translations. Unfortunately, these bits are not supported by, for example, ARM and any Intel chips older than Haswell. In particular, architectures like ARM that implement relaxed memory consistency models struggle to accommodate page table access and dirty bits, which require sequentially consistent reads and writes for correct operation [32]. In addition, our conversations with hypervisor vendors like VMware suggest that they quite hesitant to implement software modules that can only be used on specific architectures supporting these bits.

Live VM migration: This refers to the process of moving a running virtual machine between different physical machines without disconnecting the client or application. Hypervisors typically splinter all large pages into baseline pages in preparation for live migration to identify pages being written to at 4KB granularity. Once memory state has been shifted to the destination, the splintered pages are typically reconstituted into large pages. However, practically, splintering may remain at the destination node, especially if unfragmented free physical memory space to accommodate large pages is scarce there.

Limited support for large pages: Large pages require hardware and software support. In practice, many systems lack this support in some way. For example, hypervisors may splinter large pages because of limited-capacity, large-page TLBs. Specifically, if an application's working set is scattered over a wide address space range, large page TLB thrashing can occur [10,39]. System administrators may therefore disable the hypervisor's ability to back guest large pages [39].

In general, while large pages mitigate address translation overheads, they preclude many memory management techniques for large-scale software systems. Fundamental to this tradeoff is the fact that a large page essentially provides a coarse granularity of memory management and monitoring; while it reduces metadata in the form of the number of translation entries needed, it also greatly reduces the effectiveness of operations like page sharing and memory monitoring.

5. GLUE MICROARCHITECTURE

This section details GLUE hardware. We describe our TLB organization,[1] and how speculation can be overlaid on it. We then discuss hardware tradeoffs.

5.1 TLB Organization

We assume a processor organization that includes per-core two-level TLB hierarchies, as is typical in modern processors [10, 12, 13]. On a memory reference, two L1 TLBs are looked up in parallel, one devoted to 4KB PTEs and another to 2MB PTEs. L1 misses[2] prompt a lookup in the L2 TLB. GLUE detects guest large pages splintered by the hypervisor. Ordinarily, each such page's 512 4KB PTEs are placed in the 4KB L1 and L2 TLBs. GLUE, however, creates a *speculative 2MB entry for the large page in one dimension*, with two approaches:

L1-only speculation: Here, speculative 2MB entries are placed only in the 2MB L1 TLB, permitting speculation only at the first-level of the TLB hierarchy. This is a minimally-intrusive design as the 4KB L1 and L2 TLBs are left untouched.

L1-L2 speculation: Though L1-only speculation is effective, it can place a heavy burden on the 2MB L1 TLB, which must now cache PTEs for not only two-dimensional, unsplintered large pages, but also for speculative one-dimensional large pages (which, our experiments reveal, there are many of). Therefore, we also study the benefits of placing speculative 2MB entries in *both* the 2MB L1 TLB and the L2 TLB.

In order to cache speculative 2MB entries, the L2 TLB must support multiple page sizes concurrently, but this is not a problem as modern processors already have this [21]. In general, there are many ways to accommodate concurrent page sizes in TLBs, such as skew-associativity [27,33,35–37] and hash-rehashing (column-associativity) [2]. We have evaluated GLUE on L2 TLBs with both schemes; because the performance numbers are largely unchanged, we present results from skew-associative L2 TLBs in this paper. We do note, however, that GLUE merely leverages already existing techniques to support multiple page sizes concurrently in the TLB, and is not particularly reliant on skewing or hash-rehash.

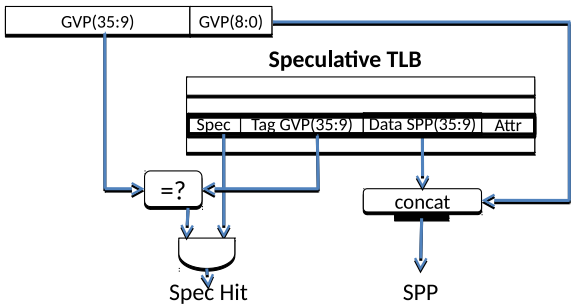

Figure 4: Lookup operation on a speculated TLB entry. A tag match is performed on the bits corresponding to its 2MB frame. On a hit, the 2MB frame in system physical memory is concatenated with the 4KB offset within it.

5.2 Speculative TLB Entries

Figure 4 shows a speculated 2MB entry in the L1 2MB TLB. Its structure is identical to a standard 2MB entry, with only a `Spec` bit added to distinguish speculated 2MB entries from standard 2MB entries (necessary to ensure that the L2 TLB and page table walker are probed to verify speculation correctness). This bit represents a minor overhead over the ∼60 bits used per 2MB TLB entry. L2 TLB entries are also minimally changed to support speculation. Once again, a `Spec` bit is required to identify speculated 2MB entries.

5.3 TLB Operations

Lookups: Figure 4 shows how GLUE performs a speculative lookup for 2MB entries. The guest virtual address is split into a page number and a page offset (not shown). The GVP is further split into a field for 2MB frames (bits 35:9) and the 4KB offset within the 2MB frames (bits 8:0). A lookup compares the 2MB frame bits with the TLB entry's tag bits. A *speculative hit* occurs when there is a match and the `Spec` bit is set. The matching TLB entry maintains the system physical page of the 2MB speculated frame (Data SPP(35:9) in the diagram). This value is interpolated by concatenating the TLB entry data field with the GVP's within-2MB frame offset (GVP(8:0)). The full system physical

[1] Our proposal does not depend on this specific TLB organization, but we describe it to provide a concrete example to explain our technique.
[2] As this paper is about TLBs, for brevity we use the term "L1" to refer to the L1 TLB, and *not* the IL1 or DL1 cache (and similarly for "L2").

address is calculated, as usual, by concatenating the guest virtual page offset bits (bits 11:0) with the speculative SPP. This value is then returned to the CPU, which can continue execution while the speculated value is verified. Speculative lookups therefore require minimal additional logic, with only the `Spec` bit check and concatenation operation to generate the SPP.

