CSALT: Context Switch Aware Large TLB *

Yashwant Marathe, Nagendra Gulur[1], Jee Ho Ryoo, Shuang Song, and Lizy K. John
Department of Electrical and Computer Engineering, University of Texas at Austin
[1]Texas Instruments
ymarathe@utexas.edu, nagendra@ti.com, {jr45842, songshuang1990}@utexas.edu, ljohn@ece.utexas.edu

ABSTRACT

Computing in virtualized environments has become a common practice for many businesses. Typically, hosting companies aim for lower operational costs by targeting high utilization of host machines maintaining just *enough* machines to meet the demand. In this scenario, frequent virtual machine context switches are common, resulting in increased TLB miss rates (often, by over 5*X* when contexts are doubled) and subsequent expensive page walks. Since each TLB miss in a virtual environment initiates a 2D page walk, the data caches get filled with a large fraction of page table entries (often, in excess of 50%) thereby evicting potentially more useful data contents.

In this work, we propose *CSALT* - a Context-Switch Aware Large TLB, to address the problem of increased TLB miss rates and their adverse impact on data caches. First, we demonstrate that the *CSALT* architecture can effectively cope with the demands of increased context switches by its capacity to store a very large number of TLB entries. Next, we show that *CSALT* mitigates data cache contention caused by conflicts between data and translation entries by employing a novel *TLB-Aware Cache Partitioning* scheme. On 8-core systems that switch between two virtual machine contexts executing multi-threaded workloads, *CSALT* achieves an average performance improvement of 85% over a baseline with conventional L1-L2 TLBs and 25% over a baseline which has a large L3 TLB.

CCS CONCEPTS

Computer systems organization → Heterogeneous (hybrid) systems;

MICRO-50, October 14-18, 2017, Cambridge, MA, USA

ACM 978-1-4503-4952-9/17/10...$15.00
https://doi.org/10.1145/3123939.3124549

KEYWORDS

Address Translation, Virtualization, Cache Partitioning

ACM Reference format:
Yashwant Marathe, Nagendra Gulur[1], Jee Ho Ryoo, Shuang Song, Lizy K. John. Department of Electrical and Computer Engineering, The University of Texas at Austin [1].Texas Instruments. 2017. CSALT: Context Switch Aware Large TLB. *In Proceedings of MICRO-50, Cambridge, MA, USA, October 14-18, 2017*, 12 pages.
https://doi.org/10.1145/3123939.3124549

1. INTRODUCTION

Computing in virtualized cloud environments [7, 23, 46, 61, 22] has become a common practice for many businesses as they can reduce capital expenditures by doing so. Many hosting companies have found that the utilization of their servers is low (see [39] for example).

In order to keep the machine utilization high, the hosting companies that maintain the host hardware typically attempt to keep just *enough* machines to serve the computing load, and allowing multiple virtual machines to coexist on same physical hardware [10, 64, 57]. High CPU utilization has been observed in many virtualized workloads [44, 45, 42].

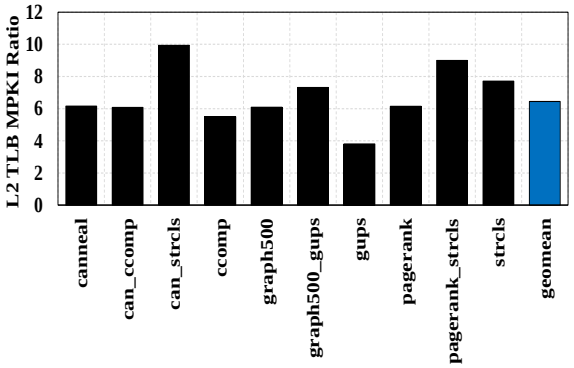


Figure 1: Increase in TLB Misses due to Context Switches. Ratio of L2 TLB MPKIs in Context Switch Case to Non-Context Switch Case

The aforementioned trend means that the host machines are constantly occupied by applications from different businesses, and frequently, different contexts are executed on the same

machine. Although it is ideal for achieving high utilization, the performance of guest applications suffer from frequent context switching. The memory subsystem has to maintain consistency across the different contexts, and hence traditionally, processors used to flush caches and TLBs. However, modern processors adopt a more efficient approach where each entry contains Address Space Identifier (ASID) [2]. Tagging the entry with ASID eliminates the needs to flush the TLB upon a context switch, and when the swapped-out context returns, some of its previously cached entries will be present. Although these optimizations worked well with traditional benchmarks where the working set, or memory footprint, was manageable between context switches, this trend no longer holds for emerging workloads. The memory footprint of emerging workloads is orders of magnitude larger than traditional workloads, and hence the capacity requirement of TLBs as well as data caches is much larger. This means the cache and TLB contents of previous context will frequently be evicted from the capacity constrained caches and TLBs since the applications need a larger amount of memory. Although there is some prior work that optimizes context switches [28, 67, 35], there is very little literature that is designed to handle the context switch scenarios caused by huge footprints of emerging workloads that flood data caches and TLBs.

Orthogonally, the performance overhead of address translation in virtualized systems is considerable as many TLB misses incur a full 2-dimensional page walk. The page walk in virtualized system begins with guest virtual address (gVA) when an application makes a memory request. However, since the guest and host system keep their own page tables, the gVA has to be translated to host physical address (hPA). First, gVA has to be translated to guest physical address (gPA), which is the host virtual address (hVA). This hVA is finally translated to gPA. This involves walking down a 2-dimensional page table. Current x86-64 employs a 4-level page table [24], so the 2-dimensional page walk may require up to 24 accesses. Making the situation worse, emerging architectures [27] introduce a 5-level page table resulting in the page walk operation to only get longer. Also, even though the L1-L2 TLBs are constantly getting bigger, they are not large enough to handle the huge footprint of emerging applications, and expensive page walks are becoming frequent.

Context switches in virtualized workloads are expensive. Since both the guest and host processes share the hardware TLBs, context switches across virtual machines can impact performance severely by evicting a large fraction of the TLB entries held by processes executing on any one virtual machine. To quantify this, we measured the increase in the L2 TLB MPKI of a context-switched system (2 virtual machine contexts, switched every 10ms) over a non-context-switched baseline. Figure 1 illustrates the increase in L2 TLB MPKIs for several multi-threaded workloads, when additional virtual machine context switches are considered. Despite only two VM contexts, the impact on the the L2 TLB is severe: an average increase in TLB MPKI of over 6X. This observation motivates us to mitigate the adverse impact of increased page walks due to context switches.

Conventional page walkers as well as addressable large-capacity translation caches (such as Oracle SPARC TSB [50]) generate accesses that get cached in the data caches. In fact, these translation schemes rely on successful caching of translation (or intermediate page walk) entries in order to reduce the cost of page walks. There has also been some recent work that attempts to improve the address translation problem by implementing a very large L3 TLB that is a part of the addressable memory [62]. The advantage of this scheme titled POM-TLB is that since the TLB is very large (several orders of magnitude larger than conventional on-chip TLBs), it has room to hold most required translations, and hence most page walks are eliminated. However, since the TLB request is serviced from the DRAM, the latency suffers. The POM-TLB entries are cached in fast data caches to reduce the latency problem, however, all of the aforementioned caching schemes suffer from the problem of cache contention due to the additional load on data caches caused by the cached translation entries.

As L2 TLB miss rates go up, proportionately, the number of translation-related accesses also go up, resulting in congestion in the data caches. Since a large number of TLB entries are stored in data caches, now the data traffic hit rate is affected. When the cache congestion effects are added on top of cache thrashing due to context switching, which is common in modern virtualized systems, the amount of performance degradation is not negligible.

In this paper, we present CSALT (read as "sea salt") which employs a novel dynamic cache partitioning scheme to reduce the contention in caches between data and TLB entries. CSALT employs a partitioning scheme based on monitoring of data and TLB stack distances and marginal utility principles. In this paper, we architect CSALT over a large L3 TLB which can practically hold all required TLB entries. However, CSALT can be easily architected atop any other translation scheme. CSALT addresses increased cache congestion when L3 TLB entries (or entries pertaining to translation in other translation schemes) are allowed to be cached into L2 and L3 data caches by means of a novel cache partitioning scheme that separates the TLB and data traffic. This mechanism helps to withstand the increased memory pressure from emerging large footprint workloads especially in the virtualized context switching scenarios.

This paper makes the following contributions:

- To the best of our knowledge, our work is the first to demonstrate the impact of virtual machine context switching on L2 TLB performance and page walk overheads.
- We identify the cache congestion problem caused by the data caching of TLB entries and propose TLB-aware cache allocation algorithms that improve both data and TLB hit rates in data caches.
- We demonstrate that CSALT effectively addresses the problem of increased page walks due to context switches.
- Through detailed evaluation, we show that the CSALT architecture achieves an average performance improvement of 85% over a conventional architecture with L1-L2 TLBs, and 25% improvement over a state-of-the-art large L3 TLB architecture.

The rest of this paper is organized as follows: Section 2 briefly discusses background on context switches and address translation in virtualized systems and shows the performance bottleneck associated with context switches. Section 3 describes the CSALT architecture. Section 4 shows the experimental platform, followed by performance results in Section 5. Section 6 discusses the related work and finally conclude the paper in Section 7.

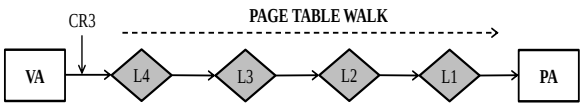
2. BACKGROUND AND MOTIVATION

In this section, we describe the background on address translation in virtualized systems and context switches. Cache contention arising from sharing of data caches with translation entries is studied.

