Rethinking TLB Designs in Virtualized Environments: A Very Large Part-of-Memory TLB

Jee Ho Ryoo Nagendra Gulur[†] Shuang Song Lizy K. John
The University of Texas at Austin [†]Texas Instruments
jr45842@utexas.edu

ABSTRACT

With increasing deployment of virtual machines for cloud services and server applications, memory address translation overheads in virtualized environments have received great attention. In the radix-4 type of page tables used in x86 architectures, a TLB-miss necessitates up to 24 memory references for one guest to host translation. While dedicated page walk caches and such recent enhancements eliminate many of these memory references, our measurements on the Intel Skylake processors indicate that many programs in virtualized mode of execution still spend hundreds of cycles for translations that do not hit in the TLBs.

This paper presents an innovative scheme to reduce the cost of address translations by using a very large Translation Lookaside Buffer that is part of memory, the *POM-TLB*. In the *POM-TLB*, only one access is required instead of up to 24 accesses required in commonly used 2D walks with radix-4 type of page tables. Even if many of the 24 accesses may hit in the page walk caches, the aggregated cost of the many hits plus the overhead of occasional misses from page walk caches still exceeds the cost of one access to the *POM-TLB*. Since the *POM-TLB* is part of the memory space, TLB entries (as opposed to multiple page table entries) can be cached in large L2 and L3 data caches, yielding significant benefits. Through detailed evaluation running SPEC, PARSEC and graph workloads, we demonstrate that the proposed *POM-TLB* improves performance by approximately 10% on average. The improvement is more than 16% for 5 of the benchmarks. It is further seen that a *POM-TLB* of 16MB size can eliminate nearly all TLB misses in 8-core systems.

CCS CONCEPTS

• **Computer systems organization** → **Heterogeneous (hybrid) systems**;

KEYWORDS

Address Translation, Very Large TLB, Virtualization, Die-Stacked DRAM

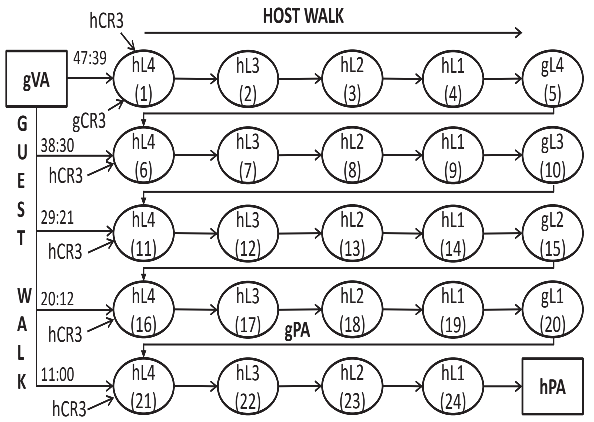

Figure 1: x86 2D Page Walk In Virtualized Environment

ACM Reference format:
Jee Ho Ryoo Nagendra Gulur[†] Shuang Song Lizy K. John The University of Texas at Austin [†]Texas Instruments . 2017. Rethinking TLB Designs in Virtualized Environments: A Very Large Part-of-Memory TLB. In *Proceedings of ISCA '17, Toronto, ON, Canada, June 24-28, 2017,* 12 pages.
https://doi.org/10.1145/3079856.3080210

1 INTRODUCTION

Cloud services such as Amazon EC2 [1] or Rackspace OpenStack [40] use virtualization platforms to provide their services. These platforms use hypervisors (ESX, KVM, Xen) to enable easier scalability of applications and higher system utilization by abstracting the underlying host resources. While virtualization provides many benefits, the overhead of virtualization is not universally low [17].

One of the largest contributors of performance overhead in virtualized environments is memory virtualization. An application executing on a guest OS generates guest virtual addresses (*gVA*) that need to be translated to host physical addresses (*hPA*). Since physical memory is under the exclusive control of a hypervisor, every guest *physical* address (*gPA*) needs to be translated to host physical before the guest application-issued memory access can complete. This requires navigating through two sets of page tables: a guest page table that the guest OS implements ($gVA \rightarrow gPA$), and a host page table that the hypervisor implements ($gPA \rightarrow hPA$). In x86 architectures, both the guest and host page tables employ a 4-level radix-tree table organization. Translating a virtual address to physical address takes 4 memory references in a bare metal case using a radix-4 table, and in the virtualized case, it becomes a full $2D$ translation with up to 24 memory accesses as depicted in Figure 1.

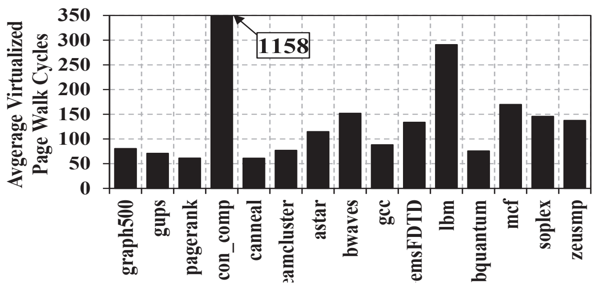

Figure 2: Average Translation Cycles per L2 TLB Miss (Virtualized Platform)

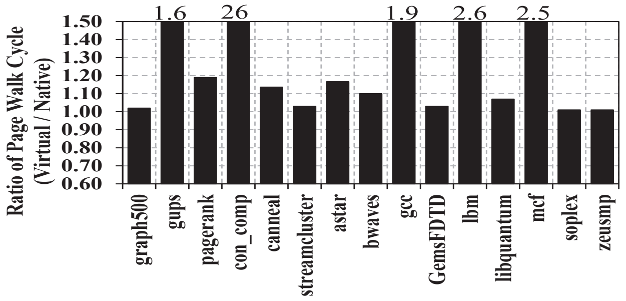

Figure 3: Ratio of Virtualized to Native Translation Costs

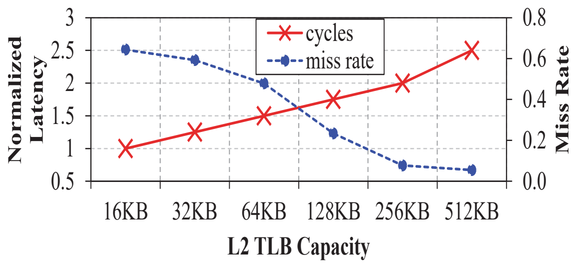

Figure 4: L2 TLB Scaling Trend

In order to bridge the performance gap associated with address translations, recent processors have added architecture support in the form of nested page tables [7] and extended page tables [22] that cache guest-to-host translations. Processor vendors have also added dedicated MMU/page walk caches [4, 23] to cache the contents of guest and host page tables. Additional techniques to reduce the overhead of page walks include caching of page table entries in data caches, agile paging [17], TLB prefetching [10], shared L2 TLBs [9], transparent huge pages (THP) [3], speculative TLB fetching [5] and splintering [38]. These page walk enhancements have significantly reduced translation costs, however the translation overhead continues to be a source of inefficiency in virtualized environments.

Figure 2 based on our experiments on a state-of-the-art Intel system (Skylake i7-6700) shows the average number of cycles spent in address translation (per L2 TLB miss) in several SPEC, PARSEC and graph workloads. The workloads are run on a VM, with Linux THP support enabled and the performance overhead is measured using the Linux *perf* utility. Further details of our experimental setup are presented in Section 3. The translation overhead per L2 TLB miss is seen to range from 61 cycles in the *canneal* benchmark to 1158 cycles in the *connected component* graph benchmark despite the fact that Skylake core includes special MMU/paging structure caches (PSCs). Translation overhead running into 100+ cycles has also been reported in prior work [12, 16, 38].

The experiments on the Intel Skylake platform also shed light on the virtualization overhead compared to native execution of the same workload. Figure 3 plots the ratio of translation cycles in virtualized and native setups. Workloads such as gups (1.5x), con_comp (26x), gcc (1.9x), lbm (2.5x) and mcf (2.5x) have far higher translation overhead in virtualized execution compared to native execution. Many benchmarks spend up to 14% execution time in translation even in the bare metal case and hence will benefit from the proposed scheme which improves both native and virtualized cases.

With the increased number of cores and big data sets, the conventional two-level SRAM TLBs cannot hold translations of all pages in the working set. Increasing L2 TLB sizes to sufficiently reduce TLB misses is not feasible because larger SRAM TLBs incur higher access latencies. We used CACTI [47] to show the access latency sensitivity study with larger L2 TLB capacities in Figure 4. The access latency is normalized to that of 16KB SRAM. As seen, naively increasing the SRAM capacity does not scale.