Fills: GLUE fills speculated 2MB entries into the L1 and L2 TLBs after a page table walk. Suppose a GVP request misses in both L1 and L2 TLBs. The hardware page table walker then traverses both guest and hypervisor page tables, and identifies 4KB GVPs that map to a large guest page but small hypervisor page. These PTEs can be identified from already-existing information on page size in the page tables. For these GVPs, the speculated 2MB frame is calculated by dropping the bottom 9 bits from the corresponding SPP. Then, the PTE for the requested 4KB GVP is placed into the 4KB TLB (as usual), while the speculated 2MB entry is also placed into the 2MB L1 TLB and L2 TLB. Therefore, identifying one-dimensional large pages requires *no additional hardware* beyond standard page table walks.

5.4 Speculation Details

We now study various aspects of GLUE, focusing on L1-L2 speculation as it is a superset of L1-only speculation. Figure 5 details the control and data flow through the TLBs to support GLUE. Figure 6 illustrates the timing of events corresponding to different hit/miss and speculation scenarios.

Correct L1 speculation, verified in the L2 TLB: Figure 6(a) illustrates the case where GLUE speculates correctly from the L1 TLB and completes verification from the L2 TLB. The CPU first checks the two L1 TLBs (4KB and 2MB) ①; it misses in the 4KB TLB, finds a speculative entry in the 2MB TLB that results in a speculation hit ②. The hit signal and corresponding speculated SPP are sent to the CPU, which can continue execution while the speculated SPP is verified in parallel. Verification proceeds by checking the L2 TLB ③, which produces a hit on either a matching 4KB entry or a clustered bitmap corresponding to the speculative 2MB entry ④ (speculation confirmed). The 4KB entry is then installed into the L1 4KB TLB, but this occurs off of the critical path.

Incorrect L1 speculation, verified in the L2 TLB: This case starts out the same as in the previous scenario, but when we hit in the L2 TLB, we discover that the actual SPP is not the same as the interpolated SPP ⑤. This triggers a pipeline flush and refetch as any consumers of the load may have started executing with an incorrectly loaded value. Figure 6(b) shows the corresponding timing of events.

Correct L1 speculation, verified by a page table walk: This case is also similar to the first scenario, except that when the L2 TLB lookup is performed, no matching entry is found. A page table walk retrieves the correct translation ⑥, which is found to be the same as the speculated SPP. As shown in Figure 6(c), the page table walk to verify the SPP occurs in parallel with the processor pipeline's continued execution using the speculative SPP. In this case, the speculation is correct and so the processor was able to run ahead.

Incorrect L1 speculation, verified by a page table walk: This case is similar to the immediate previous one, except that at the conclusion of the page table walk ⑥, the correct SPP is found to differ from the interpolated SPP. The processor initiates a pipeline flush and refetch. For this case, we also insert the 4KB translation into both L1 and L2 TLBs. The L1 insertion attempts to avoid speculation, and the L2 insertion attempts to ensure a faster verification process in the event that we speculate again from the L1 2MB TLB.

L2 speculation: Figure 6(e) and (f) show the cases where a speculative entry is found in the L2TLB. These cases parallel the L1 speculation scenarios with the only difference that the lookup misses in all L1 TLBs and the verification (whether correct or not) is performed by a page table walk.

5.5 Mitigating Verification Costs

We now study the tradeoff between capacity and verification costs of speculative large page translations, focusing on two questions.

What should we do with requested baseline PTEs if we were able to correctly speculate on them? Consider the case where we use a 2MB speculative entry to correctly ascertain 4KB PTEs. After verification, we can either place the 4KB PTE in the TLBs, or not insert it into the TLB hierarchy at all. Both approaches have merit; insertion into the TLBs reduces the cost of verification. Because these PTEs are likely to be used in the near future, placing them in the L1 TLB obviates the need for verification-induced L2 TLB lookups, and possibly full-blown page table walks (on L2 TLB misses). On the other hand, insertion into the TLBs lowers effective capacity; true large page entries replace multiple base page translations (e.g., a 2MB x86-64 PTE covers 512 4KB PTEs).

We have compared inserting non-speculative 4KB PTEs into the L1 TLB (reducing verification of future accesses to the same entry but penalizing limited L1 TLB capacity), and into the L2 TLB (promoting speculation with fast verification from the L2 TLB while saving L1 TLB capacity). We also consider non-desirable extremes where we do not insert the non-speculative 4KB entry into the TLBs at all (maximizing capacity but severely exacerbating verification costs), and where we add the non-speculative 4KB PTE in both TLBs (minimizing verification but hurting capacity).

In general, we have found (Section 7) that insertion into only the L1 TLB performs best because only tens of 4KB pages within 2MB speculated regions are typically used in temporal proximity. This means that capacity requirements may be relaxed in favor of minimizing the time taken to verify that speculation was correct. Given verification energy requirements and potential performance loss (from greater L2 TLB port contention and cache lookups for page table walks), we believe this is a suitable compromise.

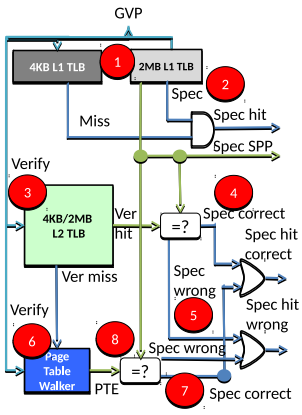

Figure 5: The mechanics of TLB speculation. We show the case when we speculate from the 2MB L1 TLB.