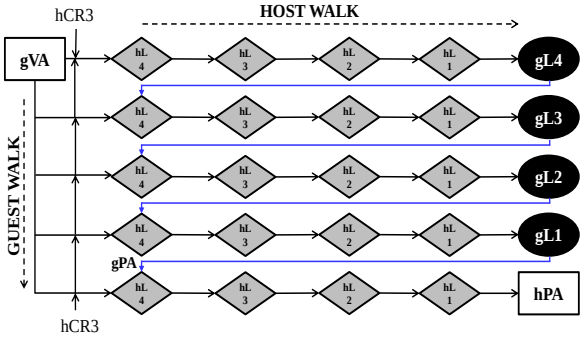
2.1 Address Translation

The address translation in modern computers requires multiple accesses to the memory subsystem. Multi-level page tables are used, and a part of the virtual address is used to index into each level. In case of today's x86-64, a four-level page table is adopted [24]. Intel recently announced that a newer generation of processors will be able to exploit the five-level page tables to further increase the reach of the physical address space [27]. However, in this paper, we focus on conventional four-level page tables. A five-level page table will only strengthen the motivation for the proposed CSALT scheme. The procedure to perform the full translation is shown in Figure 2a. A part of virtual address (VA) is used along with the CR3 register to index into the first level of the page table, which is denoted as L4 in the figure. The numbers in round parenthesis indicate the step in the address translation. For example, the step in Figure 2a involving L4 is the first step in computing the physical address, so this step is denoted as ''1" in the figure. In order to compute the physical address (PA) from virtual address (VA), four steps are needed. Although there are recent enhancements such as MMU caches [25, 12] that can reduce the number of walks by caching partial translation, the address translation incurs non-negligible performance overhead.

In virtualized systems, the address translation overhead increases. Table 1 plots the measured page walk cost per L2 TLB miss in both native and virtualized systems on a state-of-the-art system with extended page tables. While some workloads (e.g., streamcluster) have very similar page walk costs in both native and virtualized, others (e.g., connectedcomponent, gups) show significant increase under virtualization. The problem in the virtualized system is that guest virtual machine needs to keep its own page table while the host system needs to keep its own page table. Therefore, the hypervisor has to be involved in translating the guest-side addresses to host-side addresses. Having the hypervisor involved in every TLB miss is costly, so modern processors employ nested page tables [1, 24] where the page walks are done in a two-dimensional way. Figure 2b shows the full translation starting from guest virtual address (gVA) to host physical address (hPA). Such translation requires a two-dimensional radix-4 walk since each level of translation on the guest side needs the full 4-level translation on the host side. Therefore, in the worst case, the system has to access the memory subsystem 24 times as shown in Figure 2b. In practice, many of the intermediate page table entries are cached in MMU caches and data caches, so most accesses do not incur expensive off-chip DRAM accesses; however, having such a large number of accesses is still expensive.



(a) 1-Dimensional Page Table Walk (Native)



(b) 2-Dimensional Page Table Walk (Virtualized)

Figure 2: Page Table Walks in Native and Virtualized Systems

Benchmark	Native	Virtualized
canneal	53	61
connectedcomponent	44	1158
graph500	79	80
gups	43	70
pagerank	51	61
streamcluster	74	76

Table 1: Average Page Walk Cycles Per L2 TLB miss

2.2 Motivation

Today it is common to have multiple VM instances to share a common host system as cloud vendors try to maximize hardware utilization. Figure 1 shows that the context switching between virtual machines leads to a significant increase in L2 TLB miss rates in workloads with large working sets. This leads to an overall degradation in performance of the context-switched workloads. For instance, when 1 VM instance of *pagerank* was context-switched with another VM instance of the same workload, the total program execution cycles for each instance went up by a factor of $2.2X$.

The higher miss rate of the L2 TLB leads to increased translation traffic to the data caches. In the conventional radix tree based page table organization, the additional page walks result in the caching of intermediate page tables [65]. In the POM-TLB organization, the caches store *translation entries* instead of page table entries[1]. While caching of TLB entries inherently causes less congestion (one entry per translation as opposed to multiple intermediate page table entries), it still re-

[1] By *translation entry* we refer to a TLB entry that stores the translation of a virtual address to its physical address.

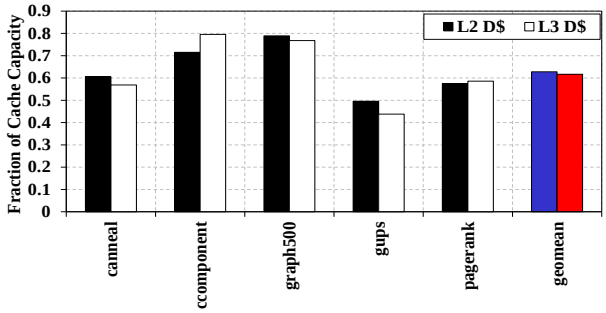


Figure 3: Fraction of Cache Capacity Occupied by TLB Entries

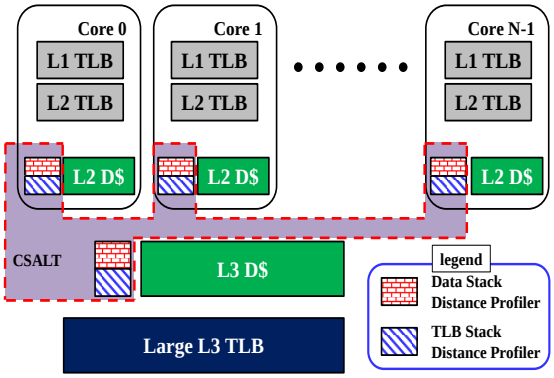


Figure 4: CSALT System Architecture

sults in polluting the data caches when the L2 TLB miss rates are high. This scenario creates an undesirable situation where neither data nor TLB traffic achieves the optimal hit rate in data caches. A conventional system is not designed to handle such scenarios as the conventional cache replacement policy does not distinguish different types of cache contents. This is no longer true as some contents are data contents while others are TLB contents. When a replacement decision is made, it does not distinguish TLB contents versus data contents. But the data and TLB contents impact system performance differently. For example, data requests are overlapped with other data requests with the help of MSHR. On the other hand, an address translation request is a blocking access, so it stalls the pipeline. Although newer processor architectures such as Skylake [24] have simultaneous page table walkers to allow up to two page table walks, the page table walk being a blocking access does not change. In the end, the conventional content-oblivious cache replacement policy makes both the TLB and data access performance suffer by making them compete for entries in capacity constrained data caches. This problem is exacerbated when frequent context switches occur between virtual machines.

To quantify the cache congestion problem, we measure the *occupancy* of TLB entries in L2 and L3 data caches. We define *occupancy* as the average fraction of cache blocks that hold TLB entries[2]. Figure 3 plots this data for several workloads[3]. We observe that an average of 60% of the cache capacity holds translation entries. In one workload (*connectedcomponent*), the TLB entry occupancy is as high as 80%. This is because the L2 TLB miss rate is approximately 10 times the L1 data cache miss rate, as a result of which translation entries end up dominating the cache capacity.

While caching of the translation entries is useful to avoid DRAM accesses, the above data suggests that unregulated caching of translation entries has a flip side of causing cache pollution or creating capacity conflict with data entries. This motivates the proposed CSALT architecture that creates a TLB-aware cache management framework.

[2]To collect this data, we modified our simulator to maintain a type field (TLB or data) with each cache block; periodically the simulator scanned the caches to record the fraction of TLB entries held in them.

[3]Refer Section 4 for details of evaluation methodology and workloads

3. CONTEXT SWITCH AWARE LARGE TLB

The address translation overhead in virtualized systems comes from one apparent reason, the lack of TLB capacity. If the TLB capacity were large enough, most of page table walks would have been eliminated. The need for a larger TLB capacity is also seen as a recent generation of Intel processors [4] doubled the L2 TLB capacity from the previous generation. Traditionally, TLBs are designed to be small and fast, so that the address translation can be serviced quickly. Yet, emerging applications require much more memory than traditional server workloads. Some of these applications have terabytes of memory footprint, so that TLBs, which were not initially designed for such huge memory footprint, suffer significantly.

Recent work [62] by Ryoo et al. uses a part of main memory to be used as a large capacity TLB. They use 16MB of the main memory, which is negligible considering high-end servers have terabytes of main memory these days. However, 16MB is orders of magnitude higher than today's on-chip TLBs, and thus, it can eliminate virtually all page table walks. This design achieves the goal of eliminating page table walks, but now this TLB suffers from slow access latency since off-chip DRAM is much slower than on-chip SRAMs. Consequently, they make this high-capacity TLB as addressable, so TLB entries can be stored in data caches. They call this TLB as POM-TLB (Part of Memory TLB) as the TLB is given an explicit address space. CSALT uses the POM-TLB organization as its substrate. It may be noted that CSALT is a cache management scheme, and can be architected over other translation schemes such as conventional page tables.

Figure 4 depicts the system architecture incorporating CSALT architected over the POM-TLB. CSALT encompasses L2 and L3 data cache management schemes. The role of the stack distance profilers shown in the figure is described in Section 3.1. In the following subsections, we describe the architecture of our Context-Switch Aware Large TLB (CSALT) scheme. First, we explain the dynamic partitioning algorithm that helps to find a balanced partitioning of the cache between TLB and data entries to reduce the cache contention. In Section 3.2, we introduce a notion of "criticality" to improve the dynamic partitioning algorithm by taking into account the relative costs of data cache misses. We also describe the hardware overheads of these partitioning

algorithms.

