Using TLB Speculation to Overcome Page Splintering in Virtual Machines

Rutgers University Technical Report DCS-TR-713, March 2015

Binh Pham* Jan Vesely* Gabriel H. Loh† Abhishek Bhattacharjee*

* Dept. of Computer Science, Rutgers University † AMD Research

{binhpham, jan.vesely, abhib}@cs.rutgers.edu gabriel.loh@amd.com

Abstract

As systems provide increasing memory capacities to support memory-intensive workloads, Translation Lookaside Buffers (TLBs) are becoming a critical performance bottleneck. TLB performance is exacerbated with virtualization, which is typically implemented with two-dimensional nested page tables. While virtualization vendors recommend using large pages to mitigate TLB overheads, in practice, page splintering (where the guest utilizes large pages, only to be broken by the hypervisor, or vice-versa) is prevalent and severely restricts large-page benefits. We characterize real-system page allocation behavior under virtualization and find that the guest operating system (OS) often aggressively allocates large pages, which are then broken by the hypervisor to baseline pages. However, in trying to allocate large pages, the hypervisor can (and does) often allocate these baseline system physical pages in aligned, contiguous frames with respect to the application's guest virtual address. In response, we propose hardware to, on TLB misses, speculate on the system physical address based on the application's virtual address. TLB speculation can be readily overlaid on various TLB organizations and is effective under a wide range of real-world virtual machine deployments. Overall, TLB speculation eliminates almost all of the TLB handling overheads of virtualization.

1. Introduction

With the advent of cloud computing, virtualization has become the primary strategy to consolidate diverse workloads (encapsulated in virtual machines or VMs) to ensure high utilization of physical machines. Many cloud providers, such as Amazon EC2 and Rackspace, use virtualization and consolidation in offering their services. The success of server-level virtualization is now being matched by interest from the mobile community, which is targeting virtualization for cell phones, tablets, and netbooks [50]. Virtualization success hinges on the hypervisor's or virtual machine monitor's (e.g., VMware ESX, KVM, Xen, Hyper-V) ability to consolidate as many VMs as possible while achieving good performance.

Memory management units (MMUs) and translation lookaside buffers (TLBs) are key determinants of VM consolidation and performance. The rise of memory-intensive, big-data workloads has forced system designers to accommodate increasingly larger memory capacities. Unfortunately, TLBs, even in the absence of virtualization, are stressed as a result, with miss penalties accounting for 5-20% of system runtime [10, 11, 15, 17, 18, 21, 24, 34, 35, 44]. Virtualization further exacerbates these overheads by requiring two-dimensional page table walks that first translate guest virtual pages (GVPs) to guest physical pages (GPPs), and then translate GPPs to system physical pages (SPPs) where data actually resides. This greatly increases TLB miss penalties, with recent studies by industry (e.g., VMware's performance analysis team) and academia concluding that TLB miss latencies are the largest contributor to the performance gap between native and virtual servers [20, 25].

Virtualization vendors and published work advocate using large pages (e.g., 2MB pages instead of baseline 4KB pages in x86-64) to counter these overheads [8, 23, 31, 44, 45, 47, 48]. Successful large page allocation and a TLB's ability to exploit its capacity benefits requires, however, both the guest OS and the hypervisor to back a memory region with a large page frame [14, 25]. This poses practical challenges in real-system deployments where hypervisors carefully reason about large pages because they can constrain consolidation opportunities and interfere with VM memory monitoring. As a result, hypervisors often choose to *splinter* or break guest large pages into system baseline pages; while these decisions are overall judicious as they improve consolidation ratios and memory management, they do present a lost opportunity.

This paper proposes novel TLB speculation techniques to reclaim the performance opportunity lost from page splintering. Our approach observes that even when hypervisors splinter guest large pages, they often (although they don't have to) align the ensuing baseline system physical pages within large page-sized memory regions corresponding to the alignment they would have had in an unsplintered system physical large page. This means that the page offset of guest virtual pages from large page-aligned boundaries (2MB boundaries in x86) are equal to the offsets between system physical pages and large page-aligned boundaries in system physical memory.

In response, we introduce Splintered Large-page Interpolation-based Speculation (SPLINTS)[1]. SPLINTS identifies large page-sized regions in the guest virtual address space and system physical address space with the alignment properties described, treating them as *speculative large pages*. By augmenting standard TLBs to store information for these regions, SPLINTS uses the guest virtual pages of memory references to *interpolate* a likely speculative system physical page number in scenarios where the TLB would otherwise have missed. Speculations are verified by page table walks, which are now removed from the processor's critical path of execution, effectively converting the performance of correct speculations into TLB hits. We show that not only is SPLINTS low-overhead and highly-accurate, it is software-

[1] Splints are commonly used to heal broken limbs; similarly, SPLINTS is used to mitigate the impact of broken large pages.

transparent and readily-implementable. Crucially, SPLINTS allows large pages to be compatible, rather than at odds, with judicious overall memory management. Specifically, our contributions are:

- We comprehensively study a range of real-world deployments of virtual machines, considering VMM types and architectures, and establish that guest OSs often choose to splinter pages for overall performance. We investigate the reasons for this which include transparent page sharing, memory sampling, TLB capacity issues, etc.
- We propose interpolation-based TLB speculation techniques to leverage splintered but well-aligned system physical page allocation to improve performance by an average of 14% across our workloads. Overall, this is almost all of the TLB overheads of virtualization.
- We investigate various design tradeoffs and splintering characteristics to study SPLINTS' wide applicability. Across all of our configurations, systems generate enough of the alignment properties SPLINTS needs for performance gains, making TLB speculation robust.