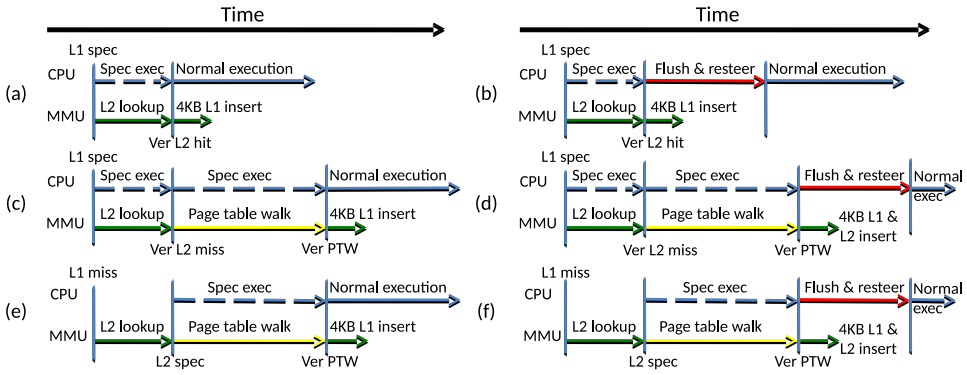

Figure 6: Timelines for (a) speculating from the 2MB L1 TLB correctly, and verifying this in the L2 TLB; (b) mis-speculating from the 2MB L1 TLB, and verifying this in the L2 TLB; (c) speculating from the 2MB L1 TLB correctly, and verifying with a page table walk; (d) mis-speculating from the 2MB L1 TLB, and verifying with a page table walk; (e) speculating from the L2 TLB correctly, and verifying with a page table walk; and (f) mis-speculating from the L2 TLB, and verifying with a page table walk.

Can we use the extra bits in a speculative large page entry in the L2 TLB to reduce verification costs? Because translations for large pages use fewer bits to represent virtual and physical page numbers, speculative large-page entries maintain many unused bits in a monolithic L2 TLB provisioned to concurrently handle multiple page sizes. In our example, the 2MB speculative large page entries have 18 unused bits. We investigate mechanisms to repurpose these 18 bits to reduce verification costs. Figure 7 shows how we use these bits to maintain bitmaps that tell us if 4KB PTEs within a 2MB speculative large page have the contiguity and alignment to permit correct speculations. For example, Figure 7(a) shows the CPU making a request for the translation for VPN 4. A speculative large page entry containing this VPN is discovered in the 2MB L1 TLB, and an interpolated, speculative PPN is sent to the CPU ①. While the CPU continues its operation, the speculative PPN must be verified. The L2 TLB is therefore probed, and the corresponding 2MB speculative translation is located ②. This translation is similar to its counterpart in the L1 TLB but also uses its spare bits to record information about a cluster of 4KB PTEs surrounding the most-recently accessed translation in this speculative 2MB region. We use the unused bits to maintain a cluster number, and a bitmap indicating which PTEs in this cluster are aligned and contiguous (in our page table, all translations aside from the one for VPN 7 are so). Initially this bitmap is empty, therefore it is not able to tell us whether our speculation is correct and we have to walk the page table to verify this ③. We then load the most recently-accessed cluster's information (cluster 0, which VPN 4 falls in) ④. Figure 7(b) shows that loading this bitmap in the L2 TLB allows us, in the future when the CPU speculates a PPN from the L1 2MB TLB for VPN 6 ⑤, to quickly verify that this is indeed a correct speculation without the need for an expensive page table walk ⑥. Using the same bitmap, we can also verify quickly if the speculation is incorrect when we speculate a PPN from the L1 2MB TLB for VPN 7.

Through our detailed experiments, we have found that using the 18 spare bits to store two clusters of

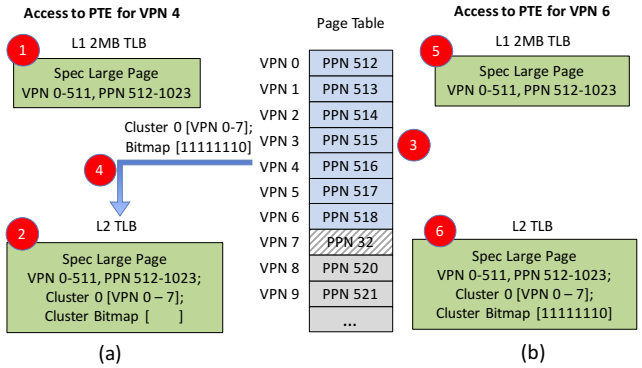

Figure 7: Storing clusters of bits (in otherwise wasted L2 TLB entry bits) to eliminate the need for verification-induced page table walks.

8 bits eliminates verification with only 2 additional bits per TLB entry.Furthermore, since typical 64-byte cache lines maintain eight 8-byte PTEs, we use simple combinational logic proposed in recent work [30,31] to set the bitmap after a page table walk *without any additional memory references for the walk.*

5.6 Mitigating Mis-speculation Overheads

Our real-system measurements showed that loads that miss in the TLBs typically then also miss in the cache hierarchy (90% go to the L3, and 55% to main memory). This is intuitive because PTEs that miss in the TLBs have not been accessed recently; therefore the memory pages they point to are also likely cold. At the point of misspeculation detection, the SPP is now known and a prefetch of the address into the DL1 can be started in parallel with the pipeline flush process. For our workloads, it turns out that the TLB speculation rate is sufficiently high such that mitigating the rare flushing event has only a minor impact on overall performance, but this mechanism can provide a more robust solution in the face of less speculation-friendly workloads.