3.1 CSALT with Dynamic Partitioning (CSALT-D)

Since prior state-of-the-art work [62] does not distinguish data and TLB entries when making cache replacement decisions, it achieves a suboptimal performance improvement. The goal of CSALT is to profile the demand for data and TLB entries at runtime and adjust the cache capacity needed for each type of cache entry.

CSALT dynamic partitioning algorithm (*CSALT-D*) attempts to maximize the overall hit rate of data caches by allocating an optimal amount of cache capacity to data and TLB entries. In order to do so, CSALT-D attempts to minimize interference between the two entry types. Assuming that a cache is statically partitioned by half for data and TLB entries, if data entries have higher miss rates with the current allocation of cache capacity, CSALT-D would allocate more capacity for data entries. On the other hand, if TLB entries have higher miss rates with the current partitioning scheme, CSALT-D would allocate more cache for TLB entries. The capacity partitioning is adjusted at a fixed interval, and we refer to this interval as an epoch in this paper. In order to obtain an estimate of cache hit/miss rate for each type of entry when provisioned with a certain capacity, we implement a cache hit/miss prediction model for each type of entry based on Mattson's Stack Distance (MSA) algorithm [43]. The MSA uses the LRU information of set-associative caches. For a K-way associative cache, LRU stack is an array of $(K+1)$ counters, namely $Counter_1$ to $Counter_{K+1}$. $Counter_1$ counts the number of hits to the Most Recently Used (MRU) position, and $Counter_K$ counts the number of hits to the LRU position. $Counter_{K+1}$ counts the number of misses incurred by the set. Each time there is a cache access, the counter corresponding to the LRU stack distance where the access took place is incremented.

LRU stack can be used to predict the hit rate of the cache when the associativity is increased/reduced. For instance, consider a 16-way associative cache where we record LRU stack distance for each of the accesses in a LRU stack. If we decrease the associativity to 4, all the accesses which hit in positions $LRU4 - LRU15$ in the LRU stack previously would result in a miss in the new cache with decreased associativity ($LRU0$ is the MRU position). Therefore, an estimate of the hit rate in the new cache with decreased associativity can be obtained by summing up the hit rates in the LRU stack in positions $LRU0 - LRU3$.

For a K-way associative cache, our dynamic partitioning scheme works by allocating certain ways $(0 : N-1)$ for data entries and the remaining ways for TLB entries $(N : K-1)$ in each set in order to maximize the overall cache hit rate. For each cache which needs to be dynamically partitioned, we introduce two additional structures: a data LRU stack, and a TLB LRU stack corresponding to data and TLB entries respectively. The data LRU stack serves as a cache hit rate prediction model for data entries whereas the TLB LRU stack serves as as a cache hit rate prediction model for TLB entries. Estimates of the overall cache hit rates can be obtained by summing over appropriate entries in the data and TLB LRU stack. For instance, in a 16-way associative cache with 10 ways allocated for data entries and remaining ways allocated for TLB entries, an estimate of the overall cache hit rate can be obtained by summing over $LRU0 - LRU9$ in Data LRU stack and $LRU0 - LRU5$ in the TLB LRU stack.

Algorithm 1 Dynamic Partitioning Algorithm

```
N = Number of ways to be allocated for data
M = Number of ways to be allocated for TLB

for n in N_min : K − 1 do
    MU_n = compute_MU(n)

N = arg max_N (MU_{N_min}, MU_{N_min+1}, ..., MU_{K−1})
M = K - N
```

This estimate of the overall cache hit rate obtained from the LRU stack is referred to as the *Marginal Utility* of the partitioning scheme [32]. Consider a K-way associative cache. Let the data LRU stack be represented as `D_LRU` and the TLB LRU stack be represented as `TLB_LRU`. Consider a partitioning scheme P that allocates N ways for data entries and $K-N$ ways for TLB entries. Then the *Marginal Utility* of P, denoted by MU_N^P is given by the following equation,

$$MU_N^P = \sum_{i=0}^{N-1} \texttt{D\_LRU}(i) + \sum_{j=0}^{K-N-1} \texttt{TLB\_LRU}(j). \qquad (1)$$

CSALT-D attempts to maximize the marginal utility of the cache at each epoch by comparing the marginal utility of different partitioning schemes. Consider the example shown in Figure 5 for an 8-way associative cache. Suppose the current partitioning scheme assigns $N = 4$ and $M = 4$. At the end of an epoch, the `D_LRU` and `TLB_LRU` contents are shown in Figure 5. In this case, the dynamic partitioning algorithm finds the marginal utility for the following partitioning schemes (not every partitioning is listed):

$$MU_4^{P1} = \sum_{i=0}^{3} \texttt{D\_LRU}(i) + \sum_{j=0}^{3} \texttt{TLB\_LRU}(j) = 34$$

$$MU_5^{P2} = \sum_{i=0}^{4} \texttt{D\_LRU}(i) + \sum_{j=0}^{2} \texttt{TLB\_LRU}(j) = 30$$

$$MU_6^{P3} = \sum_{i=0}^{5} \texttt{D\_LRU}(i) + \sum_{j=0}^{1} \texttt{TLB\_LRU}(j) = 40$$

$$MU_7^{P4} = \sum_{i=0}^{6} \texttt{D\_LRU}(i) + \sum_{j=0}^{0} \texttt{TLB\_LRU}(j) = 50$$

Among the computed marginal utilities, our dynamic scheme chooses the partitioning that yields the best marginal utility. In the above example, CSALT-D chooses partitioning scheme $P4$. This is as elaborated in Algorithm 1 and Algorithm 2.

Once the partitioning scheme P_{new} is determined by the CSALT-D algorithm, it is enforced globally on all cache sets. Suppose the old partitioning scheme P_{old} allocated N_{old} ways for data entries, and the updated partitioning scheme P_{new} allocates N_{new} ways for data entries. We consider two cases: (a) $N_{old} < N_{new}$ and (b) $N_{old} > N_{new}$ and discuss how the partitioning scheme P_{new} affects the cache lookup and

Algorithm 2 Computing Marginal Utility

```
N = Input
D_LRU = Data LRU Stack
TLB_LRU = TLB LRU Stack
MU = 0

for i in 0 : N − 1 do
    MU += D_LRU (i)
for j in 0 : K − N − 1 do
    MU += TLB_LRU (j)
return MU
```

cache replacement. While CSALT-D has no affect on cache lookup, CSALT-D does affect replacement decisions. Here, we describe the lookup and replacement policies in detail.

Cache Lookup: All K-ways of a set are scanned irrespective of whether a line corresponds to a data entry or a TLB entry during cache lookup. In case (a), even after enforcing P_{new}, there might be TLB entries resident in the ways allocated for data (those numbered N_{old} to $N_{new}-1$). On the other hand, in case (b), there might be data entries resident in the ways allocated for TLB entries (ways numbered N_{new} to $N_{old}-1$). This is why all ways in the cache is looked up as done in today's system.

Cache Replacement: In the event of a cache miss, consider the case where an incoming request corresponds to a data entry. In both case (a) and (b), CSALT-D evicts the LRU cacheline in the range $(0, N_{new}-1)$ and places the incoming data line in its position. On the other hand, if the incoming line corresponds to a TLB entry, in both case (a) and (b), CSALT-D evicts the LRU-line in the range $(N_{new}, K-1)$ and places the incoming TLB line in its position.

Classifying Addresses as Data or TLB: Incoming addresses can be classified as data or TLB by examining the relevant address bits. Since the POM-TLB is a memory mapped structure, the cache controller can identify if the incoming address is to the POM-TLB or not. For stored data in the cache, there are two ways by which this classification can be done: i) by adding 1 bit of metadata per cache block to denote data (0) or TLB (1), or ii) by reading the tag bits and determining if the stored address falls in the L3 TLB address range or not. We leave this as an implementation choice. In our work, we assume the latter option as it does not affect metadata storage.

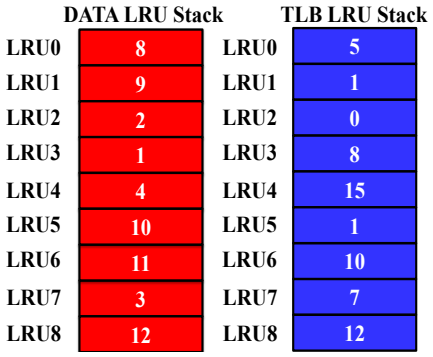


Figure 5: LRU Stack Example

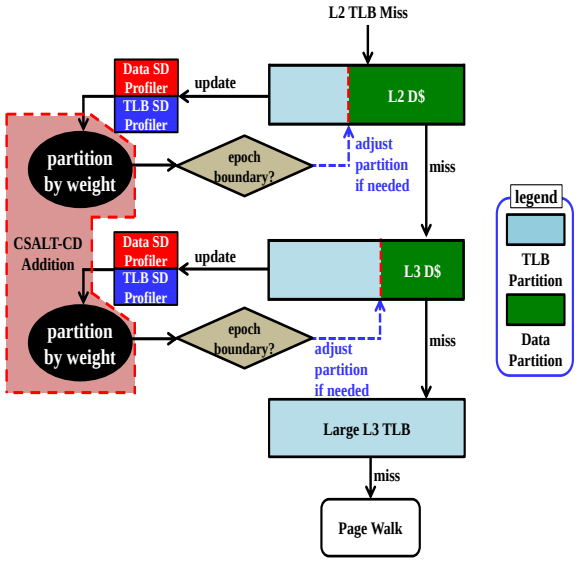


Figure 6: CSALT Overall Flowchart

Finally, the overall flow is summarized in Figure 6. Each private L2 cache maintains its own stack distance profilers and updates them upon accesses to it. When an epoch completes, it computes marginal utilities and sets up a (potentially different) configuration of the partition between data ways and TLB ways. Misses (and writebacks) from the L2 caches go to the L3 cache which performs a similar update of its profilers and configuration outcome. A TLB miss from the L3 data cache is sent to the L3 TLB. Finally, a miss in the L3 TLB triggers a page walk.