In the light of the above discussion, it would be desirable to have a TLB with a large reach at tolerable latency, so that a large majority of translations can be handled by TLBs rather than by Page Table Walkers (PTW). This paper presents a novel solution in this direction, the *POM-TLB*, a very large level-3 TLB that is Part of Memory. While TLBs are dedicated structures and not ordinarily addressable, the *POM-TLB* is mapped into the memory address space, and is stored in the DRAM (or any other emerging memory technology such as die-stacked DRAM). It is large enough to house translation entries for significantly huge working sets.

By making the *POM-TLB* as part of memory, it becomes possible to automatically take advantage of the growing L2 and L3 data cache capacities. While data caches already cache page table entries, multiple page table entries will be required per each translation, whereas a single *POM-TLB* entry will be sufficient to accomplish the virtualized translation. Hence caching TLB entries is more effective and beneficial than caching page table entries.

This paper makes the following contributions:

- We present a characterization of the virtual memory overheads on state-of-the-art hardware in several SPEC, PARSEC and graph workloads, while executing in bare metal and virtualized environments.
- We demonstrate that slow memory structures like DRAM can be used to house a large capacity TLB that can hold nearly all required address translations. On average, the proposed *POM-TLB* can achieve 10% performance improvement over a baseline system. For 5 of the benchmarks, the speedup is 16% or higher.
- We present a mechanism that makes it possible to cache TLB entries (not page table entries) into data caches. To the best of our knowledge, no prior work proposed caching of TLB entries into general (non-dedicated) caching structures.

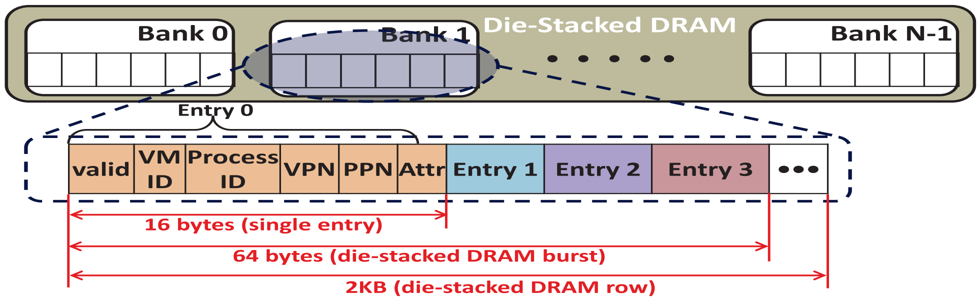

Figure 5: Metadata Format for POM-TLB

- We present several solutions to the challenges encountered while implementing a TLB in DRAM. We present a low overhead TLB location predictor and other enhancements to make a DRAM-based L3 TLB a feasible option.

The rest of this paper is organized as follows: Section 2 gives a detailed architectural description of *POM-TLB*; Section 3 describes our experimental setup; Section 4 presents our evaluation results; Section 5 discusses additional design benefits of our work; finally, Section 7 concludes the paper.

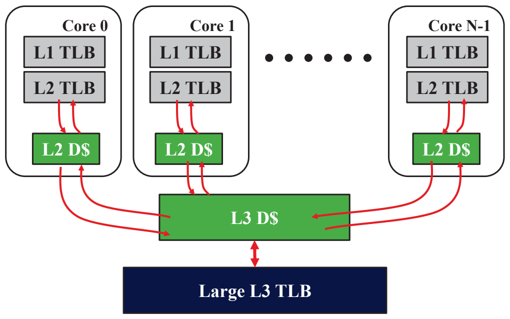

Figure 6: POM-TLB Overview

2 POM-TLB: A VERY LARGE L3 TLB

In this section, we describe the overall operation of *POM-TLB*, our very large L3 TLB, which can be implemented in off-chip memory or die-stacked DRAMs. Implementing in emerging die-stack DRAMs gives some advantages, although conceptually that is not a requirement.

2.1 System Level Organization

Most modern processors have private multi-level TLBs. In this work, we assume the TLB organization in a system similar to Intel Skylake architecture [23] where there are two levels of TLBs. Figure 6 shows an overview of *POM-TLB*. We add a large shared L3 TLB after the private L2 TLBs. An L2 TLB miss looks up the large shared TLB and initiate a page walk if this shared TLB also suffered a miss. In practice, since DRAM look-up is slow, we make our *POM-TLB addressable* thereby enabling caching of TLB entries in data caches and faster translations. Cached *POM-TLB* entries are stored in L2 and L3 data caches, so an L2 TLB miss looks up data caches before accessing the large L3 TLB. Only when the translation request misses in L3 TLB, then a page walk is initiated. In reality, since *POM-TLB* is so large that virtually almost all page walks are eliminated. This is discussed in detail in Section 2.1.3.

2.1.1 POM-TLB *Organization.*

While conceptually not a requirement, implementing *POM-TLB* in emerging die-stacked DRAMs integrated onto the processor gives bandwidth and possibly small latency advantages. Die-stacked DRAM has a DRAM-like organization, typically with access granularities such as 64B.

Figure 5 shows the metadata format for *POM-TLB* in a single channel of a typical die-stacked DRAM with multiple banks. We show the detailed layout of a single row in a bank. Each row can house multiple TLB entries as the row size is 2KB. Each entry has a valid bit, process ID, Virtual Address (VA), and Physical Address (PA) as in on-chip TLBs. To facilitate the translation in virtualized platforms, we also have Virtual Machine (VM) ID to distinguish addresses coming from different virtual machines as in Intel's Virtual Process ID (VPID) [22]. The attributes include information such as replacement and protection bits. Each entry is 16B and four entries make 64B. We implement our TLB as a four way associative structure since 1) we have found that the associativity lower than four invokes significantly higher conflict misses and 2) 64B is the common die-stacked DRAM burst length where no memory controller design modifications are necessary. Upon a request, four entries are fetched from a single die-stacked DRAM row. Each row can incorporate 128 TLB entries and with 4 way associativity, a row can hold 32 sets of TLB entries. As we will discuss in Section 4.4, due to spatio-temporal locality, the accesses to L3 TLB entries exhibit a very high row-buffer hit rate resulting in low average access times.

2.1.2 *Support for Two Page Sizes.*

In order to support both small page (4KB) and large page (2MB) TLB entries, and to avoid complexity in addressing a single TLB structure with two page sizes, we simply partitioned TLBs into two, one dedicated to hold 4KB page entries (denoted POM_TLB_{Small}) and the other 2MB page entries (denoted POM_TLB_{Large})[1]. In our implementation, their sizes are statically set and remain fixed. As they are DRAM-based and can afford large capacities, we observed that their exact sizes do not matter much.

[1]Unified designs with more complex addressing schemes such as skew-associativity [42] could be explored; we leave this for future work.

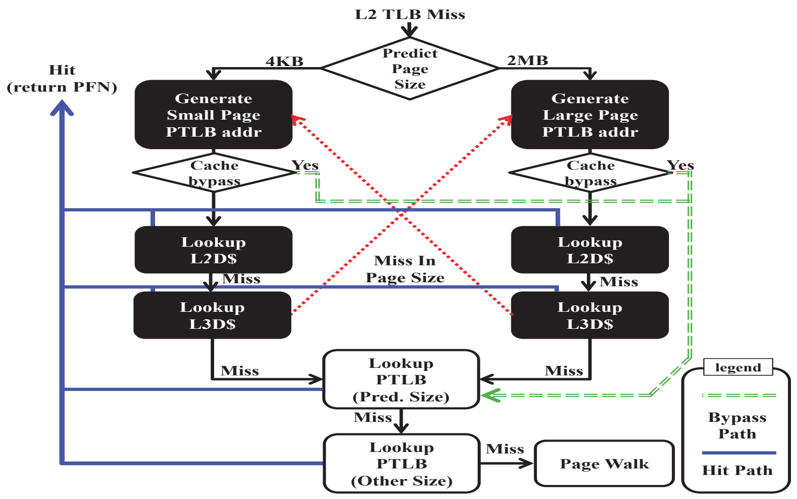

Figure 7: POM-TLB Access Flow Chart

We use a page size predictor (described in Section 2.1.4) to minimize having to perform two DRAM look-ups for the two page sizes. Based on the predicted page size, the corresponding TLB is accessed first. If it is a miss, the other TLB is accessed next. As discussed in Section 4.3, the page size predictor is highly accurate thereby almost always requiring just a single DRAM access.