2. Background

Hardware-assisted virtualization for TLBs: Early support for virtualization reconciled guest and hypervisor (or nested) page tables using hypervisor-maintained shadow page tables [9, 46] and by caching direct translations from guest virtual pages to system physical pages in the TLB. Unfortunately, shadow paging incurs many expensive VM exits [7, 14]. In response, x86 vendors introduced hardware for two-dimensional page table walks via AMD's nested page table support and Intel's extended page table [14]. In such systems, when the guest references a virtual address that misses in the TLB, 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 [14]. This paper focuses on techniques to attack these high-latency page table walks.

Large pages: To counter increased TLB overheads in virtual servers, virtualization vendors encourage using large pages aggressively [20]. OSs 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 [5]. 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), reduce the number of levels in the page table (reducing miss penalty), and reduce page table size.

In virtualized servers, these benefits are all magnified (compared to the native case) as they are applied to two page tables. For example, 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 exists in both the guest and the hypervisor page table.

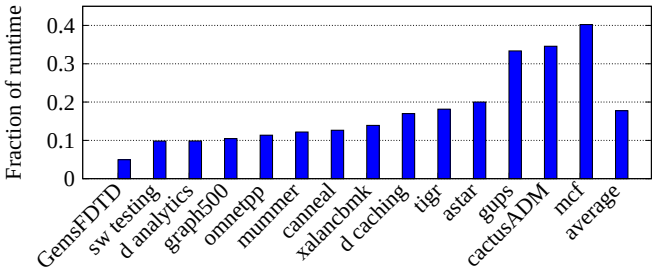

Figure 1: Percent of execution time for address translation, for applications on a Linux VM on VMware's ESX server, running on an x86-64 architecture. Overheads are 18% on average.

3. Splintering Challenges and Opportunities

3.1. Real-System Virtualized TLB Overheads

Figure 1 quantifies address translation overheads on real-system virtual machine deployments, normalized to total runtime. As detailed in Section 5, one set of our experiments assumes Ubuntu 12.04 server (3.8 kernel) as the guest OS, running on VMware's ESX 5.5 hypervisor, on an x86-64 Intel Sandybridge architecture. Although omitted here for space reasons, we have also assessed these overheads running KVM on x86-64, and KVM on an ARM architecture with Cortex A15 cores, and have found the same trends. We use SPEC, PARSEC [19], and CloudSuite [22] workloads and all measurements use on-chip performance counters.

The data show that two-dimensional page table walks degrade performance, requiring an average of close to 20% of runtime. Overheads vary, with `data analytics` from CloudSuite and `graph500` suffering 10% overheads while `mcf` and `cactusADM` suffer well over 30% overheads. Our results corroborate past work [20] that identified TLB overheads as a significant performance bottleneck in virtualized servers, regardless of architecture or hypervisor vendor.

3.2. Real-System Splintering

Figure 2 quantifies the prevalence of splintering across the workloads, when running VMware's ESX on the x86-64 architecture. For each benchmark's TLB misses, we plot the fraction eventually serviced from splintered large pages, small pages in both dimensions, and large pages in both dimensions. 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), accounting for over

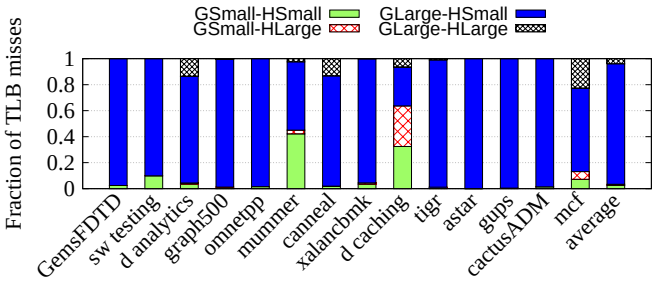

Figure 2: Fraction of TLB misses serviced from a system physical page that is backed by a small page in both guest and hypervisor (GSmall-HSmall), small in guest and large in hypervisor (GSmall-HLarge), large in guest and small in hypervisor (GLarge-HSmall), and large in both (GLarge-HLarge).

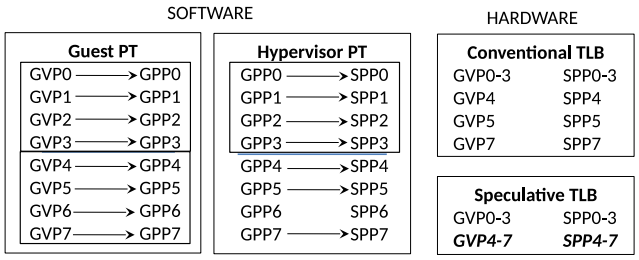

Figure 3: Guest large page GVP4-7 is splintered, but SPPs are conducive to interpolation. A speculative TLB maps the page table in two entries (using a speculative entry for GVP4-7) instead of four entries (like a conventional TLB).

90% of TLB misses on average. Some workloads like `data analytics`, `canneal`, and `mcf` are able to more aggressively use large pages in both dimensions (usually because they allocate large data structures at once) but attacking splintered guest large pages remains the significant opportunity.

This lost opportunity is particularly regrettable because modern guest operating systems go to great length to attempt to create large pages. For example, one may initially consider that increased system load (leading to memory ballooning) and fragmentation would constrain the guest from generating large pages. Our analysis in Section 6.5, however, shows that even in aggressively loaded and fragmented scenarios, modern OSs have memory defragmentation facilities that do an excellent job of creating contiguous runs of physical memory necessary for guest large pages. That these pages are splintered is a major lost opportunity.