3.2 CSALT with Criticality Weighted Partitioning (CSALT-CD)

CSALT-D assumes that the impact of data cache misses is equal for both data and TLB entries, and as a result, both the data and TLB LRU stacks had the same weight when computing the marginal utility. However, this is not necessarily true since a TLB miss can cause a long latency page walk[4]. In order to maximize the performance, the partitioning algorithm needs to take the relative performance gains obtained by TLB entry hit and the data entry hit in the data caches into account.

Therefore, we propose a dynamic partitioning scheme that considers criticality of data entries, called Criticality Weighted Dynamic Partitioning (*CSALT-CD*). We use the insight that data and TLB misses incur different penalties on a miss in the data cache. Hence, the outcome of stack distance profiler is scaled by its importance or weight, which is the performance gain obtained by a hit in the data cache. Figure 6 shows an overall flowchart with additional hardware to enable such scaling (the red shaded region shows the additional hardware).

In *CSALT-CD*, a performance gain estimator is added to estimate the impact of a TLB entry hit and a data entry hit

[4]Note that even if the translation request misses in an L3 data cache, the entry may still hit in the L3 TLB thereby avoiding a page walk.

Algorithm 3 Computing CWMU

```
N = Input
D_LRU = Data LRU Stack
TLB_LRU = TLB LRU Stack
CWMU = 0

for i in 0 : N − 1 do
    CWMU += S_Dat × D_LRU (i)
for j in 0 : K − N − 1 do
    CWMU += S_Tr × TLB_LRU (j)
return CWMU
```

on performance. In an attempt to minimize hardware overheads, *CSALT-CD* uses existing performance counters. For estimating the hit rate of the L3 data cache, *CSALT-CD* uses performance counters that measures the number of L3 hits and the total number of L3 accesses that are readily available on modern processors. For estimating the L3 TLB hit rate, a similar approach is used. Utilizing this information, the total number of cycles incurred by a miss for each kind of entry is computed dynamically. The ratio of the number of cycles incurred by a miss to the number of cycles incurred by a hit for each kind of entry is used to estimate the performance gain on a hit to each kind of entry. For instance, if a data entry hits in the L3 cache, the performance gain obtained is the ratio of the average DRAM latency to the total L3 access latency. If a TLB entry hits in the L3 cache, the performance gain obtained is the ratio of the sum of the TLB latency and the average DRAM latency to the total L3 access latency. These estimates of performance gains are directly plugged in as *Criticality Weights* which are used to scale the *Marginal Utility* from the stack distance profiler. We define a new quantity called the *Criticality Weighted Marginal Utility*. For a partitioning scheme P which allocates N data ways out of K ways, Criticality Weighted Marginal Utility (CWMU), denoted as $CWMU_N^P$, is given by the following equation[5],

$$CWMU_N^P = S_{Dat} \times \sum_{i=0}^{N-1} \texttt{D\_LRU}(i) + S_{Tr} \times \sum_{j=0}^{K-N-1} \texttt{TLB\_LRU}(j). \quad (2)$$

The partitioning scheme with the highest CWMU is used for the next epoch. Figure 6 shows the overall flow chart of CSALT-CD with the additional step required (the red shaded is the addition for CSALT-CD). We have used separate performance estimators for L2 and L3 data caches as the performance impact of L2 and L3 data caches is different. Algorithm 3 shows the pseudocode of *CSALT-CD*. For a data entry, this performance gain is denoted by S_{Dat}, and for a TLB entry, by S_{Tr}. These criticality weights are dynamically estimated using the approach elaborated earlier. The rest of the flow (cache accesses, hit/miss evaluation, replacement decisions) is the same as in CSALT-D.

3.3 Hardware Overhead

Both CSALT-D and CSALT-CD algorithms use stack distance profilers for both data and TLB. The area overhead for each stack distance profiler is negligible. This structure requires the MSA LRU stack distance structure, which is equal to the number of ways, so in case of L3 data cache, it is 16 entries. Computing the marginal utility only requires a few adders that will accumulate the sum of a few entries in the stack distance profiler. Both CSALT-D and CSALT-CD also require an internal register per partitioned cache which contains information about the current partitioning scheme, specifically, N, the number of ways allocated for data in each set. The overhead of such a register is minimal, and depends on the associativity of the cache. Furthermore, the CSALT-CD algorithm uses a few additional hardware structures, which include the hit rates of L3 data cache and L3 TLB. However, these counters are already available on modern processors as performance monitoring counters. Thus, estimating the performance impact of data caches and TLBs will only require a few multipliers that will be used to scale the marginal utility by weight. Therefore, we observe that the additional hardware overhead required to implement CSALT with criticality weighted partitioning is minimal.

3.4 Effect of Replacement Policy

Until this point, we assumed a True-LRU replacement policy for the purpose of cache partitioning. However, True-LRU is quite expensive to implement, and is rarely used in modern processors. Instead, replacement policies like Not Recently Used (NRU) or Binary Tree (BT) pseudo-LRU are used [33]. Fortunately, the cache partitioning algorithms utilized by CSALT are not dependent on the existence of True-LRU policy. There has been prior research to adapt cache partitioning schemes to Pseudo-LRU replacement policies [33], and we leverage it to extend CSALT.

For NRU replacement policy, we can easily estimate the LRU stack positions depending on the value of the NRU bit on the accessed cache line. For Binary Tree-pseudoLRU policy, we utilize the notion of an Identifier (ID) to estimate the LRU stack position. Identifier bits for a cache line represent the value that the the binary tree bits would assume if a given line held the LRU position. In either case, estimates of LRU stack positions can be used to update the LRU stack. It has been shown that using these estimates instead of the actual LRU stack position results in only a minor performance degradation [33].

4. EXPERIMENTAL SET-UP

We evaluate the performance of CSALT using a combination of real system measurements, Pin tool [40], and heavily modified Ramulator [34] simulation. The virtualization platform is QEMU [11] 2.0 with KVM [20] support. Our host system is Ubuntu 14.04 running on Intel Skylake [24] with Transparent Huge Pages (THP) [8] turned on. The system also has Intel VT-x with support for Extended Page Tables [26]. The host system parameters are shown in Table 2 under Processor, MMU, and PSC categories. The guest system is Ubuntu 14.04 also with THP turned on. Although the host system has a separate L1 TLBs for 1GB pages, we do not make use of it. The L2 TLB is a unified TLB for both 4KB and 2MB pages. In order to measure page walk overheads, we use specific performance counters (e.g., 0x0108, 0x1008,

[5]We could normalize the values in the LRU stack with respect to the number of data and TLB entry accesses, but we do not do so for the sake of simplicity

Processor	Values
Frequency	4 GHz
Number of Cores	8
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 entry, 9 cycles
L1 TLB (2MB)	32 entry, 9 cycles
	L1 TLBs 4 way associative
L2 Unified TLB	1536 entry, 17 cycles
	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 2: Experimental Parameters

VM1	VM2
canneal_x8	connected component_x8
canneal_x8	streamcluster_x8
graph500_x8	gups_x8
pagerank_x8	streamcluster_x8

Table 3: Heterogeneous Workloads Composition

0x0149, 0x1049), which take MMU caches into account. The page walk cycles used in this paper are the average cycles spent after a translation request misses in L2 TLB.

4.1 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 PARSEC [15] applications that are known to be memory intensive. In addition, we also ran graph benchmarks such as the *graph500* [5] and big data benchmarks such as *connected component* [36] and *pagerank* [51]. We paired two multi-threaded benchmarks (two copies of the same program, or two different programs) to study the problems introduced by context switching. The heterogeneous workload composition is listed in Table 3. The x8 denotes the fact that all our workloads are run with 8 threads.

4.2 Simulation

Our simulation methodology is different from prior work [56, 55] that relied on a linear additive performance model. The drawback of the linear model is that it does not take into account the overlap of instructions and address translation traffic, but merely assumes that an address translation request is blocking that the processor immediately stalls upon a TLB miss. This is not true in modern hardware as the remaining instructions in the ROB can continue to retire as well as some modern processors [24] allow simultaneous page walkers. Therefore, we use a cycle accurate simulator that uses a heavily modified Ramulator. We ran each workload 10 billion instructions. The front-end of our simulator uses the timed traces collected from real system execution using the Pin tool. During playback, we simulate two contexts by switching between two input traces every 10ms. We choose 10ms as the context switch granularity based on measured data from prior works [37, 38].

In our simulation, we model the TLB datapath where the TLB miss still lets the processor to flush the pipeline, so the overlap aspect is well modeled. We simulate the entire memory system accurately, including the effects of translation accesses on L2 and L3 data caches as well as the misses from data caches that are serviced by POM-TLB or off-chip memory. The timing details of our simulator are summarized in Table 2.

The performance improvement is calculated by using the ratio of improved IPC (geometric mean across all cores) over the baseline IPC (geometric mean across all cores), and thus, higher normalized performance improvement indicates a higher performing scheme.

5. RESULTS

This section presents simulation results from a conventional system with only L1-L2 TLBs, a POM-TLB system, and various CSALT configurations. POM-TLB is the die-stacked TLB organization using the LRU replacement scheme in L2 and L3 caches [62]. CSALT-D refers to proposed scheme with dynamic partitioning in L2, L3 data caches. CSALT-CD refers to proposed scheme with *Criticality-Weighted* dynamic partitioning in L2, L3 data caches.