6. EXPERIMENTAL METHODOLOGY

To evaluate functional, behavioral, and performance effects, we examine systems running multiple real OSs

and hypervisors. Our proposal also includes microarchitectural modifications that cannot be evaluated on real systems, while current cycle-level simulators do not support multiple VMs and OSes in full-system simulation modes. For these reasons, like most recent work on TLBs [9,10,13,18,30,31], we use a combination of techniques including tracing and performance counter measurements on real machines, functional cache-hierarchy and TLB simulators, and analytical modeling to estimate the overall impact on program execution time.

6.1 Workloads

We set up our virtualization environment on a host machine with 8 CPUs and 24GB RAM. We deploy 8 VMs, each has 3GB of RAM for BioBench and SPECcpu workloads, and 4VMs, each with 4GB of RAM for Cloudsuite workloads. The host uses VMware ESXi server to manage VMs. All VMs have Ubuntu 12.04 server, and large pages are enabled using Transparent Hugepage Support (THS) [4]. In addition, to showcase the generality of our observations across hypervisors and architectures, we evaluate KVM on the same hardware and KVM on an ARM system with four Cortex A15 cores. Finally, we use perfmon2 to read performance counter values in the VMs. We use a wide set of benchmarks, from SPECcpu 2006, BioBench [3], and CloudSuite [17] that have non-negligible TLB miss overheads. We present results on workloads sensitive to TLB activity. For our container-based studies, we use linux containers (LXC) with KSM [5].

6.2 Trace Collection

We use Pin [24] to collect guest memory traces for our workloads. The original pintool only provides virtual addresses, hence we extend the Linux pagemap to include physical addresses and intermediate page table entries (PTE) to be read by our pintool. For each workload, we select a PinPoint region of one billion instructions [28], and we validate the MPKI of the trace with performance counter measurements to ensure that the sampled region is representative of the benchmark.

We use VMware VProbes scripts [40] to collect hypervisor memory traces, which contain guest and system physical addresses. We rely on guest physical addresses, which are seen in both guest and hypervisor traces to get a complete trace of guest virtual, guest physical, and system physical addresses for our simulator. We also extend the tracing utility to VMs on KVM hypervisor to get similar information.

6.3 Functional simulator

To determine the hit-rate impact of the different TLB structures, we make use of a functional simulator that models multi-level TLBs, the hardware page-table walker, and the conventional cache hierarchy. The TLBs include a 64-entry, 4-way DTLB for 4KB pages; a 32-entry, fully-associative DTLB for 2MB pages; and a 512-entry, 4-way level-two TLB (L2TLB) with concurrent support for 4KB and 2MB pages, similar to Intel's Haswell cores. Our L2 TLB uses a skewed-associative organization [36] for multiple page size support (we have also modeled hash-rehash approaches, which negligibly changes performance benefits). The modeled TLB hierarchy also includes page walk caches that can accelerate the TLB miss latency by caching intermediate entries of a nested page table walk [8, 13]. The simulator has a three-level cache hierarchy (32KB, 8-way DL1; 256KB, 8-way L2; 8MB, 16-way L3 with stride prefetcher), which the modeled hardware page table walker uses on TLB misses.

6.4 Analytical Performance Model

For each application, we use the real-system performance counter measurements (on full-program executions) to determine the total number of cycles `CPU_CYCLES` (execution time), the total number of page-walk cycles `PWC` (translation overhead, not including TLB access latencies), the number of DTLB misses that resulted in L2TLB hits, and the total number of L2TLB misses (which then require page table walks). In addition, we also make use of several fixed hardware parameters including the penalty to flush and refill the processor pipeline (20 cycles, taken to be about the same as a branch misprediction penalty [1]), the DTLB hit latency (1 cycle), and the L2TLB hit latency (7 cycles).

The analytical performance model is conceptually simple, although it contains many terms to capture all of the different hit/miss and correct/wrong speculation scenarios covered earlier in Figure 6. In the baseline case without GLUE, the program execution time is simply `CPU_CYCLES`. From the performance counters, we can determine the total number of cycles spent on address translation overheads `ATO` (e.g., page table walks), and therefore `CPU_CYCLES - ATO` gives us the number of cycles that the processor is doing "real" execution `BASE_CYCLES` (i.e., everything else but address translations). In other words, this would be the execution time if virtual memory was completely free. From here, our analytical model effectively consists of:

$$\texttt{Execution\_Time} = \texttt{BASE\_CYCLES} + \sum_i \texttt{ATO}_i \qquad (1)$$

where each $\texttt{ATO}_i$ is the number of cycles required for address translation for each of GLUE's hit/miss and speculation/misspeculation scenarios.

For example, consider when we have a DTLB miss but we find a speculative entry in the 2MB TLB *and* the speculative translation turns out to be correct, then the number of cycles for address translation would simply be the latency of the L1 TLB (both 4KB and 2MB TLBs have the same latency in our model), as we assume that the verification of the speculation can occur off of the critical path of execution. Our functional simulation determines how often this happens in the simulated one-billion instruction trace, we linearly extrapolate this to the full-program execution to estimate the total number of such events[3], and multiply this by the L1 TLB latency to determine the execution cycles due to this scenario.

For a slightly more interesting example, consider the

[3] We are confident in our extrapolation methodology as we have validated the performance projections from our 1B instruction traces against performance counter measurements taken from the corresponding *full-program* executions.

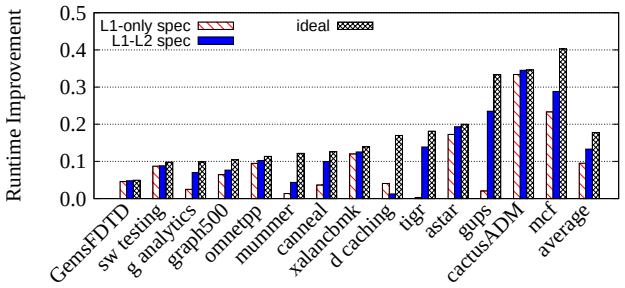

Figure 8: Performance benefits of L1-only, L1-L2 speculation, compared to the ideal case. Performance is normalized to the baseline single-VM.