In addition to our VMware ESX experiments, we also profile splintering using KVM on x86-64, and KVM on ARM, finding the same general trends. Section 6.4 showcases our results, making the point that ample splintering remains.

3.3. High-Level Overview of SPLINTS

SPLINTS leverages cases where GVPs and SPPs are aligned within 2MB memory regions corresponding to the alignment they would have had in an unsplintered page. In other words, we exploit cases where VPPs and SPPs share their 9 least significant bits (for 4KB baseline pages and 2MB large pages). We have profiled workload page tables in many real-world deployments and found that over 94% (average of 97%) of the 4KB pages *for every single benchmark in every deployment* have GVPs and SPPs sharing their bottom 9 bits. This motivates interpolation-based TLB speculation.

Figure 3 shows high-level SPLINTS operation. For illustration, 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 GVP 6 (and GPP 6) are unallocated. A conventional TLB requires one large page entry and three baseline entries to fully cover this splintered large page. SPLINTS, 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, SPLINTS 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, SPLINTS achieves performance benefits by leveraging *one-dimensional* large pages where the guest large page is splintered by the hypervisor, to try and get the same benefits as 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 to the range of real-world splintering scenarios.

SPLINTS requires careful design to be effective. It must distinguish baseline PTE entries, regular large page PTE entries, and our speculative large page PTE entries. Similarly, interpolation must be performed with low-overhead hardware (we show that simple bit concatenation operations suffice). Because incorrect speculation degrades performance (e.g., pipeline flush and refetch) and because all speculation has verification energy overheads (from additional page table walks), SPLINTS must judiciously speculate.

3.4. Sources of Page Splintering

Empirically, we find that hypervisors often decide to splinter pages for compelling reasons, which enhance overall system performance. We now briefly discuss some of these reasons, showing the tight tradeoff between two-dimensional large pages and VM memory monitoring and management.

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 [49]. 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 tracked. This

fraction is used to deduce a VM's working set size.

Unfortunately, if a randomly chosen 4KB region falls within a large 2MB page, the large page is splintered by the hypervisor. These baseline pages may be promoted at the end of the sample period; however, other memory usage characteristics may have changed at this point, precluding promotion. We comprehensively evaluate the effect of working set sampling on page splintering in Section 6.4.

At first blush, one may consider "repairing" working set estimation in software. This, however, is difficult; for example, one might use page table 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 or Intel chips older than Haswell. In particular, architectures like ARM, which implement relaxed memory consistency models struggle to accommodate page table access and dirty bits, which require sequentially consistent reads and writes for correct operation [36]. In addition, our conversations with hypervisor vendors like VMware suggest that they prefer designing software for general applicability, and are therefore loath to designing software modules for specific architectures with these bits.

Page sharing: Real-world cloud deployments attempt to consolidate as many VMs on the same physical node as possible. Transparent page sharing [6, 8] is critical for high consolidation, reducing memory requirements of multiple VMs by identifying memory pages with the same content within and between VMs, and eliminating redundant copies. While effective particularly in memory-overcommitted systems, page sharing prompts the hypervisor to splinter guest large pages for two reasons (see Section 6.4). First, baseline pages are much likelier to have equivalent content as they are smaller than large pages. Second, page sharing daemons read memory content by computing checksums for individual pages and storing them in hypervisor metadata tables. Checksum compute time and metadata table size increase with page size [8].

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 only a limited number of OSs (e.g., Linux, Windows, and Solaris). Furthermore, these systems commonly run VMs from one "template" [39], running near-identical workloads and OS images. Unfortunately, however, there is no easy way to reconcile page sharing with large pages; Microsoft's Hyper-V takes the alternate approach of prioritizing large pages but does so by ignoring page sharing, and hence suffers lower consolidation capabilities [26].

Limited-capacity large-page TLBs: Processors usually maintain separate TLBs for different page sizes [12, 35]. For example, x86 processors maintain larger 4KB TLBs (e.g., 64-entry), smaller 2MB TLBs (e.g., 32-entry), and smaller still 1GB TLBs (e.g., 8-entry). Because of the limited size of large-page TLBs, misuse of large pages may hurt performance. Specifically, if an application's working set is scattered over a wide address space range, large page TLB thrashing occurs [12, 47]. System administrators may therefore disable the hypervisor's ability to back guest large pages [47].

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, our conversations with VMware engineers suggest that for a significant number of cases, splintering remains at the destination node, especially if there unfragmented free physical memory space to accommodate large pages is scarce there. In these instances, page splintering is a serious performance limiter.

Real-world cloud deployment issues: Some cloud deployments, like Amazon's EC2, disable hypervisor large pages for performance reasons [51]. This has posed well-known problems recently for Oracle users, who have been advised to "run inefficiently without huge pages on a fully-supported Oracle-VM backed instance" [51].

Lack of hypervisor support for large pages: Finally, hypervisor vendors can take a few production cycles before fully adopting large pages. For example, VMware's ESX server currently has no support for 1GB large pages in the hypervisor, even though guests on x86-64 systems can use them.

Even when splintering large pages, hypervisors often preserve the property that 4KB system physical pages are aligned within 2MB memory regions corresponding to the alignment they would have had in an unsplintered 2MB page. As just one example, inter-VM page sharing splinters 2MB pages but leaves the constituent 4KB pages in the aligned regions where they already were. SPLINTS uses these observations to speculate by interpolation.