2.1.3 Caching TLB Entries. While L1 and L2 TLBs are designed for fast look-up, the *POM-TLB* is designed for very large reach and consequently its DRAM-based implementation incurs higher access latency. In order to alleviate this, we map the *POM-TLB* into the physical address space. By making the TLB addressable, we achieve the important benefit of enabling the *caching* of TLB entries in data caches. Both POM_TLB_{Small} and POM_TLB_{Large} are assigned address ranges. A *POM-TLB* comprising N sets is assigned an address range of $64 \times N$ bytes as each set holds four 16-byte TLB entries. The virtual address (VA) of the L2 TLB miss is converted to a *POM-TLB* set index by extracting $log_2 N$ bits of the VA (after XOR-ing them with the VM ID bits to distribute the set-mapping evenly). For the POM_TLB_{Small}, the memory address of the set that the VA maps to is given by:

$$\begin{aligned} \mathrm{Addr}_{\mathrm{POM\_TLB}}\left(VA\right) = & \\ & \left(\left(VA \oplus VM\_ID\right) >> 6\right) \,\& \\ & \left(\left(1 << log_2\left(N\right)\right) - 1\right) * 64 + \\ & \mathrm{Base\_Addr}_{\mathrm{POM\_TLB}} \end{aligned} \tag{1}$$

where $Base\_Addr_{POM\_TLB}$ is the starting address of the POM_TLB. $Base\_POM\_TLB$ addresses are different depending on the page size since *POM-TLB* is physically partitioned between large and small pages.

In our scheme, L2 TLB misses do not initiate page walks. Instead, for each L2 TLB miss, the MMU computes the *POM-TLB* (say POM_TLB_{Large}) set address where the TLB entry for the virtual address of the miss may be found. The MMU then issues a load request to the L2D$ with this address. At this point, this becomes a normal cache access. If the entry is found in the L2D$, then the MMU reads the L2D$ cache block (64B) to access all the 4 translation entries stored in it. It performs associative search of the 4 entries to find a match for the incoming virtual address. If a match is found, then the corresponding entry provides the translation for this address. Being a normal read access, if the L2D$ does not contain the POM_TLB_{Large} address, then the request is issued to the L3D$. If no match was found in the L3D$, then the physical memory (in this case a POM_TLB_{Large} location) is accessed. Associative search of the set stored in the POM_TLB_{Large} is used to identify if a translation of the virtual address is present or not. Like data misses, TLB entries that are misses in data caches are filled into the caches after resolving them at the *POM-TLB* or via page walks.

Since the *POM-TLB* provides two potential set locations where the translation for a given VA may be found (POM_TLB_{Small} and POM_TLB_{Large}), we would have to perform two cache look-ups starting with the L2D$. Assuming an equal number of accesses to 4KB and 2MB pages, this results in 50% additional TLB look-up accesses into the L2D$. This has both latency and power implications. In order to address this, we design a simple yet highly accurate *Page Size Predictor* whose implementation is described next.

2.1.4 Page Size Prediction. We implement a simple yet highly effective page size predictor. The predictor comprises 512 2-bit entries, with one of the bits used to predict the page size and the other bit used to predict whether to bypass the caches (see next Section). The predictor is indexed using 9 bits of the virtual address of the L2 TLB miss (ignoring the lower order 12 bits). If the predicted page size is incorrect (0 means 4KB, 1 means 2MB), then the prediction entry for the index is updated [2]. While consuming very little SRAM storage (128 bytes per core), it achieves very high accuracy as discussed in Section 4.3.

[2]One could improve accuracy by adding hysteresis via a multi-bit saturating predictor or by using a larger predictor table.

2.1.5 Cache Bypass Prediction. In workloads where the data load/store access rate to the data caches far exceeds the rate of L2 TLB misses, the caches tend to contain very few *POM-TLB* entries since they get evicted to make room to fill in data misses. In such a scenario, looking up the data caches before reaching the *POM-TLB* is wasteful in terms of both power and latency. Thus we incorporate a 1-bit bypass predictor to bypass the caches. The predictor implementation is shared with the page size predictor described above. As shown in Figure 7, the cache bypass prediction takes effects after the *POM-TLB* address is generated since this new address is needed to access *POM-TLB*. Once the bypass decision is made, the request directly accesses *POM-TLB* by saving tens of lookup cycles in data caches.

2.1.6 Putting It All Together. Our scheme relies on making the large TLB addressable to achieve both a high hit rate in data caches as well as lower the access latency. Figure 7 shows an overall flow of *POM-TLB*. L2 TLB misses start out by consulting the page size predictor. If the predictor indicates a cache bypass, then the MMU directly accesses the predicted *POM-TLB* depending on the predicted page size. If a translation entry is found, the PFN (Physical Page Frame) is returned. If it is predicted not bypass, the MMU checks the *POM-TLB* entries in data caches. In the common case, the predictor does not predict cache-bypassing. The L2D$ is first probed with the address of the predicted *POM-TLB* set location. If a match is found with correct VA, VM_ID, then the translation is done with a single cache access by returning the corresponding PFN. If no match is found, then the MMU probes the L3D$ similarly. If the probed *POM-TLB* address misses in both L2D$ and L3D$, then the MMU computes a new *POM-TLB* address (corresponding to the size that was not predicted) and initiates cache look-up. If the new *POM-TLB* address misses in both levels of data caches, then a page walk is initiated. In practice, we observed that a vast majority of *POM-TLB* entries hit in the L2D$ and L3D$ resulting in very few DRAM accesses. As discussed in Section 4, the caching of *POM-TLB* entries in data caches causes very little degradation to the data cache hit rate for normal load/store accesses.

2.2 Implementation Considerations

We explain the key design decisions of the *POM-TLB* here:

Consistency: Since *POM-TLB* is shared across cores, the consistency requirement between entries in L3 and underlying L1/L2 TLB has to be met. Although strictly inclusive L3 TLB is desirable, it adds significant hardware complexity. Since our TLB operates at DRAM latency, which is already much slower than on-chip SRAM TLBs, adding such structure is not a practical option. Similar to prior work [9], we adopt the mostly inclusive implementation, which is adopted in x86 caches [20]. In this design, each TLB can make independent replacement decisions, which makes it possible that some entries in L1/L2 TLBs are missing from L3 TLB. However, this significantly reduces the hardware overheads associated with keeping strictly inclusive. Therefore, our TLB is designed to be aware of TLB-shootdowns. TLB-shootdowns require that all corresponding TLBs are locked until the consistency issue is resolved. Yet, TLB-shootdowns are rare occurrences and recent work [35] has shown a shootdown mechanism that can significantly reduce the overheads. Thus, the benefits of having simpler consistency check hardware outweigh the shootdown overheads, and hence such a design can be adopted.

In addition, the consistency across different virtual machines is already handled by modern virtual machine managers such as KVM hypervisor [2]. Upon a change in TLB, a memory notifier is called to let the host system know that a guest TLB has been updated. Then, the host OS invalidates all related TLBs in other VMs. Therefore, issues such as dirty page handling, process ID recycling, etc are already incorporated in KVM and host OS. The recent adoption of VM ID facilitates this process, and thus, *POM-TLB* can maintain consistency in the presence of multiple virtual machines. Although not all virtual machine managers have such feature, since such feature is implemented in software, future virtual machines can incorporate it.

Channel Contention: Memory systems share a common command/data bus to exchange data between controllers and multiple banks. Many of today's applications experience memory contention as the bandwidth is either saturated or near saturation [19]. Implementing the L3 TLB in an integrated die-stacked DRAM offers advantages from this perspective. Our proposal adds additional traffic only to the integrated DRAM to retrieve translation entries and not to the off-chip DRAM. Also this additional traffic is minor and only incurred when the L2D$ and L3D$ return cache misses when probed for cached *VL_TLB* entries. The path from last level caches to die-stacked DRAM architecture is different from one to off-chip DRAM as it has its own dedicated high-speed bus to communicate with processors. Hence, additional traffic due to our L3 TLB does not interfere with existing main memory traffic. In fact, our TLB's high hit rate reduces a significant amount of page table walks that result in main memory accesses, so it is likely that the main memory traffic sees considerable performance benefits as well.

Entry Replacement: Since our structure is four way associative, the attribute metadata (annoted as *attr* in Figure 5) contains 2 LRU bits. These bits are updated upon each L3 TLB access and the appropriate eviction candidate is chosen using these bits. Since LRU bits of four entries are fetched in a DRAM burst, the replacement decision can be made without incurring additional die-stacked DRAM accesses.