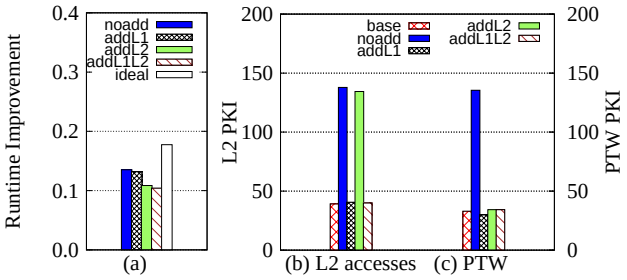

Figure 9: Average (a) performance improvements when inserting the non-speculative 4KB PTE, after correct speculation, in neither TLB (noAdd), the L1 TLB (addL1), the L2 TLB (addL2), or both (addL1L2), compared with the ideal improvement; (b) number of L2 TLB accesses per kilo-instruction (APKI) including verification compared to a baseline with speculation; and (c) number of page table walks per kilo-instruction.

case shown in Figure 6(d) where we miss in the L1 4KB TLB, find a speculative entry in the L1 2MB TLB, the speculation turns out to be wrong, but it required a full page-table walk to determine this (i.e., the PTE was not in the L2TLB). The number of cycles for such an event is:

$$\texttt{L2TLB\_LAT} + \texttt{PW\_LAT} + \max(\texttt{DATA\_LAT}, \texttt{BMP}) \quad (2)$$

which corresponds to the L2TLB access latency (need to perform a lookup even though it misses), the page-walk latency `PW_LAT` (we use the average as determined by performance counters), and then the longer of either the data cache lookup `DATA_LAT` or the branch misprediction penalty `BMP`. Assuming we use the prefetch optimization described in Section 5.6, when we detect the misspeculation, we can then concurrently flush the pipeline and start the prefetch of the load into the DL1. Because these events occur in parallel, we take the maximum of these terms. Due to space constraints, we omit explanations for the remaining `ATO_`i equations, but they all follow a similar form that reflects what has already been described in Figure 6. It should be noted that such a methodology based on analytically adjusting real-machine measurements has been used in other recent virtual memory research studies [10].

7. EXPERIMENTAL RESULTS

7.1 GLUE Performance Results: Single VM

We first consider the performance improvements of L1-only and L1-L2 speculation, also showing the importance of careful verification control.

GLUE performance: Figure 8 quantifies the benefits of GLUE (all results are normalized to the runtime of the application on a single VM) for L1 and L1-L2 speculation, showing it eliminates the vast majority of TLB overheads in virtualized systems with splintering. On average, runtime is improved by 14%, just 4% away from the performance of an ideal system with no address translation overheads (i.e., there are never any L1 TLB misses). Most benchmarks are actually significantly closer to the ideal case with only `mummer`, `data caching`, `gups`, and `mcf` showing a difference. For `mummer` and `data caching`, this occurs because they are the only benchmarks where the guest generates fewer large pages (see Figure 3); nevertheless, performance benefits are still 5%. For `gups`, and `mcf`, the difference occurs because these benchmarks require more 2MB entries (speculative or otherwise) than the entire TLB hierarchy has available; nevertheless, we still achieve 24%, and 30% performance gains, respectively.

Interestingly, Figure 8 also shows that L1-only speculation is highly-effective, achieving 10% performance benefit. In fact, only `mummer`, `tigr`, `gups`, and `graph analytics` see significantly more performance from speculating with both L1 and L2 TLBs.

Mitigating verification costs: To balance the capacity benefits of speculative 2MB entries against the overheads of verification, we insert the non-speculative 4KB PTE corresponding to a correct speculation into the 4KB L1 TLB. Figure 9 evaluates this design decision versus a scheme that inserts the non-speculative 4KB PTE into the L2 TLB instead, into both, or into neither. We show the performance implications of these decisions and the number of additional L2 TLB lookups and page table walks they initiate to verify speculations (per kilo-instruction). All results assume L1-L2 speculation; we show average results because the trends are the same across benchmarks.

Figure 9(a) shows that in general, `noAdd` performs the best because a single speculative 2MB entry is used in the entire TLB hierarchy for information about any constituent 4KB SPP. However, inserting non-speculative 4KB PTEs into the L1 TLB (`addL1`), the L2 TLB (`addL2`), or even both (`addL1L2`) performs almost as well (within 2%). Figures 9(b)-(c) shows, however, that these schemes have vastly different verification costs, by comparing the additional page walks per kilo-instruction and L2 TLB accesses per kilo-instruction they initiate. Not only does `noAdd` roughly triple the page table walks and L2 TLB accesses, even `addL2` only marginally improves L2 TLB access count. Therefore, we use `addL1` because its verification costs are comparable to the baseline case without sacrificing performance.

Figure 10a quantifies the impact of the L2 entry cluster bitmaps. We show the fraction of original page table walks eliminated; on average, 27% of the costly page table walks are eliminated, with absolutely no loss in performance and only 2 additional bits per TLB entry. Some workloads like `cactusADM` are almost entirely

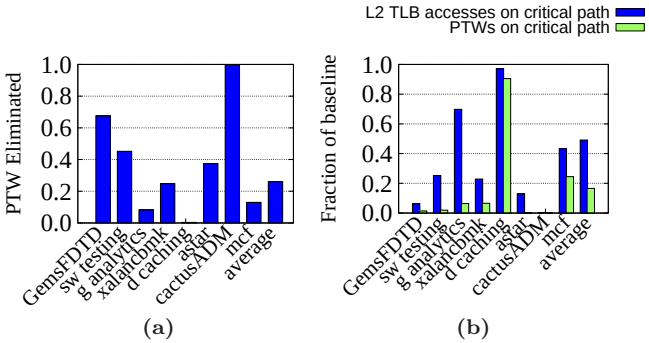

Figure 10: (a) Fraction of page table walks eliminated using clustered bitmaps in speculative L2 TLB entries; and (b) fraction of the baseline L2 TLB accesses and page table walks remaining on the critical path of execution with TLB speculation.