4. SPLINTS Microarchitecture

This section details hardware for interpolation-based TLB speculation. We begin by describing the baseline TLB organization we assume,[2] and how speculation can be overlaid on it. We then describe speculation details and hardware tradeoffs.

4.1. TLB Organization

We assume a processor organization that includes per-core two-level TLB hierarchies, as is typical in modern processors [12, 15, 18]. Upon a memory reference, two L1 TLBs are looked up in parallel, one devoted to 4KB PTEs and an-

[2] Our proposal does not depend on this specific TLB organization, but we describe it to provide a concrete example to explain our technique.

other to 2MB PTEs exclusively. L1 misses[3] prompt a lookup in the L2 TLB, typically reserved for 4KB PTEs.

SPLINTS 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. SPLINTS, however, creates a *speculative 2MB entry for the one-dimensional large page*. We investigate two SPLINTS variants:

L1-only speculation: Here, speculative 2MB entries are placed only in the 2MB L1 TLB. Therefore, speculation is permitted only at the first-level of the TLB hierarchy. This is a minimally-intrusive design because the set-indexing scheme of the 2MB L1 TLBs readily accommodates 2MB speculative entries. 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 the 2MB L1 TLB *and* the L2 TLB. Our baseline assumes L2 TLBs that cache only 4KB PTEs (typical of many processors); hence we must modify the L2 TLB to accommodate PTEs of multiple page sizes. While SPLINTS can use any strategy to achieve this, this work leverages skew-associativity techniques [32, 37, 40–42] and hash-rehashing (column-associativity) [2]. Because SPLINTS is similarly effective in both cases, our results focus on skew-associative TLBs. Note that processor vendors have already begun incorporating TLBs that support concurrent page sizes; for example, Intel's Haswell cores reportedly implement L2 TLBs that concurrently support both 4KB and 2MB PTEs [28]. SPLINTS merely leverages already-implementable techniques, and is not tied to skewing or hash-rehash.

We do not explore speculation possibilities on the 4KB L1 TLB because this would require modifying it to support multiple page sizes via skewing (or hash-rehashing) as well. While this is certainly possible, we focus on less-intrusive designs that leave timing-critical L1 lookups unaffected.

4.2. Speculative TLB Entries

Figure 4 shows the structure of a speculated 2MB entry in the L1 2MB TLB. The structure is the same as a standard 2MB entry, with only a `Spec` bit added to distinguish speculated 2MB entries from standard 2MB entries (necessary to ensure that L2 TLB and page table walker are probed to verify speculation correctness). This additional bit represents a negligible 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. 2MB entries require 9 fewer tag bits than 4KB entries, and are hence unused for speculated 2MB frames; we recycle one of these bits for the `Spec` field.

[3] 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").

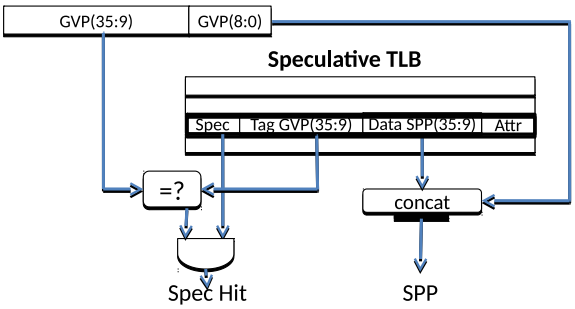

Figure 4: Lookup operation on a speculated TLB entry. A tag match is performed on the bits corresponding to its 2MB frame. On a hit, the 2MB frame in system physical memory is concatenated with the 4KB offset within it.

4.3. Key TLB Operations

Lookups: Figure 4 shows how SPLINTS 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 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: SPLINTS fills speculated 2MB entries into the L1 and L2 TLBs after a page table walk. Suppose a GVP request misses in both L1 TLBs and the L2 TLB. The 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 support* beyond standard page table walks.

TLB capacity, bandwidth tradeoffs: A key benefit of large-page entries in TLBs is their greater memory reach. In x86-64 systems, one large page TLB entry replaces 512 4KB page entries. TLB speculation can potentially achieve similar capac-

ity, but at the cost of increased verification bandwidth requirements. Specifically, L1-level speculation verification always requires L2 TLB lookups and may even require page table walks on L2 TLB misses. Similarly, L2-level speculation verification always requires page table walks. Without care, verification energy requirements and potential performance loss (from greater L2 TLB port contention and cache lookups for page table walks) can make speculation impractical.

We address this by, even on correct speculation at the L1 level, inserting the desired non-speculative 4KB PTE into the TLB(s) after locating it with verification. Because the PTE is likely to be reused soon, future accesses experience non-speculative TLB hits, minimizing verification costs.

We consider multiple approaches to balance the capacity benefits of speculative 2MB entries with verification costs. We compare inserting non-speculative 4KB PTEs into the L1 TLB (reducing verification 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 don't insert the non-speculative 4KB entry into the TLBs at all (maximizing capacity), and where we add the non-speculative 4KB PTE in both TLBs (minimizing verification).

In general, we have found (Section 6) 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.

4.4. Speculation Details

We now study various aspects of SPLINTS, 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 SPLINTS. 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 SPLINTS 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 a matching 4KB 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 then 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.

4.5. Mitigating Mis-speculation Overheads

TLB mis-speculation requires flushing the pipeline and refetching from the offending memory reference. 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 stale. At the point of misspeculation detection, the SPP is now known, and so 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.