5.1 CSALT Performance

We compare the performance (normalized IPC) of the baseline, POM-TLB, CSALT-D and CSALT-CD in this section. Figure 7 plots the performance of these schemes. Note that we have normalized the performance of all schemes using the POM-TLB. POM-TLB, CSALT-D and CSALT-CD all gain over the conventional system in every workload. The large shared TLB organization helps reduce expensive page walks and improves performance in the presence of context switches and high L2 TLB miss rates. This is confirmed by Figure 8 which plots the reduction in page walks after the POM-TLB is added to the system. In the presence of context switches (that cause L2 TLB miss rates to go up by $6X$), the POM-TLB eliminates the vast majority of page walks, with average reduction of 97%. It may be emphasized that no prior work has explored the use of large L3 TLBs to mitigate the page walk overhead due to context switches.

Both CSALT-D and CSALT-CD outperform POM-TLB, with average performance improvements of 11% and 25% re-

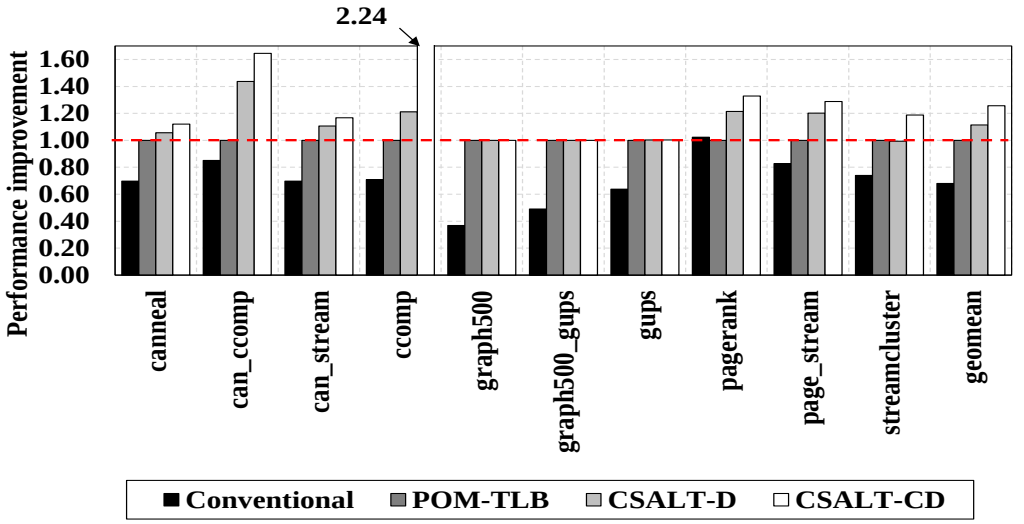


Figure 7: Performance Improvement of CSALT (normalized to POM-TLB)

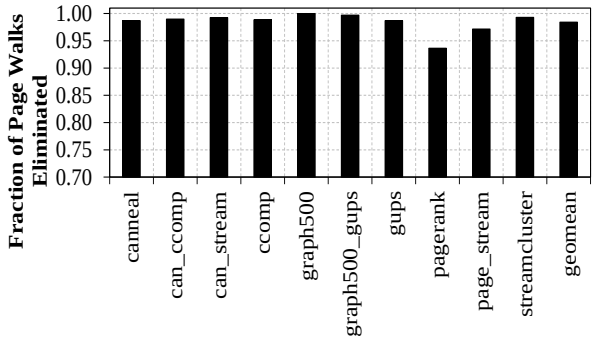


Figure 8: POM-TLB: Fraction of Page Walks Eliminated

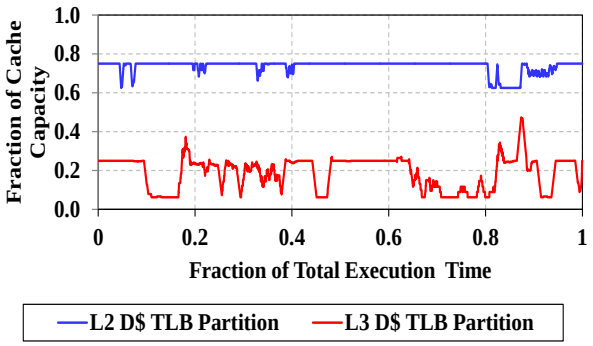


Figure 9: Fraction of TLB Allocation in Data Caches

spectively. Both the dynamic schemes[6] show steady improvements over POM-TLB highlighting the need for cache decongestion on top of reducing page walks. In the *connected-component* workload[7], CSALT-CD improves performance by a factor of 2.2*X* over POM-TLB demonstrating the benefit of carefully balancing the shared cache space to TLB and data storage. In gups and graph500, just having a large L3 TLB improves performance significantly but then there is no additional improvement obtained by partitioning the caches.

In order to analyze how well our CSALT scheme works, we deep dive into one workload, *connected_component*. Figure 9 plots the fraction of L2 and L3 cache capacity allocated to TLB entries during the course of execution for *connected_component*. The TLB capacity allocation follows closely with the application behaviors. For example, the workload processes a list of active vertices (a segment of graph) in each iteration. Then, a new list of active vertices is generated based on the edge connections of vertices in the current list. Since vertices in the active list are placed in random number of pages, this workloads produces different levels of TLB pressure when a new list is generated. This is apparent that the L2 data cache, which is more performance critical, favors TLB entries in some execution phases. This phase is when the new list is generated. By dynamically assessing and weighing the data and TLB traffic, CSALT-CD is able to vary the proportion allocated to TLB, which satisfies the requirements of application. Interestingly, when more of L2 data cache capacity is allocated to TLB entries, we see a drop in L3 allocation for TLB entries. Since a larger L2 capacity for TLB entries reduces the number of TLB entry misses, the L3 data cache needs lesser capacity for TLB entries. Even though L2 and L3 data cache partitioning works independently, our stack distance profiler as well as performance estimators work cooperatively and optimize the overall system performance. The significant improvement in performance of CSALT over POM-TLB can be quantitatively explained by examining the reduction in the L2 and L3 MPKIs. Figures 10 and 11 plot the relative MPKIs of POM-TLB, CSALT-D and CSALT-CD in L2 and L3 data caches respectively (relative to POM-TLB MPKI). Both CSALT-D and CSALT-CD achieve MPKI reductions in both L2 and L3 data caches. In *connected-component*, both CSALT-D and CSALT-CD reduce MPKI of the L2 cache by as much as 30%. CSALT-CD achieves a reduction of 26% in the L3 MPKI as

[6]We also implemented static cache partitioning schemes and found that no one static scheme performed well across all the workloads.
[7]When we refer to a single benchmark, we refer to two instances of the benchmark co-scheduled.

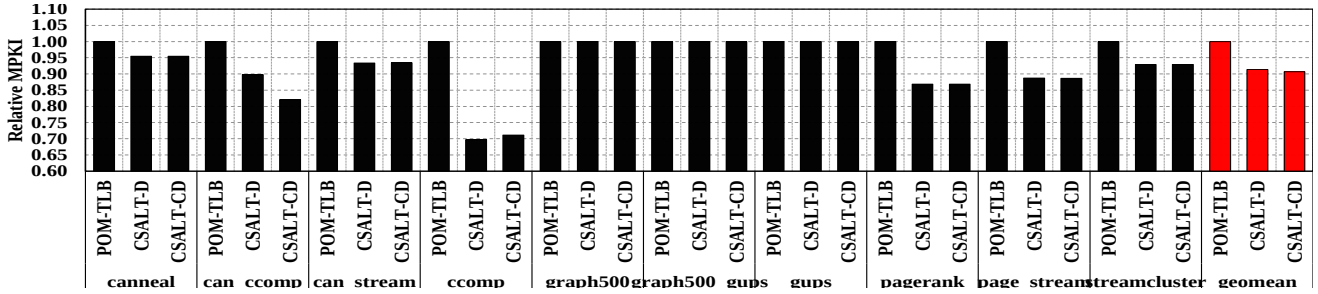


Figure 10: Relative L2 Data Cache MPKI over POM-TLB

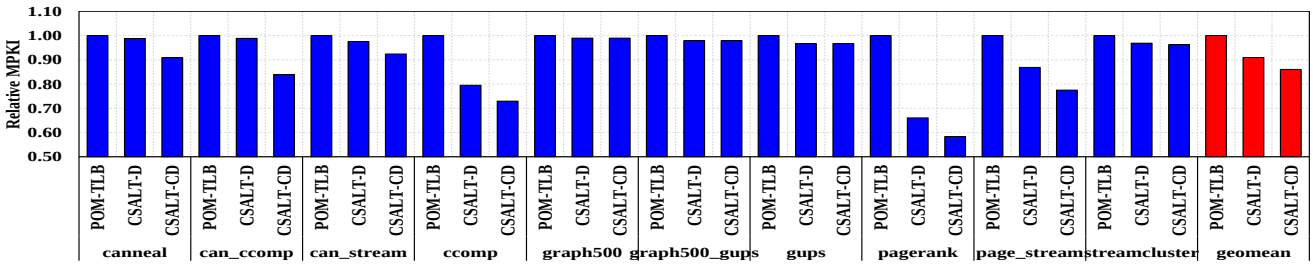


Figure 11: Relative L3 Data Cache MPKI over POM-TLB

well. These reductions indicate that CSALT is successfully able to reduce cache misses by making use of the knowledge of the two streams of traffic.

These results also show the effectiveness of our *Criticality-Weighted* Dynamic partitioning. In systems subject to virtual machine context switches, since the L2 TLB miss rate goes up significantly, a careful management of cache capacity factoring in the TLB traffic becomes important. While TLB traffic is generally expected to be a small fraction in comparison to data traffic, our investigation shows that this is not always the case. In workloads with large working sets, frequent context switches can result in generating significant TLB traffic to the caches. CSALT-CD is able to handle this increased demand by judiciously allocating cache ways to TLB and data.