Other Die-Stacked DRAM Use: Die-stacked DRAM capacity is growing to multi-gigabytes, and in our experiments, *POM-TLB* achieves good performance at capacities like 32MB. The remaining die-stacked DRAM capacity can be used as a large last level data cache or a part of memory as proposed by prior work [11, 13–15, 18, 21, 25–27, 31–33, 39, 43, 44, 50]. Since JEDEC standard incorporates multiple channels in the HBM specification [24], we assume we are using one dedicated channel to service the *POM-TLB* requests. When the large die-stacked DRAM is used as both a large TLB and a large last level cache, the performance improvement will be even higher than results shown in this work, which only presents performance improvement from address translation.

Assuming 16MB capacity of *POM-TLB*, there can be a tradeoff between using this additional capacity as L4 data cache vs L3 TLB. In a cache design, a hit saves one memory access. However, in the case of an L3 TLB, especially in virtualized environment, the L3 TLB hit can save up to 24 accesses. This significantly reduces

Processor	Values
Frequency	4 GHz
L1 I- Cache	32KB, 8 way, 4 cycles
L1 D-Cache	32KB, 8 way, 4 cycles
L2 Unified Cache	256KB, 4 way, 12 cycles
L3 Unified Cache	8MB, 16 way, 42 cycles
MMU	Values
L1 TLB (4KB)	64 entries
	9 cycle miss penalty
L1 TLB (2MB)	32 entries
	9 cycle miss penalty
	L1 TLBs 4 way associative
L2 Unified TLB	1536 entries
	17 cycle miss penalty
	L2 TLBs 12 way associative
PSC	Values
PML4	2 entries, 2 cycle
PDP	4 entries, 2 cycle
PDE	32 entries, 2 cycle
Die-Stacked DRAM	Values
Bus Frequency	1 GHz (DDR 2 GHz)
Bus Width	128 bits
Row Buffer Size	2KB
tCAS-tRCD-tRP	11-11-11
DDR	Values
Type	DDR4-2133
Bus Frequency	1066 MHz
	(DDR 2133 MHz)
Bus Width	64 bits
Row Buffer Size	2KB
tCAS-tRCD-tRP	14-14-14

Table 1: Experimental Parameters

the total number of overall memory accesses. Furthermore, data accesses are non-blocking accesses where multiple requests can be on the fly. The access latency can be hidden by means of memory level parallelism such as bank level parallelism, which is common in today's DRAM. On the other hand, an address translation is a critical blocking request where upon a TLB miss, the processor execution stalls. Therefore, the impact of serving the translation request is much higher. Consequently, using the same capacity as a large TLB is likely to save more cycles than using it as L4 data cache. Note that 16MB is a small fraction of a die-stacked DRAM, and as previously mentioned, the rest of die-stacked DRAM can be used as a large data cache via separate channel without translation traffic contention.

3 EXPERIMENTAL EVALUATION

We evaluate the performance of *POM-TLB* using a combination of real system measurement, PIN-based and Ramulator-like [30] simulation, and performance models. Our virtualization platform is QEMU 2.0 with KVM support. Our host system is Ubuntu 14.04 running on Intel Skylake [23] with Transparent Huge Pages (THP) [3] turned on. Our host system has Intel VT-x with support for Extended Page Tables while the guest OS is Ubuntu 14.04 installed on QEMU also with THP turned on. The host system parameters are listed in Table 1 under Processor, MMU, and PSC categories. The system has separate L1 TLBs for each page size (4KB, 2MB, and 1GB in our system) though our applications do not use 1GB size. The L2 TLB is a unified TLB for both 4KB and 2MB pages. Finally, the specific performance counters (e.g., 0x0108, 0x1008, 0x0149, 0x1049) that we used read page walk cycles taking MMU cache hits into account, so the page walk cycles we use in this paper are the average cycles spent after a translation request misses in L2 TLB.

3.1 Workloads

The main focus of this work is on memory subsystems, and thus, applications, which do not spend a considerable amount of time in memory, are not meaningful. Consequently, we chose a subset of SPEC CPU and PARSEC applications that are known to be memory intensive. In addition, we also ran graph workloads such as the *graph500* and big data workloads such as *connected components* and *pagerank*. The benchmark characteristics are collected from the Intel Skylake platform, and they are presented in Table 2. We included applications whose page walk cycles, walk overheads, etc are in a wide range of spectrum (low to high). Since SPEC CPU applications are single threaded, we run multiple copies of SPEC CPU applications (as in the *SPECrate* mode), to evaluate performance on our multicore simulator. We ensure that they do not share the physical memory space via proper virtual-to-physical address translation. For multithreaded workloads, we profiled benchmarks with 8 threads.

3.2 Evaluation Methodology

A combination of measurement (on real hardware), simulation and performance modeling is used to estimate the performance of the proposed scheme. First, the workloads listed in Table 2 are executed to completion and the Linux *perf* utility is used to measure the total instructions (I_{total}), cycles (C_{total}), number of L2 TLB misses (M_{total}) and total L2 TLB miss penalty cycles (P_{total}) in a manner similar to the methodology in prior work [9, 10, 17, 48]. We obtain the baseline IPC as: $IPC_{baseline} = I_{total} C_{total}$. We also compute the ideal cycles C_{ideal} and average translation penalty cycles per L2 TLB miss $P_{Avg}^{Baseline}$ as:

$$C_{ideal} = C_{total} - P_{total} \qquad (2)$$

$$P_{Avg}^{Baseline} = P_{total} / M_{total} \qquad (3)$$

Note that the effects of various caching techniques like page walk caches, caching of PTEs in data caches, Intel extended page tables and nested TLBs are already included in the performance measurement since they are part of the base commodity processor. The average translation costs per L2 TLB miss as computed above are also listed in the workloads table.

Next, we use PIN and the Linux pagemap to generate memory traces for our workloads. For each workload, all load and store requests are recorded. The Linux pagemap is used to extend the PIN tool to include page size and other OS related metadata. Our trace contains virtual address, instruction count, read/write flag, thread ID and page size information of each reference. Memory instructions are traced in detail while the non-memory instructions are abstracted. We collected the memory traces for 20 billion instructions.

	astar	bwaves	canneal	ccomponent	gcc
Overhead Native (%)	13.89	0.73	3.19	0.73	0.30
Overhead Virtual (%)	16.08	7.70	6.34	7.40	12.12
Average Cycles-per-L2TLB-miss Native	98	128	53	44	46
Average Cycles-per-L2TLB-miss Virtual	114	151	61	1158	88
Frac Large Pages (%)	41.7	0.8	16.0	50.0	29.0
	GemsFDTD	graph500	gups	lbm	libquantum
Overhead Native (%)	10.58	1.03	12.20	0.05	0.02
Overhead Virtual (%)	16.01	7.66	17.20	12.02	7.37
Average Cycles-per-L2TLB-miss Native	129	79	43	110	70
Average Cycles-per-L2TLB-miss Virtual	133	80	70	290	75
Frac Large Pages (%)	71.0	7.0	2.59	57.4	32.9
	mcf	pagerank	soplex	streamcluster	zeusmp
Overhead Native (%)	10.32	4.07	4.16	0.07	0.01
Overhead Virtual (%)	19.01	6.96	17.07	2.11	10.22
Average Cycles-per-L2TLB-miss Native	66	51	144	74	136
Average Cycles-per-L2TLB-miss Virtual	169	61	145	76	137
Frac Large Pages (%)	60.7	60.0	12.3	87.2	72.1

Table 2: Benchmark Characteristics Related to TLB misses

Furthermore, we developed a detailed memory hierarchy simulator that simulates two levels of private TLBs, two levels of private data caches, a 3rd level shared data cache, and finally, the proposed 3rd level shared memory-based TLB. The simulator also models the size predictor and the cache bypass predictor, which are indexed using memory addresses. The simulator executes memory references from multiple traces while we schedule them at the proper issue cadence by using their instruction order in a manner similar to Ramulator. Information on the number of instructions in between the memory instructions are captured in the traces and thus memory level parallelism and overlap/lack of overlap between memory instructions are simulated. Note that our simulator is simulating both address translation traffic as well as data request traffic that go into underlying data caches. Finally, our simulator reports the L2 TLB miss cycles and detailed statistics such as hits and misses in the L1, L2 TLBs, data caches, the *POM-TLB* and predictor performance.