5. Methodology

To evaluate functional, behavioral, and performance effects, we examine systems running multiple real OSs and hypervisors. At the same time, 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 [11, 12, 18, 25, 34, 35], we use a combination of techniques

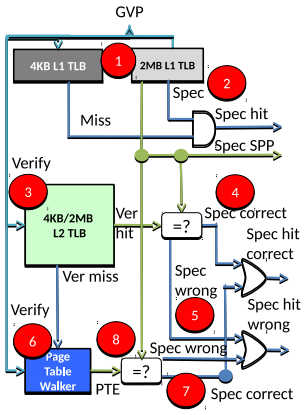

Figure 5: The mechanics of TLB speculation. We show the case when we speculate from the 2MB L1 TLB

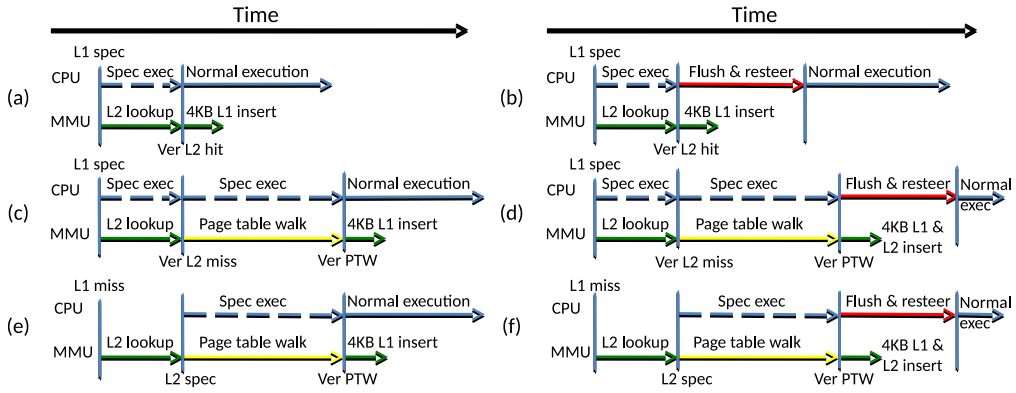

Figure 6: Timelines recording speculation and verification activities for (a) speculating from the 2MB L1 TLB correctly, and verifying this in the L2 TLB; (b) speculating from the 2MB L1 TLB incorrectly, and verifying this in the L2 TLB; (c) speculating from the 2MB L1 TLB correctly, and verifying this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

including tracing and performance counter measurements on real machines, functional cache-hierarchy and TLB simulators, and an analytical modeling approach to estimate the overall impact on program execution time.

5.1. Workloads

We set up our virtualization environment on a host machine with 8 CPUs and 16GB RAM. 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) [5]. 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, including SPEC CPU 2006, BioBench [4], and CloudSuite [22], which have non-negligible TLB miss overheads as shown in Figure 1. We present results on workloads sensitive to TLB activity.

5.2. Trace Collection

We use Pin [29] to collect guest memory traces for our workloads. Because Pintools only output virtual addresses, we first extend the Linux's pagemap to include physical addresses and intermediate page table entries (PTE), and then our Pintool reads this information and incorporates it into the generated traces. For each workload, we select a PinPoint region of one billion instructions [33], and we correlate the generated trace with performance counter measurements to verify that the sampled billion-instruction region is representative of the overall benchmark.

We use scripts supported by VMware to collect hypervisor memory traces that contain guest physical pages, system physical pages, and the corresponding page size. Finally, we merge the guest with hypervisor trace to generate a complete trace for our simulator. We also extend the tracing utility to VMs on KVM hypervisor to get the similar information.

5.3. 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), similar to Intel's Sandybridge cores. We also consider (and model) an alternative L2 TLB baseline that makes use of a skewed-associative organization [41]. 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 [10, 18]. 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 also makes use of on TLB misses.

5.4. 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 SPLINTS, 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 the hit/miss and speculation/misspeculation scenarios (see Figure 6).

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[4], 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 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:

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

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 4.5, 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 max() 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 [12].

[4]We are confident in this extrapolation because we have validated that our one-billion instruction traces' performance characteristics closely match those collected from the performance counters over the full program executions.

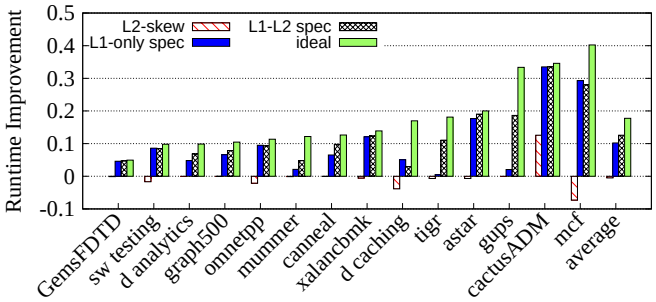

Figure 7: Performance benefits of L1-only, L1-L2 speculation, only skewing the L2 cache, compared to the ideal case without TLB miss overhead. Performance is normalized to the baseline single-VM.

6. Experimental Results

We now evaluate SPLINTS, and explore several system-level issues and along with the prevalence of page splintering.

6.1. SPLINTS Performance Results: Single VM

We begin by considering the performance improvements afforded by L1-only and L1-L2 speculation, and show the importance of careful speculation to control verification costs.