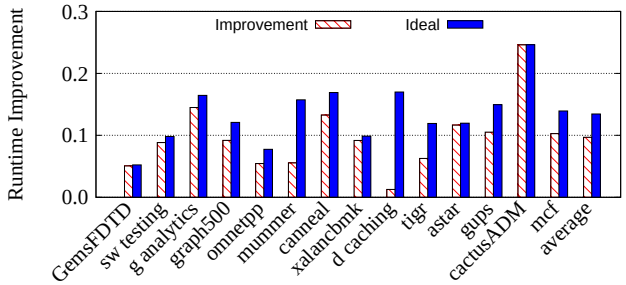

Figure 11: Performance gains achieved by GLUE on a multi-VM configuration, compared against the ideal performance improvement where all address translation overheads are eliminated.

freed of page table walks, while others like `GemsFDTD` and `software testing` see 70% and 48% eliminated.

Analyzing TLB miss rates: Figure 10b profiles how many of the baseline VM's L2 TLB accesses (caused by L1 TLB misses) and page table walks (caused by L2 TLB misses) remain on the program's critical path. The others are removed from the execution critical path (because they are correctly speculated) and hence do not incur a performance penalty. Overall, Figure 10b shows that TLB speculation removes 45% of the L2 TLB lookups and 80% of the page tables walks from the critical path. Some workloads, like `cactusADM`, see almost no page table walks because the clustered bitmap in the L2 TLB completely eliminates verification-induced page table walks (see Figure 10a).

7.2 GLUE Performance Results: Multiple VMs

VMs with similar workloads: We have investigated the benefits of TLB speculation in scenarios with multiple virtual machines, which may change splintering rates because of inter-VM page sharing, etc. We have studied scenarios with 2, 3, 4, and 8 VMs, but because performance trends are very similar, we show 8-VM results for the SPECcpu and PARSEC workloads. CloudSuite applications, which have far greater memory needs, overcommit system memory with fewer VMs; we hence present 4-VM studies for them.

Figure 11 quantifies GLUE's benefits on the multi-VM setup (averaged across VMs, since we find negligible inter-VM variance), compared to an ideal scenario with no TLB misses. Multiple VMs stress TLBs even further due to greater contention. Fortunately, sufficient

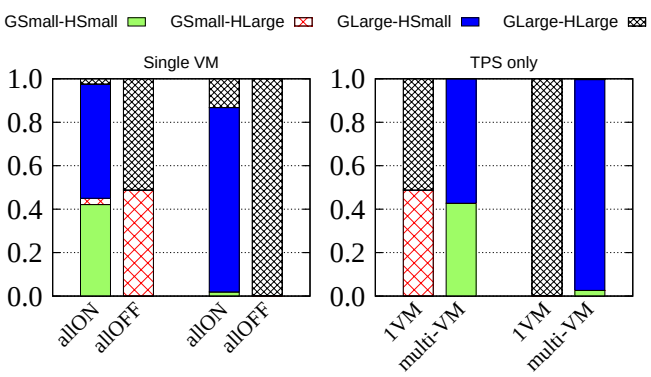

Figure 12: (a) Effect of page sharing and memory sampling turned on (allOn) in a single VM versus all off (allOff) on page splintering; and (b) Effect of inter-VM page sharing on page splintering in multi-VM settings.

page splintering and good alignment remains, letting GLUE (with L1-L2 speculation) eliminate 75% overheads on average. `GemsFDTD`, `astar`, `cactusADM`, and `software testing` see virtually no address translation overheads. Because real-world virtualization deployments commonly share a single physical node among multiple VMs, GLUE has high real-world utility.

VMs with different workloads: We also study the rarer case where one physical node runs multiple VMs with the same OS (stock Linux) but different workloads. We were surprised to observe that even with different workloads, there is significant inter-VM page sharing, leading to ample page splintering. For example, we found that at least 80% of all TLB misses were to pages that were large in the guest and small in the hypervisor when running VMs with `mcf`, `graph500`, and `gups`. Notable culprits were pages shared from the OS images across VMs and shared libraries. In addition, zero-value pages across VMs were also shared [5]. Fortunately, GLUE counters page sharing-induced splintering even when VMs run different workloads, greatly boosting performance.

7.3 Characterizing Page Splintering Sources

By default, ESX uses working set sampling and page sharing for overall performance [7]. Figure 12a shows how guests and the hypervisor allocate small and large pages, assuming a single VM running on the physical host. Results are shown for `canneal` (whose behavior almost exactly matches all the other benchmarks and is hence indicative of average behavior) and `mummer`, which illustrates more unique behavior. We compare the default setting against a case where sampling and sharing are turned off. Clearly, turning off sampling and page sharing recovers almost all of the opportunity lost by page splintering.

Figure 12b extends these observations when multiple VMs share a physical machine. Because multiple co-located VMs have many pages to share (from the kernel and application), assuming sampling is disabled, page sharing splinters most of the guest large pages.

7.4 Importance of GLUE in Future Systems

Our application of GLUE to hypervisor-based virtualization targets scenarios where guests can indeed cre-

ate large pages, which is purely a function of OS large page support. Modern operating systems have sophisticated and aggressive support for large pages [4, 31, 38], so guests are likely to continue generating large pages. Nevertheless, we now consider unusual scenarios which could impede guest large page creation.