3.3 Performance Simulation of *POM-TLB*

The detailed cache and memory system parameters used in the simulation are listed in Table 1. DRAM simulation accounts for access latencies resulting from row-buffer hits and misses. It may also be noted that, since our baseline performance (obtained from real system measurements) already includes the benefits of hardware structures such as large pages, EPT and Page Structure Caches, we do not model these in our simulator and instead, use the baseline ideal cycles together with the estimated cost incurred by the *POM-TLB* when the DRAM-based TLB incurs a miss.

Total cycles taken by the *POM-TLB* and the resulting IPC for each core are obtained as:

$$C_{total}^{POM\_TLB} = C_{ideal} + M_{total} * P_{Avg}^{POM\_TLB} \quad (4)$$

$$IPC_{POM\_TLB} = I_{total} / C_{total}^{POM\_TLB} \quad (5)$$

$P_{Avg}^{POM\_TLB}$ denotes the average L2 TLB miss cycles in *POM-TLB* obtained from simulation. Having obtained the baseline and *POM-TLB* IPCs for each core, we obtain the overall performance improvement of the *POM-TLB*. It may be observed that we use the linear additive formula to add the L2 TLB miss cycles to the ideal cycles. This linear performance model ignores potential overlap of TLB processing cycles with execution cycles but is similar to models used in previous research [9, 10, 17, 48]. Such effects not only exist in *POM-TLB* but also in the baseline, so the performance impact of this exists equally in all schemes.

In addition, we compare *POM-TLB* against another prior work, which we annotate as *Shared_L2* in the rest of this paper. We implemented this scheme similar to [9] to the best of our knowledge. *Shared_L2* combines private SRAM based L2 TLBs into a single shared TLB, so when a request misses in L1 TLB, the large SRAM based shared TLB is looked up.

Finally, we implemented Translation Storage Buffer (TSB) that exists in SPARC processors. Since TLB misses are handled by OS in SPARC processors, TSB is managed by OS although indexing and address calculation is done by dedicated hardware. Unlike x86 architecture, upon a TLB miss in the SPARC architecture, an OS trap is called and appropriate TSB lookup or a software page table walk is initiated. TSB can be considered as a large MMU cache implemented in a large software-allocated buffer. We compare *POM-TLB* against such a TSB design to show the benefits of *POM-TLB*.

4 RESULTS

This section presents performance improvement for *POM-TLB* in comparison to other related schemes proposed in prior research for multicore systems [9, 10] as well as a current existing system feature that is most closely similar to ours, SPARC's TSB. The results presented in this section are on top of a baseline with dedicated page structure caches as in the Intel Skylake processors.

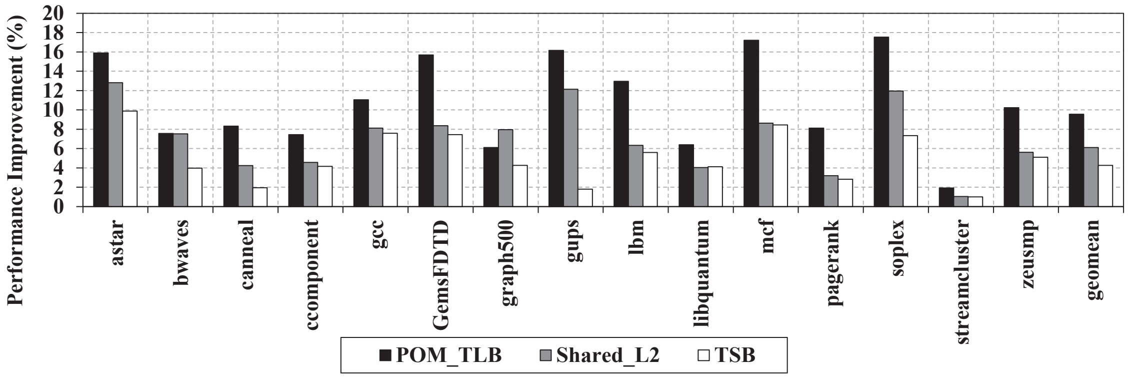

Figure 8: Performance Improvement of *POM-TLB* (8 Core)

4.1 Performance Improvement

Figure 8 plots the performance improvements of *POM-TLB* on 8 core configuration with 16MB *POM-TLB* size. Here, the baseline is the execution time gathered from our experimental runs on SkyLake processors. Note that the improvement is shown in percentage (%) and 2 different comparables are presented in addition to *POM-TLB*.

The improvement ranges from 1% in streamcluster to 17% in soplex. We observe that workloads with high page walk overheads in virtualized platforms (see Table 2) have the highest improvement (such as mcf, soplex, GemsFDTD, astar and gups), which indicates that *POM-TLB* is effective in reducing costly page walks. The streamcluster benchmark does not contain significant page walk overhead to begin with (2.11%). Therefore, this benchmark does not possess a lot of headroom for improvement from *POM-TLB*. On average, *POM-TLB* is able to achieve a performance improvement of 9.57%. It may also be noted that these performance gains are obtained on top of the use of large pages. Even where a large fraction of pages are 2MB pages (for example, mcf has 70% and astar has 40% large pages in Table 2), the workloads exhibit considerable performance improvements.

Shared_L2 is able to achieve 6.10% performance improvement on average. This scheme does benefit from sharing of the combined L2 TLB capacities, yet it is still limited in terms of capturing the hot set of TLB entries. Thus, it encounters high shared L2 TLB miss penalty. On the other hand, *POM-TLB* is able to capture much more TLB entries. First, the physical capacity of our design is 16MB, which is a few orders of magnitude higher than TLBs in existing systems or L2 TLB capacity in *Shared_L2* design. Upon a TLB miss in *Shared_L2*, the page walk is initiated when data caches are also searched, since intermediate PTEs are stored in data caches. However, in order to get a complete translation, many of these intermediate entries must be searched until the last level PTE is found. Only then, the translation is done. Even though the access is done in SRAM latency in this case, multiple accesses have to be made in order to complete the page walk. MMU caches, such as PSC, help to reduce the number of such intermediate entry accesses, yet their capacity is very limited, so it only caches a small amount of TLB misses. In addition, even though *POM-TLB* is located in die-stacked DRAM, which incurs an access latency similar to DRAM, we cache many of TLB entries in data caches. This enables us to achieve much lower access latency, as many of these entries are cached in data caches. Also, the use of L2D$ and L3D$ allow us to have a lot of TLB entries stored in caches. An additional advantage is that single virtual address only requires single entry in data caches whereas *Shared_L2* has multiple intermediate PTEs stored in caches, which consumes much more capacity in data caches.

TSB achieves an average performance improvement of 4.27% across our workloads. This is surprising considering that it uses 16MB capacity as in *POM-TLB*. However, the performance of this scheme is limited as each TLB miss incurs a trap operation, which is required in the operation of TSB as it is software managed. Also, unlike *POM-TLB* which has an associativity of 4, TSB is a direct mapped organization, so it sees more conflict misses. *POM-TLB* uses the 64B cacheline size, so each cacheline has 4 TLB entries. Since the data transfer granularity between die-stacked DRAM and on-chip caches is done at 64B, we exploit this and allow *POM-TLB* to have an associativity of 4. Furthermore, TSB entries are not direct guest-VA to host-PA translations, and require multiple accesses to the TSB to complete the virtual-to-physical translation thereby incurring higher access latency. An interesting observation is made for the gups benchmark. This benchmark is known to have low locality in page tables, so an ability to achieve high performance for

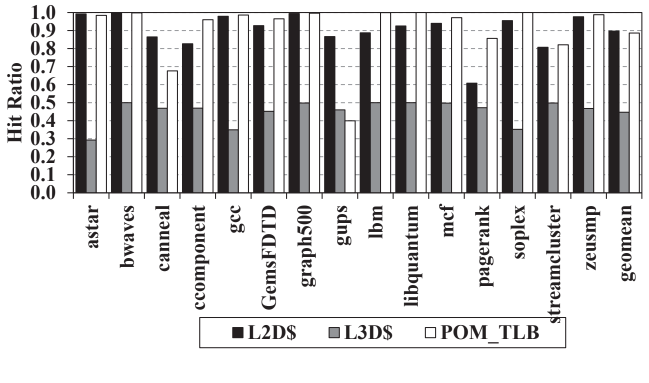

Figure 9: Hit Ratio of *POM-TLB* (8 Core)

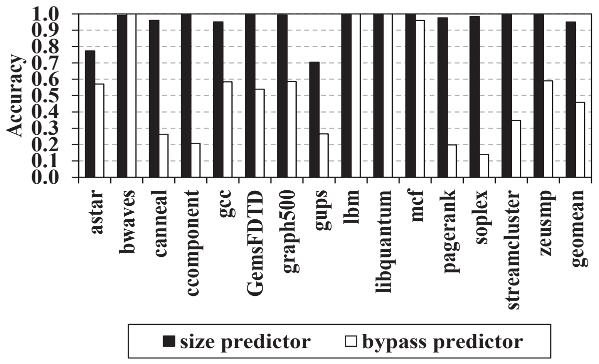

Figure 10: Predictor Accuracy (8 core)

such low spatial locality workloads can show how well each scheme retains translation entries. In case of TSB, it is not able to capture many of these entries even with 16MB as it only achieves 1.80% improvement. However, *POM-TLB* achieves performance improvement of 16%, approximately an order of difference in performance. Therefore, we can see that *POM-TLB* makes much better use of the 16MB space that is located in die-stacked DRAM.