SPLINTS performance: Figure 7 quantifies the benefits of SPLINTS (all results are normalized to the runtime of the application on a single VM), showing that they eliminate the vast majority of TLB overheads on virtualization. We also modify the L2 TLB to support multiple page sizes concurrently via skewing; to isolate any performance improvements from skewing alone, we show its results too. Overall, Figure 7 shows that skewing alone is minimally effective, increasing performance a bit for cactusADM but either being ineffective, or actually degrading performance (e.g., mcf) in other cases. However, when combined with SPLINTS (L1-L2 speculation), results improve drastically. On average, runtime is improved by 14%, just 4% away from the performance of an ideal system with no TLB overheads. Most benchmarks are actually significantly closer to the ideal case with only mummer, data caching, tigr, 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 tigr, 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 10%, 18%, and 30% performance gains, respectively.

Interestingly, Figure 7 also shows that L1-only speculation is highly-effective, achieving 10% performance benefit. In fact, only mummer, tigr, and gups see significantly more performance from speculating with both L1 and L2 TLBs. Some cases (e.g. data caching and mcf), even see higher performance gain for L1-only speculation than L1-L2 speculation, although the difference is small. This is because non-speculative and speculative entries compete for L2 capacity in

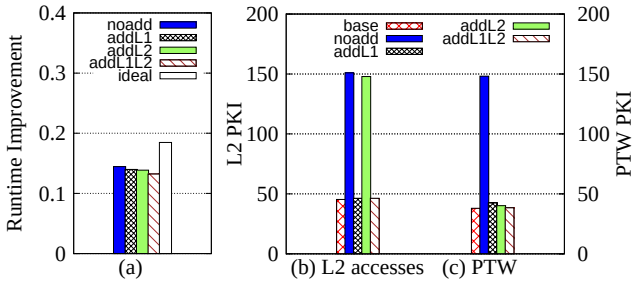

Figure 8: Average (a) performance improvements when inserting the non-speculative 4KB PTE, after correct speculation, in neither TLB (noAdd), the L1 TLB (addL1), the L2 TLB (addL2), or both (addL1L2), compared with the ideal improvement; (b) number of L2 TLB accesses per kilo-instruction (APKI) including verification compared to a baseline with speculation; and (c) number of page table walks per kilo-instruction.

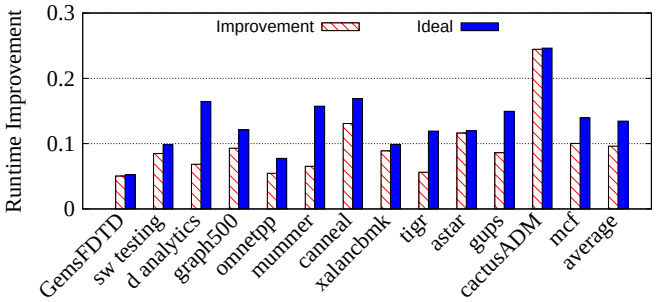

Figure 9: Performance gains achieved by SPLINTS on a multi-VM configuration, compared against the ideal performance improvement where all address translation overheads are eliminated.

L1-L2 speculation. Overall, even simple SPLINTS schemes are effective.

Capacity versus verification bandwidth tradeoff: 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 8 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 for every single benchmark.

Figure 8(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 1%). Figures 8(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.

6.2. SPLINTS Performance Results: Multiple VMs

VMs with similar workloads: We have investigated the benefits of TLB speculation in scenarios with multiple virtual machines, which potentially change splintering rates because of inter-VM page sharing, etc. We have studied scenarios with 2, 3, 4, and 8 VMs but find that the performance trends remain the same. Therefore, we show 8-VM results for the Spec and PARSEC workloads. Cloudsuite applications, which have far greater memory needs, overcommit system memory with fewer VMs; we hence present 4-VM studies for these workloads. Figure 9 quantifies the performance improvement SPLINTS offers on the multi-VM setup (averaged across VMs, since we find negligible inter-VM variance in results), compared to an ideal scenario where all address translation overheads have been removed. In general, multiple VMs stress the TLB hierarchy even further due to greater contention. Fortunately, even with multiple VMs, sufficient page splintering and good alignment exists, letting SPLINTS (with L1-L2 speculation) eliminate the bulk of this overhead: 75% on average. Workloads like `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, SPLINTS has high real-world utility.

VMs with different workloads: We also study the more atypical case where one physical node runs multiple VMs with the same OS (stock Linux) but different workloads. On a range of configurations, 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 [6]. Fortunately, SPLINTS is effective at countering page sharing-induced splintering even when VMs run different workloads, greatly boosting performance.

6.3. Analyzing TLB Miss Rates

We now deconstruct the sources of performance benefit. To this end, Figure 10 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 remainder of these events are removed from the execution critical path (because they are correctly speculated) and hence do not incur a performance penalty. Overall, Figure 10 shows that TLB speculation removes 45% of the L2 TLB lookups and 80% of the page tables walks from the critical path. In multi-VM scenarios, these elimination rates remain consistent, explaining the performance improvements.

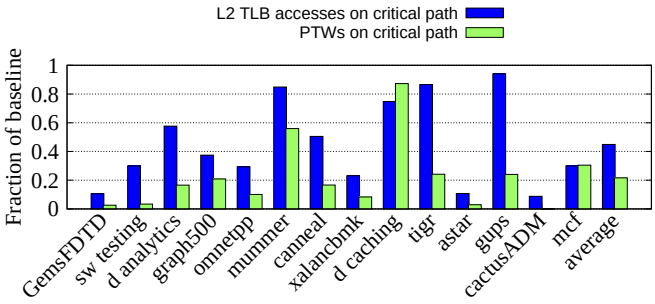

Figure 10: Fraction of the baseline L2 TLB accesses and page table walks remaining on the critical path of execution with TLB speculation. Performance improves because the bulk of these are removed from program critical path.