5.1.1 CSALT Performance in Native Systems

While CSALT is motivated by the problem of high translation overheads in context switched virtualized workloads, it is equally applicable to native workloads that suffer high translation overheads. Figure 12 shows that CSALT achieves an average performance improvement of 5% in native context-switched workloads with as much as 30% improvement in the *connectedcomponent* benchmark.

5.2 Comparison to Prior Works

Since CSALT uses a combination of an addressable TLB and a dynamic cache partitioning scheme, we compare its performance against two relevant existing schemes: i) Translation Storage Buffers (TSB, implemented in Sun Ultrasparc III, see [50]), and ii) DIP [58], a dynamic cache insertion policy which we implemented on top of POM-TLB.

We chose TSB for comparison as it uses addressable software-managed buffers to hold translation entries. Like POM-TLB, TSB entries can be cached. However, unlike

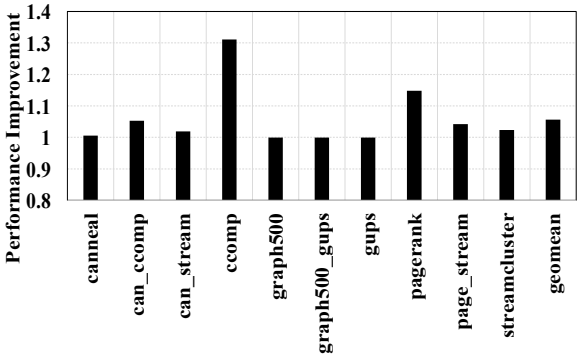


Figure 12: Performance Improvement of CSALT-CD in the native context

POM-TLB, the TSB organization requires multiple look-ups to perform guest-virtual to host-physical translation.

DIP is a cache insertion policy, which uses two competing cache insertion policies and selects the better one to reduce conflicts in order to improve cache performance. We chose DIP for comparison as we believed that the TLB entries may have different reuse characteristics that would be exploited by DIP (such as inserting such entries into cache sets at non-MRU positions in the recency stack). As DIP is not a page-walk reduction scheme, for a fair comparison, we implemented DIP on top of POM-TLB. By doing so, this scheme leverages the benefits of POM-TLB (page walk reduction) while also incorporating a dynamic cache insertion policy that is implemented based on examining all of the incoming traffic (data + TLB) into the caches.

Figure 13 compares the performance of TSB, DIP and CSALT-CD on context-switched workloads. Clearly, CSALT-CD outperforms both TSB and DIP. Since TSB requires multiple cacheable accesses to perform guest-virtual to host-

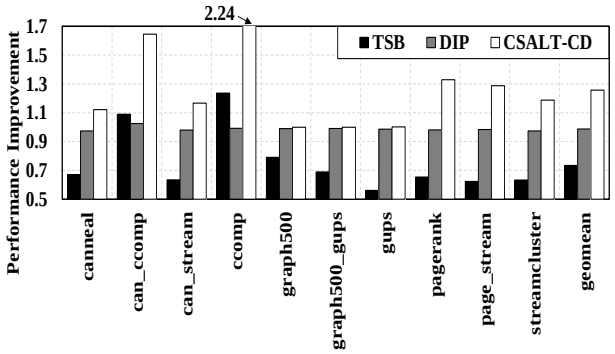


Figure 13: Performance Comparison of CSALT with Other Comparable Schemes

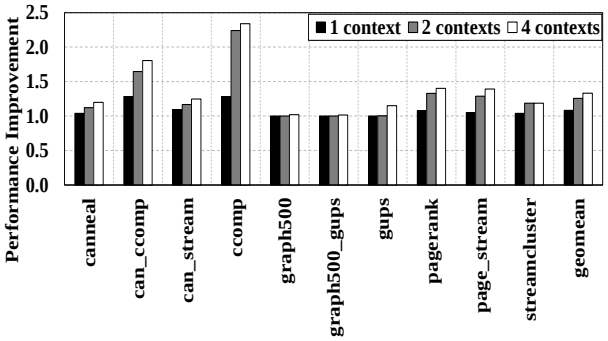


Figure 14: Performance of CSALT with Different Number of Contexts

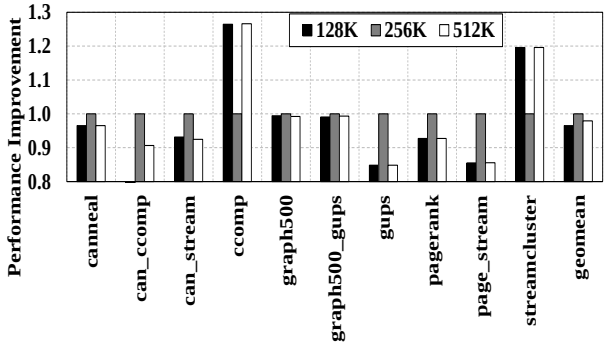


Figure 15: Performance of CSALT with Different Epoch Lengths

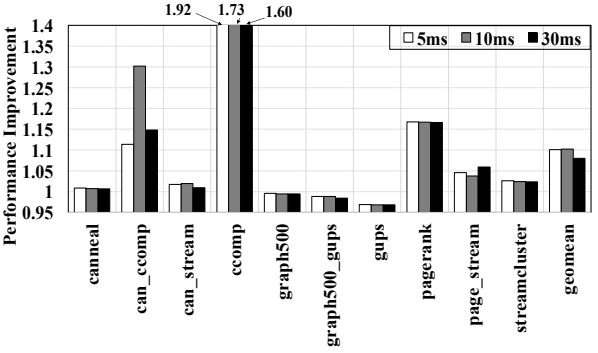


Figure 16: Performance of CSALT with Different Context Switch Intervals

physical translation, it causes greater congestion in the shared caches. Since it has no cache-management scheme that is aware of the additional traffic caused by accesses to the software translation buffers, the TSB suffers from increased load on the cache, often evicting useful data to make room for translation buffer entries. This results in the TSB under-performing all other schemes (except in *connected-component*, where it performs superior to DIP, but inferior to CSALT-CD). It may also be noted that the TSB system organization can leverage CSALT cache partitioning schemes.

As such, DIP does not distinguish between data and TLB entries in the incoming traffic and is unable to exploit this distinction for cache management. As a result, DIP achieves nearly the same performance as that of POM-TLB. This is not surprising considering that we implemented DIP on top of POM-TLB. CSALT-CD, by virtue of its TLB-conscious cache allocation, leverage cache capacity much more effectively and as a result, performs 30% better than DIP, on average.

5.3 Sensitivity Studies

In this section, we vary some of our design parameters to see their performance effects.

Number of contexts sensitivity: The number of contexts that can run on a host system vary across different cloud services. Some host machines can choose to have more contexts running than others depending on the resource allocations. In order to simulate such effects, we vary the number of contexts that run on each core. We have used a default value of 2 contexts per core, but in this sensitivity analysis, we vary it to 1 context and 4 contexts per core. We present the results on how well CSALT is able to handle the increased resource pressure. Figure 14 shows the performance improvement results for varying number of contexts. The results are normalized to POM-TLB. As expected, 1 context achieves the lowest performance improvement as there is no resource contention between multiple threads. Likewise, when we further increased the pressure by executing 4 contexts (doubled the default 2 context case), the performance increase is only 33%. This study shows that CSALT is very effective at withstanding increased system pressure by reducing the degree of contention in shared resources such as data caches.

Epoch length sensitivity: The dynamic partitioning decision is made in CSALT at regular time intervals, referred to as epochs. Throughout this paper, the default epoch length was 256,000 accesses for both L2 and L3 data cache. The epoch length at which the partitioning decision is made determines how quickly our scheme reacts to changes in the application phases. We change this epoch length after experimental evaluation. Figure 15 shows the performance improvement normalized to our default epoch length of 256K accesses when the epoch length, at which the dynamic partitioning decision is made, is changed. In some cases such as *connected_component* and *streamcluster*, shorter and longer epoch length achieve higher performance improvement than our default case. This indicates that our default epoch length is not chosen well for these workloads as it results in making

a partitioning decision based on non-representative regions of workloads. However, in all other workloads, our default is able to achieve the highest performance improvement. Therefore, in this paper, we chose the default of 256K accesses as the epoch length.

Context Switch Interval Sensitivity: The rate of context switching affects the congestion/interference on data caches and results in eviction of useful data/TLB entries. Figure 16 plots the performance gain achieved by CSALT (relative to POM-TLB) at context-switch intervals of 5, 10, and 30 ms. CSALT exhibits steady performance improvement at each of these intervals, with a slightly lower (8%) average improvement at 30 ms in comparison to 10 ms.

6. RELATED WORK

Virtual Memory: Oracle UltraSPARC mitigates expensive software page walks by using TSB [50]. Upon TLB misses, the trap handling code quickly loads the TLB from TSB where the entry can reside anywhere from the L2 cache to off-chip DRAM. However, TSB requires multiple memory accesses to load the TLB entry in virtualized environments as opposed to a single access in our scheme (refer to Figure 15 in [73] for an overview of the TSB address translation steps in virtualized environments). Further, our TLB-aware cache partitioning scheme is applicable to the TSB as well, and as demonstrated in Section 5, TSB architecture also sees performance improvement.