4.2 Hit Ratio

The effectiveness of *POM-TLB* can be shown using the hit ratio. Here, we analyze how well *POM-TLB* can capture L2 TLB misses, thereby reducing costly page walks. Figure 9 shows the hit ratio perceived at different level of the memory subsystem where TLB entries are stored by our scheme. First, the L2D$ has a very high hit rate of 89.7% on average. Since L2 data caches are private, it is not affected by interference from other cores. Since the L2 capacity is much larger than other private TLB structures in the processor, it keeps a lot of translation traffic from performing page walks. Note that caching of TLB entries is only feasible since we have a very large TLB and that is addressable. In conventional systems, the TLB entries are not visible as they are entirely managed by MMUs, yet our novel idea of making the large TLB addressable has enabled a larger number of TLB requests to be cached in rather large on-chip caches.

When a request misses in L2D$, then it is looked up in shared L3D$. The hit ratio here is not as good as L2D$. First, it is a shared data structure, so interference starts degrading performance. Also, a majority of TLB requests are filtered by L2D$, so only requests with a low degree of locality are passed down to the L3D$. However, *POM-TLB* in die-stacked DRAM again picks up a lot of these requests as shown by a higher hit ratio of 88% on average. *POM-TLB* can achieve this as the capacity is rather large, so it can recapture many translation requests that missed in a smaller L3D$. It may also be noted that the data caches are also caching the normal data accesses made by the cores and are not being used solely for TLB entries.

4.3 Predictor Accuracy

In our scheme, we implemented two predictors, which are size and bypass predictors. The size predictor speculates whether the incoming translation address is going to be a request for a large or small pages. Although there are proposals [42] that enable simultaneous accesses to TLB structures to check both small and large pages, we avoid doing this as it requires sophisticated design/verification efforts as well as consumes more power. We rather used a simple predictor, but as seen in Figure 10, the size predictor is highly accurate as it achieves an average accuracy of 95%. The accuracy is calculated by dividing the total number of correct speculations by the total number of speculations. In such cases, we can reduce 95% of the second TLB accesses to look for the TLB entry of the other size. Yet, in comparison to performing a serialized access, our predictor can fetch the correct TLB entry in a single *POM-TLB* access.

Our implementation adds a miss penalty if translations miss in data caches as additional on-chip cache lookups are performed prior to accessing *POM-TLB*. The bypass predictor effectively eliminates such latency and forwards the request directly to *POM-TLB* upon L2 TLB miss. Our predictor is achieving a low accuracy of 45.8% on average. Although some workloads such as bwaves, lbm and libquantum are able to achieve close to perfect accuracy, others such as soplex and pagerank have a low hit rate. Although the data cache access latencies are an order of magnitude lower than page walk cycles, the misprediction penalty keeps *POM-TLB* from achieving the best performance. We leave it to the future work to perform other approaches such as using the instruction address to increase the accuracy.

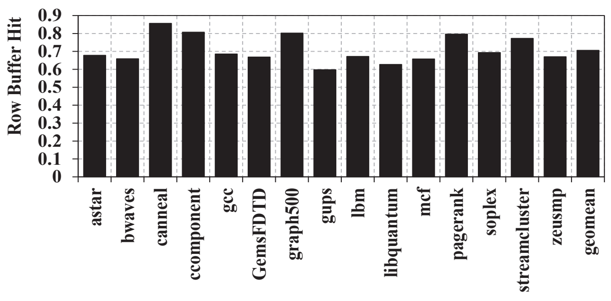

Figure 11: Row Buffer Hits in L3 TLB (8 core)

4.4 Row Buffer Hits (RBH) in the L3 TLB

We quantify the intuition that the spatial locality of TLB accesses leads to a high Row Buffer Hits (RBH) in the stacked DRAM. Figure 11 plots the RBH values. As reported, the stacked DRAM achieves a high average RBH of 71%, thereby ensuring a low latency L3 TLB lookup. Each row contains 128 TLB entries, which is similar in capacity as an on-chip L2 TLB. Since they are located in the same row, these TLB accesses are likely to hit in the row buffer. As expected, applications with high spatial locality show high RBH values. For example, streamcluster has streaming behaviors, which thus has high spatial locality [36], explaining its high RBH value in Figure 11.

4.5 *POM-TLB* without Data Caches

In this section, we quantify the performance benefits *POM-TLB* gets from storing TLB entries in data caches. Figure 12 shows the performance improvement when TLB entries are cached in data caches and when not cached. As shown, caching significantly helps

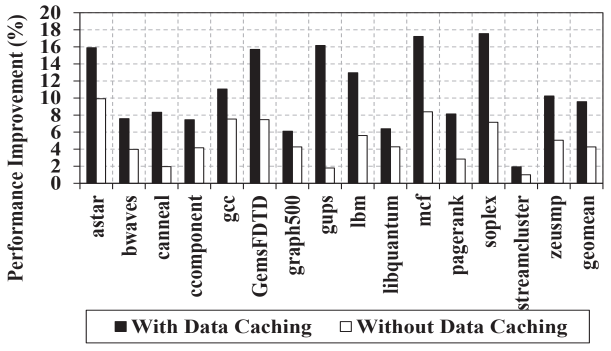

Figure 12: *POM-TLB* With and Without Data Caching (8 core)

performance as it provides an additional performance improvement of 5%. The performance aspect that caching helps is not in reducing the number of page walks. Whether data caches are used or not does not affect the number of page walks as this reduction is performed by the large capacity of *POM-TLB*. Instead, what caching enables is hiding the long latency of die-stacked DRAM accesses, bridging the latency gap between on-chip TLBs and die-stacked DRAM-based *POM-TLB*.

4.6 *POM-TLB* Discussion

We have experimented with different *POM-TLB* capacities. Although not presented as they do not provide meaningful data, we have found that varying *POM-TLB* capacity to either 8MB or 32MB changes our performance improvement less than 1%. We have found that it is extremely difficult to find workloads whose memory footprint exceed such large TLB sizes. In fact, 16MB capacity used in our default value is a scaled down version of die-stacked DRAM capacity to be a representative fraction of our workloads working set. In the market today, we see die-stacked DRAM capacity reaching several gigabytes [34]. Therefore, if we scale up and use a fraction of the capacity of today's die-stacked DRAM, we expect that *POM-TLB* will be able to eliminate a significant amount of page walks for even emerging applications with vastly larger footprint than ones we see today. Note that the TLB reach of *POM-TLB* is orders of magnitude larger than today's on-chip TLBs even with only 16MB. Similarly, we have also varied the core count to 4 and 32 cores. Yet, the performance improvement stays approximately the same as again even with fewer cores or more cores, *POM-TLB* is so large that, most of the page walks are eliminated[3].

POM-TLB is the first of its kind that implemented a TSB-like structure in purely hardware. TSB acts like a partial translation cache for SPARC's expensive software page walkers. Although the fact that both use a very large capacity in DRAM is similar, our contribution is fundamentally different in that we implement it as hardware TLB, so that page walks are not even initiated. Once the page walk is initiated, the processor is stalled, whereas *POM-TLB* does not block any execution as almost all translation requests hit in *POM-TLB*.

[3]Per-core L2D$ continue to provide the bulk of the latency improvements in all core counts.

5 UNLOCKING ADDITIONAL BENEFITS

In this section, we present additional benefits unlocked by our addressable *POM-TLB* and briefly discuss those benefits.

5.1 TLB-Aware Caching

Since TLB entries are addressable and get cached in L2D$ and L3D$, it is possible to design cache allocation and replacement policies that can adaptively allocate appropriate cache capacity to hold TLB entries and normal data. In workloads where L2 TLB misses incur high penalty, the data caches could prioritize retaining *POM-TLB* entries, while in workloads with higher data misses as compared to L2 TLB misses, the caches could prioritize retaining data contents.