One might initially consider that memory fragmentation on the guest might curtail large page use in some scenarios. To this, we make two observations. First, VMs are typically used on server and cloud settings to host an isolated, single service or logically related sets of services. It is highly unlikely that fragmentation from competing processes are an issue. Second, in the unusual case where this is an issue, many past studies on large-page support conclude that sophisticated already-existing memory defragmentation and compaction algorithms in OS kernels [4,31] drastically reduce system fragmentation. To test this, we ran our workloads in setups where we artificially fragmented system memory heavily and completely using the random access memhog process [31]. We found that even for workloads with the largest memory footprints (e.g., `mcf`, `graph500`, `data analytics`, `data caching`, and `software testing`), there were negligible changes in the number of guest large pages allocated, their splintering rates, and how well-aligned the ensuing splintered 4KB pages were. GLUE remains effective in every single, aggressively-fragmented setup that we investigated.

One might also consider the impact of memory ballooning on the guest's ability to generate large pages. Ballooning is a memory reclamation technique used when the hypervisor is running low on memory (possibly in response to the demands of concurrently-running VMs). When the balloon driver is invoked, it identifies 4KB page regions as candidates to relinquish to the hypervisor, and unallocates them. In effect, this fragments the guest's view of physical memory, hampering large page allocation, or breaking already-existing large pages. To study this, we have run several experiments on our setups. Since hypervisors like KVM and Xen expose ballooned pages to the memory defragmentation software run by the kernel, ballooning has no impact on guest large page generation [4].

7.5 Understanding GLUE's Limitations

GLUE activates TLB speculation only for memory regions where a large guest page is splintered by the hypervisor and identified by the page table walker as such. Therefore, GLUE is ineffective (though not harmful) when the guest is unable to generate large pages. However, TLB speculation can actually be harmful when the 4KB pages in a speculated large page region are not well-aligned; in these cases, frequent TLB mis-speculations introduce pipeline flushes and refetches, degrading performance. We have not encountered a single case where mis-speculations degrade performance in practice, but we detail how to handle this should it become an issue for other workloads we have not evaluated.

Section 5.4 explained that TLB misses are frequently followed by long-latency accesses to the lower-level caches or to main memory to retrieve the requested data. Because L3 caches and main memory typically require 40-200 cycles on modern systems [12, 30, 31], these latencies usually exceed (or are at least comparable) to the cost of flushing and steering the pipeline on a mis-speculation. Therefore, by initiating a cache prefetch for these data items as soon as a mis-speculation is detected, we can usually overlap mis-speculation penalties with useful work. Because all our real-system configurations enjoy accurate speculation, cache prefetching is not really necessary (on average, we gain roughly 1% more performance). We have calculated the minimum correct speculation rate required to ensure *no performance loss*; for every single benchmark evaluated, 48% speculation accuracy (a pessimistic scenario compared to the 90% accuracy we see in all our configurations) results in no performance degradation.

8. RELATED WORK

Beyond recent work on TLB speculation [9], the rising costs of address translation on big-memory systems have prompted researchers to perform many other studies on TLB design [8, 10, 13, 18, 31]. While some of these efforts have focused on mostly hardware efforts [8,13,30,31], others have shown the benefits of efficient OS-hardware co-design [10,16,22]. In particular, recent work on redundant memory mappings [22] has interesting implications on page splintering since it employs eager allocation - depending on workload configuration and hypervisor decision-making, splintering could occur at the granularity of ranges of (possibly large) pages. We will investigate the interplay between our splintering approaches and redundant mappings in future work.

9. CONCLUSION

This work observes the fundamental conflict between the address translation benefits of large pages versus the desire for finer-grained monitoring and agile memory management. We ask the question: is it possible to provide hardware support that enables us to ally the TLB reach benefits of large pages with memory management issues like lightweight memory monitoring, smoother page sharing, and seamless management with NUMA systems. Our proposed hardware uses interpolation-based TLB speculation to achieve this, boosting hypervisor-based virtualization and containers. Overall, while we observed that splintering is a problem and can cause significant performance problems, our proposed GLUE architecture can largely mitigate these issues, thereby making virtualized systems more attractive to deploy.

10. ACKNOWLEDGMENTS

We thank Jim Mattson for his help and valuable feedback. We thank Kathryn McKinley and Mark Hill for their insights and feedback in preparing the final version of the paper. Further, we thank the National Science Foundation, which partially supported this work through grants 1253700 and 1337147, and VMware for its support. 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.

11. REFERENCES

[1] "Software Optimization Guide for AMD Family 15h Processors," Advanced Micro Devices Inc, Tech. Rep., 2014.

[2] A. Agarwal and S. Pudar, "Column-Associative Caches: A Technique for Reducing the Miss Rate of Direct-Mapped Caches," *ISCA*, 1993.

[3] K. Albayraktaroglu, A. Jaleel, X. Wu, M. Franklin, B. Jacob, C.-W. Tseng, and D. Yeung, "BioBench: A Benchmark Suite of Bioinformatics Applications," in *Proc. of the Intl. Symp. on Performance Analysis of Systems and Software*, Austin, TX, March 2005, pp. 2–9.

[4] A. Arcangeli, "Transparent Hugepage Support," *KVM Forum*, 2010.

[5] A. Arcangeli, I. Eidus, and C. Wright, "Increasing Memory Density by Using KSM," *Ottawa Linux Symposium*, 2009.

[6] G. Atwood, "Current and Emerging Memory Technology Landscape," *Flash Memory Summit*, 2011.

[7] I. Banerjee, F. Guo, K. Tati, and R. Venkatasubramanian, "Memory Overcommittment in the ESX Server," *VMware Technical Journal*, 2013.