Modern processors implement MMU caches such as Intel's PSC [24] and AMD's PWC [12] that store partial translation to eliminate page walks. However, the capacity is still much smaller than application footprints that a large number of page walks are still inevitable. Other proposals like cooperative caching [16], shared last level TLBs [13], and cooperative TLBs [14] exploit predictable memory access patterns across cores. These techniques are orthogonal to our approach and can be applied on top of our scheme since we use a shared TLB implemented in DRAM. Although software-managed TLBs have been proposed for virtualized contexts [18], we limit our work on hardware managed TLBs.

Speculation schemes [9, 56] continue the processor execution with speculated page table entries and invalidate speculated instructions upon detecting the mispeculation. These schemes can effectively hide the overheads of page table walks. On the other hand, our scheme addresses more a fundamental that the TLB capabity is not enough, so we aim to reduce the number of page walks significantly by having much larger capacity.

Huge pages (e.g., 2MB or 1GB in x86-64) can reduce TLB misses by having a much larger TLB reach [19, 49, 53]. Our approach is orthogonal to huge pages since our TLB supports caching TLB entries for multiple page sizes. Various prefetching mechanisms [31, 14] have been explored to fetch multiple TLB or PTE entries to hide page walk miss latency. However, the fundamental problem that the TLB capacity is not enough is not addressed in prior work. Hybrid TLB coalescing [54] aims to increase TLB coverage by encoding memory contiguity information and does not deal with managing cache capacity. Page Table Walk Aware Cache Management [3] uses a cache replacement policy to preferentially store page table entries on caches and does not use cache partitioning.

Cache Replacement: Recent cache replacement policy work such as DIP [59], DRRIP [29], SHiP [71] focuses on homogeneous data types, which means they are not designed to achieve the optimal performance when different data types of data (e.g., POM-TLB and data entries) coexist. Hawkeye cache replacement policy [6], also targets homogeneous data types, has a considerable hardware budget for LLC, and cannot be implemented for L2 data caches. EVA cache replacement policy [48] cannot be used in this case due to a similar problem.

Cache Partitioning: Cache partitioning is an extensively researched area. Several previous works ([60, 69, 66, 74, 47, 70, 30, 68, 21, 63, 75, 52, 17, 72, 41]) have proposed mechanisms and algorithms for partitioning shared caches with diverse goals of latency improvement, bandwidth reduction, energy saving, ensuring fairness and so on. However, none of these works take into account the adverse impact of higher TLB miss rates due to virtualization and context switches. As a result, they fail to take advantage of this knowledge to effectively address the TLB related cache congestion.

7. CONCLUSION

In this work, we study the problem of TLB misses and cache contention caused by context-switching between virtual machines. We show that with just two contexts, L2 TLB MPKI goes up by a factor of $6X$ on average across a variety of large-footprint workloads. We presented CSALT - a dynamic partitioning scheme that adaptively partitions the L2-L3 data caches between data and TLB entries. CSALT achieves page walk reduction of over 97% by leveraging the large L3 TLB. By designing a TLB-aware dynamic cache management scheme in L2 and L3 data caches, CSALT is able to improve performance. CSALT-CD achieves a performance improvement of 85% on average over a conventional system with L1-L2 TLBs and 25% over the POM-TLB baseline. The proposed partitioning techniques are applicable for any designs that cache page table entries or TLB entries in L2-L3 caches.

8. ACKNOWLEDGEMENTS

The researchers are supported in part by National Science Foundation grants 1337393. The authors would also like to thank Texas Advanced Computing Center (TACC) at UT Austin for providing compute resources. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF, or any other sponsors.

9. REFERENCES

[1] "AMD Nested Paging," http://developer.amd.com/wordpress/media/2012/10/NPT-WP-1%201-final-TM.pdf.

[2] "ARM1136JF-S and ARM1136J-S," http://infocenter.arm.com/help/topic/com.arm.doc.ddi0211k/ddi0211k_arm1136_r1p5_trm.pdf.

[3] "bpoe8-eishi-arima," http://prof.ict.ac.cn/bpoe_8/wp-content/uploads/arima.pdf, (Accessed on 08/24/2017).

[4] "Intel(R) 64 and IA-32 Architectures Optimization Reference Manual," http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.

[5] The Graph500 List. [Online]. Available: Graph500:http://www.graph500.org/

[6] C. L. Akanksha Jain, "Back to the Future: Leveraging Belady's Algorithm for Improved Cache Replacement," https://www.cs.utexas.edu/~lin/papers/isca16.pdf, 2016.

[7] Amazon, "Amazon EC2 - Virtual Server Hosting," https://aws.amazon.com/ec2/.

[8] A. Arcangeli, "Transparent hugepage support," in *KVM Forum*, vol. 9, 2010.

[9] T. W. Barr, A. L. Cox, and S. Rixner, "SpecTLB: A Mechanism for Speculative Address Translation," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ser. ISCA '11. New York, NY, USA: ACM, 2011, pp. 307–318. [Online]. Available: http://doi.acm.org/10.1145/2000064.2000101

[10] K. Begnum, N. A. Lartey, and L. Xing, "Cloud-Oriented Virtual Machine Management with MLN," in *Cloud Computing: First International Conference, CloudCom 2009, Beijing, China, December 1-4, 2009. Proceedings*. Springer Berlin Heidelberg, 2009.

[11] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '05. Berkeley, CA, USA: USENIX Association, 2005, pp. 41–41. [Online]. Available: http://dl.acm.org/citation.cfm?id=1247360.1247401

[12] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne, "Accelerating Two-dimensional Page Walks for Virtualized Systems," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIII. New York, NY, USA: ACM, 2008, pp. 26–35. [Online]. Available: http://doi.acm.org/10.1145/1346281.1346286

[13] A. Bhattacharjee, D. Lustig, and M. Martonosi, "Shared last-level TLBs for chip multiprocessors." in *HPCA*. IEEE Computer Society, 2011, pp. 62–63. [Online]. Available: http://dblp.uni-trier.de/db/conf/hpca/hpca2011.html#BhattacharjeeLM11

[14] A. Bhattacharjee and M. Martonosi, "Inter-core Cooperative TLB for Chip Multiprocessors," in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XV. New York, NY, USA: ACM, 2010, pp. 359–370. [Online]. Available: http://doi.acm.org/10.1145/1736020.1736060

[15] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '08. New York, NY, USA: ACM, 2008, pp. 72–81. [Online]. Available: http://doi.acm.org/10.1145/1454115.1454128

[16] J. Chang and G. S. Sohi, *Cooperative caching for chip multiprocessors*. IEEE Computer Society, 2006, vol. 34, no. 2.

[17] J. Chang and G. S. Sohi, "Cooperative Cache Partitioning for Chip Multiprocessors," in *ACM International Conference on Supercomputing 25th Anniversary Volume*. New York, NY, USA: ACM, 2014, pp. 402–412. [Online]. Available: http://doi.acm.org/10.1145/2591635.2667188

[18] X. Chang, H. Franke, Y. Ge, T. Liu, K. Wang, J. Xenidis, F. Chen, and Y. Zhang, "Improving virtualization in the presence of software managed translation lookaside buffers," in *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3. ACM, 2013, pp. 120–129.

[19] N. Ganapathy and C. Schimmel, "General purpose operating system support for multiple page sizes." in *USENIX Annual Technical Conference*, no. 98, 1998, pp. 91–104.

[20] I. Habib, "Virtualization with KVM," *Linux J.*, vol. 2008, no. 166, Feb. 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1344209.1344217

[21] W. Hasenplaugh, P. S. Ahuja, A. Jaleel, S. Steely Jr., and J. Emer, "The Gradient-based Cache Partitioning Algorithm," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, pp. 44:1–44:21, Jan. 2012. [Online]. Available: http://doi.acm.org/10.1145/2086696.2086723

[22] HP, "HPE Cloud Solutions," https://www.hpe.com/us/en/solutions/cloud.html.

[23] IBM, "SmartCloud Enterprise," https://www.ibm.com/cloud/.

[24] Intel, " Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide Part 1."

[25] Intel. Intel(R) 64 and IA-32 Architectures Software DeveloperâĂŹs Manual Volume 3A: System Programming Guide, Part 1. [Online]. Available: http://www.intel.com/Assets/en_US/PDF/manual/253668.pdf

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

[27] Intel, "5-Level Paging and 5-Level EPT," 2016, https://software.intel.com/sites/default/files/managed/2b/80/5-level_paging_white_paper.pdf,.

[28] P. Jääskeläinen, P. Kellomäki, J. Takala, H. Kultala, and M. Lepistö, "Reducing context switch overhead with compiler-assisted threading," in *Embedded and Ubiquitous Computing, 2008. EUC'08. IEEE/IFIP International Conference on*, vol. 2. IEEE, 2008, pp. 461–466.

[29] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3. ACM, 2010, pp. 60–71.

[30] R. Kandemir, Mahmut a nd Prabhakar, M. Karakoy, and Y. Zhang, "Multilayer Cache Partitioning for Multiprogram Workloads," in *Proceedings of the 17th International Conference on Parallel Processing - Volume Part I*, ser. Euro-Par'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 130–141. [Online]. Available: http://dl.acm.org/citation.cfm?id=2033345.2033360

[31] G. B. Kandiraju and A. Sivasubramaniam, *Going the distance for TLB prefetching: an application-driven study*. IEEE Computer Society, 2002, vol. 30, no. 2.

[32] D. Kaseridis, J. Stuecheli, and L. K. John, "Bank-aware dynamic cache partitioning for multicore architectures," in *Parallel Processing, 2009. ICPP'09. International Conference on*. IEEE, 2009, pp. 18–25.

[33] K. Kędzierski, M. Moreto, F. J. Cazorla, and M. Valero, "Adapting cache partitioning algorithms to pseudo-lru replacement policies," in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 1–12.