5.2 Efficient Virtual Machine Switching

Even though modern TLBs can simultaneously hold translations for multiple virtual machines (identified by VM ID), the major bottleneck is the small size of the SRAM-based TLBs. Running multiple VMs is common in today's large scale virtual platforms such as Amazon EC2 [1] and each VM contains a full OS, making the TLB reach again the problem. The large L3 TLB can alleviate this capacity issue by simultaneously retaining the address translations of multiple VMs. With the L3 TLB, when different VMs interfere with each other, the L3 TLB can provide a high hit rate that results in avoiding the vast majority of expensive page walks caused by L1, L2 TLB conflicts.

5.3 Reduction in Design Complexity

Modern processors employ an array of complex hardware techniques to accelerate page walks, including sophisticated page walk caches [4] and intra/inter-core TLB prefetchers [10]. These techniques result in significant design complexity to ensure TLB consistency and integrity of page table contents. With the introduction of the L3 TLB that offers very high hit rates at moderately low latencies, these structures could be simplified or even eliminated, resulting in an overall simpler hardware implementation.

6 RELATED WORK

Caching and speculation techniques have been proposed to improve the two dimensional address translation overheads in virtualized platforms [4, 5, 7, 16, 17, 38]. Caching schemes such as page walk cache [4, 4, 17] attempt to bypass the intermediate level walks. Speculation schemes [5, 38] let the processor execution continue with speculated page table entries and invalidate speculated instructions upon detecting misspeculation. These schemes are motivated by the fact that conventional TLBs are likely to cause more page table walks [7] for emerging big data workloads with large memory footprints. Therefore, they focus on reducing/hiding the overheads of page table walks. In our work, we address a more fundamental problem that very large TLBs can withstand increased address translation pressure from virtualization by offering a translation storage with high capacity and high bandwidth, thereby significantly reducing the number of walks.

Some other schemes [6, 17, 29] attempt to reduce the levels of page table walks in either native or virtualized system. However, our scheme tackles the fundamental problem of current TLB's insufficient capacities. Thus, by increasing the capacity significantly, we are solving the inherent structural bottleneck in today's system. However, our scheme is orthogonal to aforementioned schemes, as we do not alter the existing or proposed page walk hardware structures. Thus, our scheme can easily be augmented to these schemes. Moreover, TLB prefetching [10, 28, 41] improves the TLB hit rate by fetching entries ahead of time. These TLB prefetchers are also orthogonal to our scheme. Our *POM-TLB* augmented with a prefetcher can reduce the prefetching latency and achieve considerable performance improvement.

The Linux Transparent Huge Page [3] along with various schemes [8, 16, 29, 37, 45, 46, 49] try to increase the fraction of large pages either in hardware or by OS to reduce the number of TLB misses. Although we do not take advantage of such schemes, but rather use large pages created by existing OS, using their scheme can even further increase the data cache and *POM-TLB* hit rates as a single 2MB entry incorporates 512 4KB entries, thereby further increasing the already large the reach of *POM-TLB*.

7 CONCLUSIONS

In this work, we evaluated the feasibility of building very large level 3 TLBs in DRAMs. We have presented a TLB which is part of the memory space, thereby allowing the possibility of caching TLB entries in conventional L2 and L3 data caches. Our thorough analysis using a combination of measurements on state of the art Skylake processors, detailed simulation and an additive performance model show that the proposed *POM-TLB* can practically eliminate the page walk overhead in most memory intensive workloads, particularly multi-threaded workloads and virtualized environments. In addition, we have shown that the proposed *POM-TLB* provides higher performance improvement than previously proposed shared TLB or prefetching techniques at the L1/L2 TLB level. Simulation studies using SPEC, PARSEC and graph workloads demonstrate that more than 16% performance improvement can be obtained in a third of the experimented benchmarks (with an average of 10% over all benchmarks). In most configurations 99% of the page walks can be eliminated by a very large TLB of size 16 MB.

8 ACKNOWLEDGEMENT

This work is supported in part by the National Science Foundation under Grant 1337393 and the Texas Advanced Computing Center (TACC).

REFERENCES

[1] Amazon. 2006. Amazon EC2 - Virtual Server Hosting. (2006). https://aws.amazon.com/ec2/.

[2] Andrea Arcangeli. 2008. Linux KVM Forum. (2008). http://www.linux-kvm.org/page/KVM_Forum_2008.

[3] Andrea Arcangeli. 2010. Transparent hugepage support. In *KVM Forum*, Vol. 9. http://www.linux-kvm.org/page/KVM_Forum_2010.

[4] Thomas W. Barr, Alan L. Cox, and Scott Rixner. 2010. Translation Caching: Skip, Don't Walk (the Page Table). In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA '10)*. ACM, New York, NY, USA, 48–59. https://doi.org/10.1145/1815961.1815970

[5] Thomas W. Barr, Alan L. Cox, and Scott Rixner. 2011. SpecTLB: A Mechanism for Speculative Address Translation. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA '11)*. ACM, New York, NY, USA, 307–318. https://doi.org/10.1145/2000064.2000101

[6] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. 2013. Efficient Virtual Memory for Big Memory Servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM, New York, NY, USA, 237–248. https://doi.org/10.1145/2485922.2485943

[7] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. 2008. Accelerating Two-dimensional Page Walks for Virtualized Systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*. ACM, New York, NY, USA, 26–35. https://doi.org/10.1145/1346281.1346286

[8] Abhishek Bhattacharjee. 2013. Large-reach memory management unit caches: Coalesced and shared memory management unit caches to accelerate TLB miss handling. In *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 383–394.

[9] Abhishek Bhattacharjee, Daniel Lustig, and Margaret Martonosi. 2011. Shared last-level TLBs for chip multiprocessors. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*. 62–63. https://doi.org/10.1109/HPCA.2011.5749717

[10] Abhishek Bhattacharjee and Margaret Martonosi. 2010. Inter-core Cooperative TLB for Chip Multiprocessors. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*. ACM, New York, NY, USA, 359–370. https://doi.org/10.1145/1736020.1736060

[11] Bryan Black, Murali Annavaram, Ned Brekelbaum, John DeVale, Lei Jiang, Gabriel H. Loh, Don McCaule, Pat Morrow, Donald W. Nelson, Daniel Pantuso, Paul Reed, Jeff Rupley, Sadasivan Shankar, John Shen, and Clair Webb. 2006. Die Stacking (3D) Microarchitecture. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 39)*. IEEE Computer Society, Washington, DC, USA, 469–479. https://doi.org/10.1109/MICRO.2006.18

[12] Jeffrey Buell, Daniel Hecht, Jin Heo, Kalyan Saladi, and RH Taheri. 2013. Methodology for performance analysis of VMware vSphere under Tier-1 applications. *VMware Technical Journal* 2, 1 (2013).

[13] Chiachen Chou, Aamer Jaleel, and Moinuddin K. Qureshi. 2014. CAMEO: A Two-Level Memory Organization with Capacity of Main Memory and Flexibility of Hardware-Managed Cache. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 1–12. https://doi.org/10.1109/MICRO.2014.63

[14] Chiachen Chou, Aamer Jaleel, and Moinuddin K. Qureshi. 2015. BEAR: Techniques for mitigating bandwidth bloat in gigascale DRAM caches. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. 198–210. https://doi.org/10.1145/2749469.2750387

[15] Xiangyu Dong, Yuan Xie, Naveen Muralimanohar, and Norman P. Jouppi. 2010. Simple but Effective Heterogeneous Main Memory with On-Chip Memory Controller Support. In *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–11. https://doi.org/10.1109/SC.2010.50

[16] Jayneel Gandhi, Arkaprava Basu, Mark D. Hill, and Michael M. Swift. 2014. Efficient Memory Virtualization: Reducing Dimensionality of Nested Page Walks. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*. IEEE Computer Society, Washington, DC, USA, 178–189. https://doi.org/10.1109/MICRO.2014.37

[17] Jayneel Gandhi, Mark D. Hill, and Michael M. Swift. 2016. Agile Paging: Exceeding the Best of Nested and Shadow Paging. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 707–718. https://doi.org/10.1109/ISCA.2016.67

[18] Nagendra Dwarakanath Gulur, Mahesh Mehendale, R. Manikantan, and R. Govindarajan. 2014. Bi-Modal DRAM Cache: Improving Hit Rate, Hit Latency and Bandwidth.. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*. IEEE, 38–50. http://dblp.uni-trier.de/db/conf/micro/micro2014.html#GulurMMG14

[19] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. 2011. Toward Dark Silicon in Servers. *IEEE Micro* 31, 4 (July 2011), 6–15. https://doi.org/10.1109/MM.2011.77

[20] Glenn Hinton, Dave Sager, Mike Upton, Darrell Boggs, and others. 2001. The microarchitecture of the Pentium® 4 processor. In *Intel Technology Journal*. Citeseer.