6.4. Characterizing Page Splintering Sources

Page splintering across hypervisors and hardware architectures: As detailed in Section 3, we have assessed the prevalence of page splintering across different hypervisor classes and architectures. We have found that splintering on our KVM/x86-64 and KVM/ARM setups closely mirror our results from ESX/x86-64. Figure 11(a) shows the average splintering data, demonstrating that page splintering is a problem that can be observed across architectures and hypervisors. A general-purpose hardware solution like SPLINTS offers the opportunity to mitigate this problem on a range of architectures and hypervisors.

Sources of page splintering and alignment: We now isolate the impact of two sources of page splintering, namely working set sampling and page sharing. By default, ESX turns these facilities on for better overall performance [8].

Figure 11(b) 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 page sharing is turned off, and both page sharing and sampling are turned off. Within a VM, zero pages are typically shared with few other pages being redundant. Therefore, on average and for `canneal`, turning off page sharing reduces splintering marginally. For `mummer`, the change is more pronounced. However, turning off memory sampling has an enormous impact on page splintering. On average, almost all guest large pages are now backed by hypervisor large pages.

Figure 11(c) 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 when multiple VMs are run.

6.5. Importance of SPLINTS in Future Systems

SPLINTS targets scenarios where the guest OS has gone to the trouble of generating large pages, which are then splintered by the hypervisor. As such, an implicit assumption is that the guest can indeed create large pages, which is purely a function of OS large page support. Modern operating systems have sophisticated and aggressive support for large pages [5, 35, 44], 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, VM's 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 [5, 35] 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 [35]. 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. TLB speculation 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 KVM exposes ballooned pages to the memory defragmentation software run by the kernel, ballooning has no impact on guest large page generation [5]. Even for ESX, which cannot currently expose ballooned pages to memory defragmentation daemons, Figure 11(d) shows that we find ballooning to only slightly decrease guest large pages, in the multi-VM case (we show the average case and benchmarks with the largest change, `mcf` and `graph500`). Furthermore, our conversations with VMware engineers have revealed that they are likely to also adopt defragmentation for ballooned pages, just like KVM and Xen [23].

Overall, these trends suggest that it is even more imperative in future systems that SPLINTS-style TLB splintering mitigation techniques be used. Guest OSs will undoubtedly continue generating large pages, which will remain wasted without mechanisms like SPLINTS.

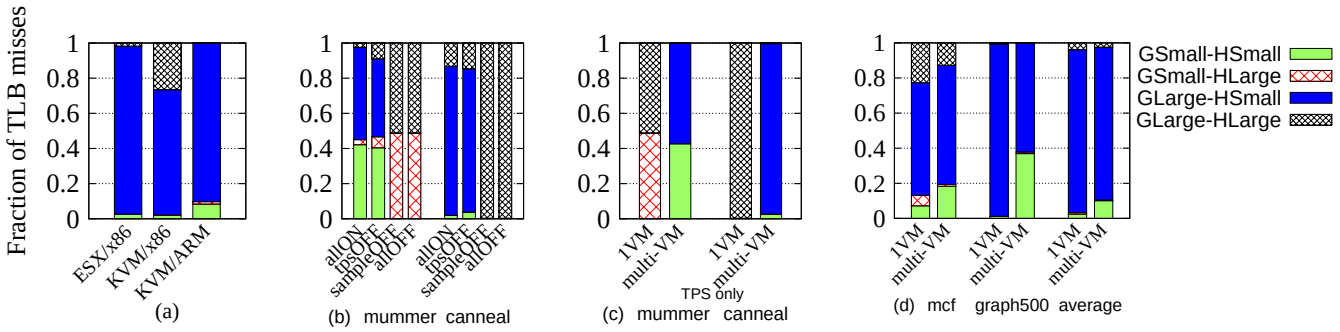

Figure 11: (a) Splintering across hypervisors. (b) Effect of memory sampling and intra-VM page sharing on page splintering, when running 1 VM. The default setting (allON) enables page sharing and sampling, tpsOFF disables transparent page sharing, sampleOFF disables sampling, allOFF disables sampling and page sharing. (c) Effect of inter-VM page sharing on page splintering when multiple VMs are run. (d) Page splintering in the presence of ballooning.

6.6. Understanding the Limitations of TLB Speculation

SPLINTS 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, SPLINTS 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 this happens in practice, but nevertheless we revisit cache prefetching optimizations to mitigate TLB mis-speculation penalties. Section 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 [15, 34, 35], 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 high speculation accuracy, cache prefetching is not really necessary for performance (on average, we gain roughly 1% more performance) for our workloads. However, we have calculated the minimum correct speculation rate required to ensure *no performance loss*. For every single benchmark, a correct speculation rate of at least 48% guarantees that performance will *not be degraded*; even if half the speculations are incorrect (a very pessimistic scenario), performance is unaffected. In practice, all our correct speculation rates have far exceeded 90%. However, should this be deemed an issue on some systems, standard confidence-based adaptive techniques could be applied to modulate speculation aggressiveness as necessary.