[34] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A Fast and Extensible DRAM Simulator," *IEEE Comput. Archit. Lett.*, vol. 15, no. 1, pp. 45–49, Jan. 2016. [Online]. Available: http://dx.doi.org/10.1109/LCA.2015.2414456

[35] A. Kivity, D. Laor, G. Costa, P. Enberg, N. Har'El, D. Marti, and V. Zolotarov, "Osv—Optimizing the Operating System for Virtual Machines," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, 2014, pp. 61–72. [Online]. Available: https://www.usenix.org/conference/atc14/technical-sessions/presentation/kivity

[36] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi: Large-scale Graph Computation on Just a PC," in *Conference on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2012, pp. 31–46.

[37] C. Li, C. Ding, and K. Shen, "Quantifying the Cost of Context Switch," in *Proceedings of the 2007 Workshop on Experimental Computer Science*, ser. ExpCS '07. New York, NY, USA: ACM, 2007. [Online]. Available: http://doi.acm.org/10.1145/1281700.1281702

[38] F. Liu and Y. Solihin, "Understanding the Behavior and Implications of Context Switch Misses," *ACM Trans. Archit. Code Optim.*, vol. 7, no. 4, pp. 21:1–21:28, Dec. 2010. [Online]. Available: http://doi.acm.org/10.1145/1880043.1880048

[39] H. Liu, "A Measurement Study of Server Utilization in Public Clouds," 2011, http://ieeexplore.ieee.org/document/6118751/media.

[40] 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 *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 190–200. [Online]. Available: http://doi.acm.org/10.1145/1065010.1065034

[41] R. Manikantan, K. Rajan, and R. Govindarajan, "Probabilistic Shared Cache Management (PriSM)," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ser. ISCA '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 428–439. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337159.2337208

[42] Z. A. Mann, "Allocation of virtual machines in cloud data centers—a survey of problem models and optimization algorithms," *ACM Comput. Surv.*, vol. 48, no. 1, pp. 11:1–11:34, Aug. 2015. [Online]. Available: http://doi.acm.org/10.1145/2797211

[43] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation Techniques for Storage Hierarchies," *IBM Syst. J.*, vol. 9, no. 2, pp. 78–117, Jun. 1970. [Online]. Available: http://dx.doi.org/10.1147/sj.92.0078

[44] Y. Mei, L. Liu, X. Pu, S. Sivathanu, and X. Dong, "Performance analysis of network I/O workloads in virtualized data centers," *IEEE Trans. Services Computing*, vol. 6, no. 1, pp. 48–63, 2013. [Online]. Available: https://doi.org/10.1109/TSC.2011.36

[45] X. Meng, C. Isci, J. Kephart, L. Zhang, E. Bouillet, and D. Pendarakis, "Efficient resource provisioning in compute clouds via VM multiplexing," in *Proceedings of the 7th International Conference on Autonomic Computing*, ser. ICAC '10. New York, NY, USA: ACM, 2010, pp. 11–20. [Online]. Available: http://doi.acm.org/10.1145/1809049.1809052

[46] Microsoft, "Microsoft Azure," https://www.microsoft.com/en-us/cloud-platform/server-virtualization.

[47] M. Moreto, F. J. Cazorla, A. Ramirez, and M. Valero, "Transactions on High-performance Embedded Architectures and Compilers III," P. Stenström, Ed. Berlin, Heidelberg: Springer-Verlag, 2011, ch. Dynamic Cache Partitioning Based on the MLP of Cache Misses, pp. 3–23. [Online]. Available: http://dl.acm.org/citation.cfm?id=1980776.1980778

[48] D. S. Nathan Beckmann, "Maximizing Cache Performance Under Uncertainty," http://people.csail.mit.edu/sanchez/papers/2017.eva.hpca.pdf, 2017.

[49] J. Navarro, S. Iyer, P. Druschel, and A. Cox, "Practical, transparent operating system support for superpages," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 89–104, 2002.

[50] Oracle. Translation Storage Buffers. [Online]. Available: https://blogs.oracle.com/elowe/entry/translation_storage_buffers

[51] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank citation ranking: Bringing order to the web," 1999.

[52] A. Pan and V. S. Pai, "Imbalanced Cache Partitioning for Balanced Data-parallel Programs," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-46. New York, NY, USA: ACM, 2013, pp. 297–309. [Online]. Available: http://doi.acm.org/10.1145/2540708.2540734

[53] M.-M. Papadopoulou, X. Tong, A. Seznec, and A. Moshovos, "Prediction-based superpage-friendly TLB designs," in *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*. IEEE, 2015, pp. 210–222.

[54] C. H. Park, T. Heo, J. Jeong, and J. Huh, "Hybrid tlb coalescing: Improving tlb translation coverage under diverse fragmented memory allocations," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 2017, pp. 444–456.

[55] B. Pham, J. Veselý, G. H. Loh, and A. Bhattacharjee, "Large Pages and Lightweight Memory Management in Virtualized Environments: Can You Have It Both Ways?" in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48. New York, NY, USA: ACM, 2015, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/2830772.2830773

[56] B. Pham, J. Vesely, G. H. Loh, and A. Bhattacharjee, "Using TLB Speculation to Overcome Page Splintering in Virtual Machines," 2015.

[57] G. C. Platform, "Load Balancing and Scaling." [Online]. Available: https://cloud.google.com

[58] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive Insertion Policies for High Performance Caching," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ser. ISCA '07. New York, NY, USA: ACM, 2007, pp. 381–391. [Online]. Available: http://doi.acm.org/10.1145/1250662.1250709

[59] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," in *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2. ACM, 2007, pp. 381–391.

[60] M. K. Qureshi and Y. N. Patt, "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 39. Washington, DC, USA: IEEE Computer Society, 2006, pp. 423–432. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2006.49

[61] Rackspace, "OPENSTACK - The Open Alternative To Cloud Lock-In," https://www.rackspace.com/en-us/cloud/openstack.

[62] J. H. Ryoo, N. Gulur, S. Song, and L. K. John, "Rethinking TLB Designs in Virtualized Environments: A Very Large Part-of-Memory TLB," in *Computer Architecture, 2017 IEEE International Symposium on*, ser. ISCA '17. ACM, 2017. [Online]. Available: http://lca.ece.utexas.edu/pubs/isca2017.pdf

[63] D. Sanchez and C. Kozyrakis, "Vantage: Scalable and Efficient Fine-grain Cache Partitioning," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ser. ISCA '11. New York, NY, USA: ACM, 2011, pp. 57–68. [Online]. Available: http://doi.acm.org/10.1145/2000064.2000073

[64] A. W. Services, "High Performance Computing," https://aws.amazon.com/hpc/,.

[65] SUN, "The SPARC Architecture Manual," http://www.sparc.org/standards/SPARCV9.pdf.

[66] K. T. Sundararajan, T. M. Jones, and N. P. Topham, "Energy-efficient Cache Partitioning for Future CMPs," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '12. New York, NY, USA: ACM, 2012, pp. 465–466. [Online]. Available: http://doi.acm.org/10.1145/2370816.2370898

[67] V. Vasudevan, D. G. Andersen, and M. Kaminsky, "The Case for VOS: The Vector Operating System," in *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, ser. HotOS'13. Berkeley, CA, USA: USENIX Association, 2011, pp. 31–31. [Online]. Available: http://dl.acm.org/citation.cfm?id=1991596.1991638

[68] P.-H. Wang, C.-H. Li, and C.-L. Yang, "Latency Sensitivity-based Cache Partitioning for Heterogeneous Multi-core Architecture," in *Proceedings of the 53rd Annual Design Automation Conference*, ser. DAC '16. New York, NY, USA: ACM, 2016, pp. 5:1–5:6. [Online]. Available: http://doi.acm.org/10.1145/2897937.2898036

[69] R. Wang and L. Chen, "Futility Scaling: High-Associativity Cache Partitioning," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-47. Washington, DC, USA: IEEE Computer Society, 2014, pp. 356–367. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2014.46

[70] W. Wang, P. Mishra, and S. Ranka, "Dynamic Cache Reconfiguration and Partitioning for Energy Optimization in Real-time Multi-core Systems," in *Proceedings of the 48th Design Automation Conference*, ser. DAC '11. New York, NY, USA: ACM, 2011, pp. 948–953. [Online]. Available: http://doi.acm.org/10.1145/2024724.2024935

[71] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely Jr, and J. Emer, "SHiP: Signature-based hit predictor for high performance caching," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2011, pp. 430–441.

[72] Y. Xie and G. H. Loh, "PIPP: Promotion/Insertion Pseudo-partitioning of Multi-core Shared Caches," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA '09. New York, NY, USA: ACM, 2009, pp. 174–183. [Online]. Available: http://doi.acm.org/10.1145/1555754.1555778

[73] C.-H. Yen, "SOLARIS OPERATING SYSTEM HARDWARE VIRTUALIZATION PRODUCT ARCHITECTURE," 2007. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=3F5AEF9CE2ABE7D1D7CC18DC5208A151?doi=10.1.1.110.9986&rep=rep1&type=pdf

[74] C. Yu and P. Petrov, "Off-chip Memory Bandwidth Minimization Through Cache Partitioning for Multi-core Platforms," in *Proceedings of the 47th Design Automation Conference*, ser. DAC '10. New York, NY, USA: ACM, 2010, pp. 132–137. [Online]. Available: http://doi.acm.org/10.1145/1837274.1837309

[75] M. Zhou, Y. Du, B. Childers, R. Melhem, and D. Mossé, "Writeback-aware Partitioning and Replacement for Last-level Caches in Phase Change Main Memory Systems," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, pp. 53:1–53:21, Jan. 2012.