[21] Cheng-Chieh Huang and Vijay Nagarajan. 2014. ATCache: Reducing DRAM Cache Latency via a Small SRAM Tag Cache. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT '14)*. ACM, New York, NY, USA, 51–60. https://doi.org/10.1145/2628071.2628089

[22] Intel. 2006. Intel(R) Virtualization Technology. (2006). http://www.intel.com/content/www/us/en/virtualization/virtualization-technology/intel-virtualization-technology.html.

[23] Intel. 2015. 6th Generation Intel Core i7-6700K and i5-6600K Processors. (2015). http://www.intel.com/content/www/us/en/processors/core/6th-gen-core-family-desktop-brief.html.

[24] JEDEC. 2016. High Bandwidth Memory (HBM) DRAM Gen 2 (JESD235A). https://www.jedec.org

[25] Djordje Jevdjic, Gabriel H. Loh, Cansu Kaynak, and Babak Falsafi. 2014. Unison Cache: A Scalable and Effective Die-Stacked DRAM Cache. In *47th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2014, Cambridge, United Kingdom, December 13-17, 2014*. 25–37. https://doi.org/10.1109/

MICRO.2014.51

[26] Djordje Jevdjic, Stavros Volos, and Babak Falsafi. 2013. Die-stacked DRAM Caches for Servers: Hit Ratio, Latency, or Bandwidth? Have It All with Footprint Cache. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM, New York, NY, USA, 404–415. https://doi.org/10.1145/2485922.2485957

[27] Xiaowei Jiang, Niti Madan, Li Zhao, Mike Upton, Ravi Iyer, Srihari Makineni, Don Newell, Yan Solihin, and Rajeev Balasubramonian. 2011. CHOP: Integrating DRAM Caches for CMP Server Platforms. *IEEE Micro* 31, 1 (Jan 2011), 99–108. https://doi.org/10.1109/MM.2010.100

[28] Gokul B. Kandiraju and Anand Sivasubramaniam. 2002. Going the distance for TLB prefetching: an application-driven study. In *Proceedings 29th Annual International Symposium on Computer Architecture*. 195–206. https://doi.org/10.1109/ISCA.2002.1003578

[29] Vasileios Karakostas, Jayneel Gandhi, Adrian Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman S. Unsal. 2016. Energy-efficient address translation. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 631–643.

[30] Yoongu Kim, Weikun Yang, and Onur Mutlu. 2016. Ramulator: A Fast and Extensible DRAM Simulator. *IEEE Comput. Archit. Lett.* 15, 1 (Jan. 2016), 45–49. https://doi.org/10.1109/LCA.2015.2414456

[31] Gabriel H. Loh and Mark D. Hill. 2011. Efficiently Enabling Conventional Block Sizes for Very Large Die-stacked DRAM Caches. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44)*. ACM, New York, NY, USA, 454–464. https://doi.org/10.1145/2155620.2155673

[32] Gabriel H. Loh, Nuwan Jayasena, Kevin Mcgrath, Mike O'Connor, Steve Reinhardt, and Jaewoong Chung. 2012. Challenges in Heterogeneous Die-Stacked and Off-Chip Memory Systems. In *the 3rd Workshop on SoCs, Heterogeneous Architectures and Workloads (SHAW)*.

[33] Mitesh R. Meswani, Serigey Blagodurov, David Roberts, John Slice, Mike Ignatowski, and Gabriel H. Loh. 2015. Heterogeneous memory architectures: A HW/SW approach for mixing die-stacked and off-package memories. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 126–136. https://doi.org/10.1109/HPCA.2015.7056027

[34] Micron. 2013. Micron's Hybrid Memory Cube Earns High Praise in Next-Generation Supercomputer. (2013). http://investors.micron.com/releasedetail.cfm?ReleaseID=805283.

[35] Mark Oskin and Gabriel H. Loh. 2015. A Software-Managed Approach to Die-Stacked DRAM. In *2015 International Conference on Parallel Architecture and Compilation, PACT 2015, San Francisco, CA, USA, October 18-21, 2015*. 188–200. https://doi.org/10.1109/PACT.2015.30

[36] Celal Ozturk. 2013. Analyzing and Quantifying Dynamc Program Behavior in Terms of Regularities and Patterns. In *Open Access Dissertations*. 63. http://digitalcommons.uri.edu/oa_diss/63

[37] Binh Pham, Abhishek Bhattacharjee, Yasuko Eckert, and Gabriel H. Loh. 2014. Increasing TLB reach by exploiting clustering in page translations. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 558–567. https://doi.org/10.1109/HPCA.2014.6835964

[38] Binh Pham, Ján Veselý, Gabriel H. Loh, and Abhishek Bhattacharjee. 2015. Large Pages and Lightweight Memory Management in Virtualized Environments: Can You Have It Both Ways?. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/2830772.2830773

[39] Moinuddin K. Qureshi and Gabriel H. Loh. 2012. Fundamental Latency Trade-off in Architecting DRAM Caches: Outperforming Impractical SRAM-Tags with a Simple and Practical Design.. In *MICRO*. IEEE Computer Society, 235–246. http://dblp.uni-trier.de/db/conf/micro/micro2012.html#QureshiL12

[40] Rackspace. 2012. OPENSTACK - The Open Alternative To Cloud Lock-In. (2012). https://www.rackspace.com/en-us/cloud/openstack.

[41] Ashley Saulsbury, Fredrik Dahlgren, and Per Stenström. 2000. Recency-based TLB Preloading. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA '00)*. ACM, New York, NY, USA, 117–127. https://doi.org/10.1145/339647.339666

[42] André Seznec. 2004. Concurrent Support of Multiple Page Sizes on a Skewed Associative TLB. *IEEE Trans. Computers* 53, 7 (2004), 924–927. https://doi.org/10.1109/TC.2004.21

[43] Jaewoong Sim, Alaa R. Alameldeen, Zeshan Chishti, Chris Wilkerson, and Hyesoon Kim. 2014. Transparent Hardware Management of Stacked DRAM as Part of Memory. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 13–24. https://doi.org/10.1109/MICRO.2014.56

[44] Jaewoong Sim, Gabriel H. Loh, Hyesoon Kim, Mike OConnor, and Mithuna Thottethodi. 2012. A Mostly-Clean DRAM Cache for Effective Hit Speculation and Self-Balancing Dispatch. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. 247–257. https://doi.org/10.1109/MICRO.2012.31

[45] Shekhar Srikantaiah and Mahmut Kandemir. 2010. Synergistic TLBs for High Performance Address Translation in Chip Multiprocessors. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. 313–324. https://doi.org/10.1109/MICRO.2010.26

[46] Madhusudhan Talluri, Shing Kong, Mark D. Hill, and David A. Patterson. 1992. Tradeoffs in Supporting Two Page Sizes. In *[1992] Proceedings the 19th Annual International Symposium on Computer Architecture*. 415–424. https://doi.org/10.1109/ISCA.1992.753337

[47] Steven J. E. Wilton and Norman P. Jouppi. 1996. CACTI: an enhanced cache access and cycle time model. *IEEE Journal of Solid-State Circuits* 31, 5 (May 1996), 677–688. https://doi.org/10.1109/4.509850

[48] Idan Yaniv and Dan Tsafrir. 2016. Hash, Don'T Cache (the Page Table). In *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science*. ACM, New York, NY, USA, 337–350. https://doi.org/10.1145/2896377.2901456

[49] Lixin Zhang, Evan Speight, Ram Rajamony, and Jiang Lin. 2010. Enigma: Architectural and Operating System Support for Reducing the Impact of Address Translation. In *Proceedings of the 24th ACM International Conference on Supercomputing (ICS '10)*. ACM, New York, NY, USA, 159–168. https://doi.org/10.1145/1810085.1810109

[50] Li Zhao, Ravi Iyer, Ramesh Illikkal, and Don Newell. 2007. Exploring DRAM cache architectures for CMP server platforms. In *2007 25th International Conference on Computer Design*. 55–62. https://doi.org/10.1109/ICCD.2007.4601880