7. Related Work

Recent efforts take note of the pressure that emerging big data workloads are placing on address translation [10–13, 15–18, 30, 34, 35]. In response, TLB prefetching [17, 27, 38], shared last-level TLBs [15, 30, 52], synergistic TLBs [43], direct segments [12], sub-blocking [44], coalescing [35], and clustering [34] approaches have all been proposed. Of these, our work is closest in spirit to SpecTLB [11], where the authors target interpolation-based speculation for bare-metal systems with operating systems like FreeBSD that use reservation-based large page policies. In these systems, the authors note that baseline physical pages are allocated in well-aligned patterns similar to the ones SPLINTS exploits. The authors implement a separate speculative L1 TLB to cache information about these reservations, substantially boosting performance.

We also use TLB speculation; however, unlike SpecTLB but like other recent work on virtualization [3, 14], we target the overheads of two dimensional page-table walks. We target virtualized environments and mechanisms that can be applied to guest OSs beyond those that use reservation-based large pages. We also propose different microarchitectural solutions (e.g., we integrate speculation into the existing TLB hierarchy and study various design tradeoffs) tailored to splintering issues in virtualization.

8. Conclusion

This work proposed and evaluated the benefits of TLB speculation to mitigate page splintering in virtualized systems. We observe that though guest large pages are often broken by the hypervisor (e.g., memory sampling, page sharing, lack of large page support), the alignment of the ensuing baseline pages are often conducive to interpolation-based speculation. We conducted a comprehensive set of experiments across single- and multi-VM scenarios, and showed that low-overhead, hardware-only speculation schemes can be overlaid on existing TLB organizations to eliminate almost all of the virtualization-related address translation overheads. Overall, while we observed that splintering is a problem and can cause significant performance problems, our proposed SPLINTS architecture can largely mitigate these issues, thereby making virtualized systems more attractive to deploy.

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] J. Ahn, S. Jin, and J. Huh, "Revisiting Hardware-Assisted Page Table Walks for Virtualized Systems," *ISCA*, 2012.

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

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

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

[7] C. Bae, J. Lange, and P. Dinda, "Enhancing Virtualized Application Performance Through Dynamic Adaptive Paging Mode Selection," *ICAC*, 2011.

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

[9] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," *SOSP*, 2003.

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

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

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

[13] A. Basu, M. Hill, and M. Swift, "Reducing Memory Reference Energy with Opportunistic Virtual Caching," *ISCA*, 2012.

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

[15] A. Bhattacharjee, D. Lustig, and M. Martonosi, "Shared Last-Level TLBs for Chip Multiprocessors," *HPCA*, 2010.

[16] A. Bhattacharjee and M. Martonosi, "Characterizing the TLB Behavior of Emerging Parallel Workloads on Chip Multiprocessors," *PACT*, 2009.

[17] ——, "Inter-Core Cooperative TLB Prefetchers for Chip Multiprocessors," *ASPLOS*, 2010.

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

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

[20] 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.

[21] J. B. Chen, A. Borg, and N. Jouppi, "A Simulation Based Study of TLB Performance," *ISCA*, 1992.

[22] 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.

[23] A. Gadre, K. Kabra, A. Vasani, and K. Darak, "X-XEN: Huge Page Support in Xen," *Ottawa Linux Symposium*, 2011.

[24] J. Gandhi, A. Basu, M. Hill, and M. Swift, "Badger-Trap: A Tool to Instrument x86-64 TLB Misses," *CAN*, 1994.

[25] ——, "Efficient Memory Virtualization," *MICRO*, 2014.

[26] http://www.pocs.nl/pocs/fr/technique/vmware/83-vmware-esx5i-vs-hyper v.pdf, "VMware ESX5i vs Hyper-V," 2013.

[27] G. Kandiraju and A. Sivasubramaniam, "Going the Distance for TLB Prefetching: An Application-Driven Study," *ISCA*, 2002.

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

[29] 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 *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, Chicago, IL, June 2005, pp. 190–200.

[30] D. Lustig, A. Bhattacharjee, and M. Martonosi, "TLB Improvements for Chip Multiprocessors: Inter-Core Cooperative Prefetchers and Shared Last-Level TLBs," *TACO*, 2012.

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

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

[33] 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 *37th Intl. Symp. on Microarchitecture*, Portland, OR, December 2004.

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

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

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

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

[38] A. Saulsbury, F. Dahlgren, and P. Stenström, "Recency-Based TLB Preloading," *ISCA*, 2000.

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

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

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

[42] 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.

[43] S. Srikantaiah and M. Kandemir, "Synergistic TLBs for High Performance Address Translation in Chip Multiprocessors," *MICRO*, 2010.

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

[45] M. Talluri, S. Kong, M. Hill, and D. Patterson, "Tradeoffs in Supporting Two Page Sizes," *ISCA*, 1992.

[46] L. van Doorn, "Hardware Virtualization Trends," *VEE Keynote*, 2006.

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

[48] ——, "Performance Best Practices for VMware vSphere 5.0," *VMware*, 2011.

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

[50] S. Whittle, "VMware Forsees Mobile Virtualization in 2010," *http://www.cnet.com/news/vmware-forsees-mobile-virtualization-in-2010*, 2010.

[51] J. Wilton, "Linux Hugepages for Oracle on Amazon EC2: Possible, but not Convenient, Easy, or Fully Supported," *http://www.pythian.com/blog/linux-hugepages-for-oracle-on-amazon-ec2-possible-but-bot-convenient-easy-or-fully-supported*, 2013.

[52] L. Zhang, E. Speight, R. Rajamony, and J. Lin, "Enigma: Architectural and Operating System Support for Reducing the Impact of Address Translation," *ICS*, 2010.