[8] T. Barr, A. Cox, and S. Rixner, "Translation Caching: Skip, Don't Walk (the Page Table)," *ISCA*, 2010.

[9] ——, "SpecTLB: A Mechanism for Speculative Address Translation," *ISCA*, 2011.

[10] A. Basu, J. Gandhi, J. Chang, M. Hill, and M. Swift, "Efficient Virtual Memory for Big Memory Servers," *ISCA*, 2013.

[11] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne, "Accelerating Two-Dimensional Page Walks for Virtualized Systems," *ASPLOS*, 2008.

[12] A. Bhattacharjee, D. Lustig, and M. Martonosi, "Shared Last-Level TLBs for Chip Multiprocessors," *HPCA*, 2011.

[13] A. Bhattacharjee, "Large-Reach Memory Management Unit Caches," *MICRO*, 2013.

[14] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Simplications," *IISWC*, 2008.

[15] J. Buell, D. Hecht, J. Heo, K. Saladi, and R. Taheri, "Methodology for Performance Analysis of VMware vSphere under Tier-1 Applications," *VMWare Technical Journal*, 2013.

[16] Y. Du, M. Zhou, B. Childers, D. Mosse, and R. Melhem, "Supporting Superpages in Non-Contiguous Physical Memory," *HPCA*, 2015.

[17] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, , and B. Falsafi, "Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware," *ASPLOS*, 2012.

[18] J. Gandhi, A. Basu, M. Hill, and M. Swift, "Efficient Memory Virtualization," *MICRO*, 2014.

[19] F. Gaud, B. Lepers, J. Decouchant, J. Funston, and A. Fedorova, "Large Pages May be Harmful on NUMA Systems," *USENIX ATC*, 2014.

[20] F. Guo, S. Kim, Y. Baskakov, and I. Banerjee, "Proactively Breaking Large Pages to Improve Memory Overcommitment Performance in VMware ESXi," *VEE*, 2015.

[21] D. Kanter, "Haswell Memory Hierarchy," *http://www.realworldtech.com/haswell-cpu/5/*, 2012.

[22] V. Karakostas, J. Gandhi, F. Ayar, A. Cristal, M. Hill, K. McKinley, M. Nemirovsky, M. Swift, and O. Unsal, "Redundant Memory Mappings for Fast Access to Large Memories," *ISCA*, 2015.

[23] B. Kero, "Running 512 Containers on a Laptop," *http://bke.ro/running-512-containers-on-a-laptop*, 2015.

[24] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*, Chicago, IL, June 2005, pp. 190–200.

[25] M. Meswani, S. Blagodurov, D. Roberts, J. Slice, M. Ignatowski, and G. Loh, "Heterogeneous Memory Architectures: A HW/SW Approach for Mixing Die-Stacked and Off-Package Memories," *HPCA*, 2015.

[26] J. Navarro, S. Iyer, P. Druschel, and A. Cox, "Practical, Transparent Operating System Support for Superpages," *OSDI*, 2002.

[27] M. Papadopoulou, X. Tong, A. Seznec, and A. Moshovos, "Prediction-Based Superpage-Friendly TLB Designs," *HPCA*, 2014.

[28] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi, "Pinpointing Representative Portions of Large Intel Itanium Programs with Dynamic Instrumentation," in *Proc. of the 37th Intl. Symp. on Microarchitecture*, Portland, OR, December 2004.

[29] S. Phadke and S. Narayanasamy, "MLP Aware Heterogeneous Memory System," *DATE*, 2011.

[30] B. Pham, A. Bhattacharjee, Y. Eckert, and G. Loh, "Increasing TLB Reach by Exploiting Clustering in Page Translations," *HPCA*, 2014.

[31] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee, "CoLT: Coalesced Large-Reach TLBs," *MICRO*, 2012.

[32] B. Romanescu, A. Lebeck, and D. Sorin, "Specifying and Dynamically Verifying Address Translation-Aware Memory Consistency," *ASPLOS*, 2010.

[33] D. Sanchez and C. Kozyrakis, "The ZCache: Decoupling Ways and Associativity," *MICRO*, 2010.

[34] A. W. Services, "AWS Cloud Formation User Guide," 2010.

[35] A. Seznec, "A Case for Two-Way Skewed Associative Cache," *ISCA*, 1993.

[36] ——, "Concurrent Support of Multiple Page Sizes on a Skewed Associative TLB," *IEEE Transactions on Computers*, 2004.

[37] M. Spjuth, M. Karlsson, and E. Hagersten, "The Elbow Cache: A Power-Efficient Alternative to Highly Associative Caches," *Uppsala University Technical Report 2003-46*, 2003.

[38] M. Talluri and M. Hill, "Surpassing the TLB Performance of Superpages with Less Operating System Support," *ASPLOS*, 1994.

[39] VMware, "Large Page Performance: ESX Server 3.5 and ESX Server 3i v3.5," *VMware Performance Study*, 2008.

[40] ——, "VProbes Programming Reference," 2008.

[41] C. Waldspurger, "Memory Resource Management in VMware ESX Server," *OSDI*, 2002.

[42] T. Wood, G. Tarasuk-Levin, P. Shenoy, P. Desnoyers, E. Cecchet, and M. Corner, "Memory Buddies: Exploiting Page Sharing for Smart Colocation in Virtualized Data Centers," *VEE*, 2009.

[43] J. Xiao, Z. Xu, H. Huang, and H. Wang, "Security Implications of Memory Deduplicationi in a Virtualized Environment," *DSN*, 2013.

[44] Y. Xie, "Modeling, Architecture, and Applications for Emerging Non-Volatile Memory Technologies," *IEEE Computer Design and Test*, 2011.