Set-Associative Virtual Memory Regions

ASPLOS Submission #329– Confidential Draft – Do Not Distribute!

Abstract

As the computing industry embraces heterogeneity, a key research challenge is the question of how to integrate hardware accelerators in modern platforms without impacting the conventional virtual memory abstractions modern complex software stacks rely on. Unfortunately, VM support for accelerators is fundamentally different from CPU's due to prohibitive TLB reach requirements, long-latency page table walks, and tight area and power budgets. Moreover, recent proposals for accelerator VM break conventional VM abstractions with intrusive OS changes to facilitate address translation.

We propose spryVM, a set-associate VM for in-memory workloads. We show that maintaining the portion of the address space accelerators operate on set-associatively has little impact on page fault traffic, but reduces address translation hardware requirements. SpryVM leverages this observation to improve TLB hit rates by desigining TLBs targeted to accelerator-local partitions, and by breaking page table walk-data fetch serialization upon TLB misses. SpryVM achieves within 1.2% and 0.6% of ideal translation in scenarios where working sets are memory-resident and exceed the available memory capacity, respectively. Finally, we implement spryVM in stock Linux, showing that these benefits are achievable while retaining conventional VM abstractions and with only modest changes to existing VM software stacks.

1. Introduction

As the computing industry embraces hardware specialization, research questions pertaining to the system integration of accelerators in modern platforms are becoming increasingly important. One such question is that of how to expose the virtual memory (VM) abstraction to accelerators. While initial solutions allow accelerators to have direct physical access to memory, the consequent lack of protection, isolation and paging significantly complicates the software stack and creates vulnerabilities. More recent studies have argued that the conventional VM abstraction and a global address space programming model is best for software development, enabling "a pointer is a pointer everywhere" semantics [7, 25, 46, 48, 56], extending memory protection to accelerators, and obviating the need for manual inter-CPU-accelerator data marshalling.

Unfortunately, it is challenging to extend conventional VM to accelerators in an efficient manner. While general-purpose cores rely on large hiearchical TLBs to achieve satisfactory VM performance, such large structures are ill-suited to resource-constrained accelerators. This is true for a wide spectrum of accelerators ranging from programmable GPGPUs [46, 48] to graph processing hardware [25] to even more area-contrained processing in/near memory techniques [47].

In response, recent studies propose changes to conventional TLB hardware and the memory allocation/management stack for efficient address translation and page management on accelerators [7, 46, 48]. Among the most promising techniques are those that obviate the need for large TLBs by relying on translation contiguity, i.e., situations where large contiguous ranges of virtual pages are mapped to a corresponding range of spatially-adjacent physical pages. Contiguous translations can be compressed or coalesced by TLBs into individual entries, reducing address translation pressure. While some of these techniques rely on translation contiguity that OSes may serendipitously generate [12, 17, 43, 44], the most successful ones are those that rely on changes to the OS's memory allocation path so that it produces vast swaths of translation contiguity [25]. Unfortunately, these techniques, along with similar ones proposed for CPUs [10, 20], sacrifice OS flexibility to generate translation contiguity by requiring, for example, identity mapping where virtual and physical addresses must be identical (which can be difficult to achieve in real-world cloud settings where many applications utilize a machine), or at-allocation contiguity generation (which can be difficult to achieve in real-world systems with fragmented memory).

In contrast, we observe that traditional VM requires large TLBs because it is *fully-associative*; i.e., a virtual page can map to any physical frame, without restrictions. On the other hand, techniques that cut TLB requirements by creating massive translation contiguity [10, 20, 25] impose strong restrictions on which physical frames to assign to a virtual page, reminiscent of the notion of *direct-mapping*. Our approach is to find the sweet spot between these techniques and propose *set-associative* VM, where a virtual page can map to one among a set of page frames. We dub this approach spryVM (Set-Associative Regions of Your Virtual Memory).

SpryVM *slightly* restricts VM associativity so that a virtual address uniquely identifies the region of the physical address space (a memory chip or partition) that holds the data. Once the target memory region is identified, SpryVM performs translation on the memory side, using local per-region TLBs situated next the corresponding memory region. In doing so, SpryVM solves two key problems. First, it facilitates area-efficient TLBs with high hit rates because separate TLBs are responsible for disjoint parts of the physical address space, reducing their reach requirements. Second, it dramatically cuts miss penalties by co-locating translation and data in TLB's physical partition, overlapping translation with data fetch. We

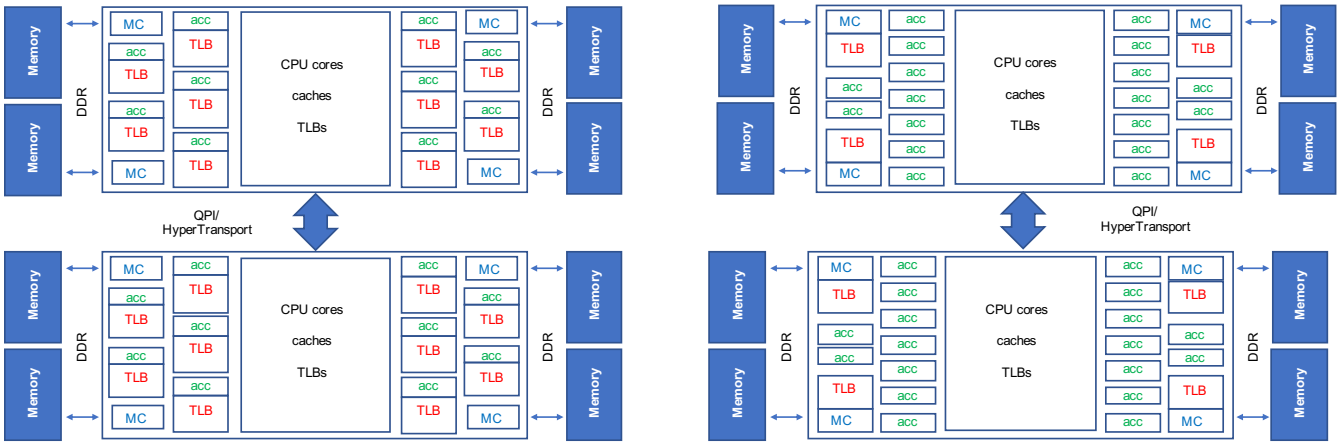

Figure 1: Overview of conventional (left) and SpryVM system (right).

also show that the OS support necessary to achieve this approach is compatible with existing kernels and does not sacrifice flexibility in the mould of prior work [10,25]. To showcase spryVM's effectiveness, we contribute the following:

• A study of the VM associativity needs of modern server workloads. We are inspired by previous work on caches, which shows how associativity affects capacity, conflict, and compulsory misses [26]. We find that full VM associativity is often unnecessary, as most misses—page faults—are either compulsory or capacity, and are hence insensitive to associativity.

• We develop a working prototype of spryVM by integrating support for set-associative VM in stock Linux, and show that these changes are readily implementable. Moreover, we selectively apply set associativity to *regions* of the virtual address space that benefit from this approach. In so doing, we leave full-associativity for regions where it is better for performance, and retain cross-compatibility with existing systems

• SpryVM, a translation mechanism which restricts VM associativity so that a virtual address uniquely identifies a memory chip/partition, allowing memory accesses to be issued as soon as the virtual address is known. Each memory partition features an MMU with a TLB hierarchy, whose location next to the data allows data fetch and translation to be almost entirely overlapped (for TLB hits and misses). Each MMU's TLB hierarchy serves only its memory partition, enabling graceful performance scaling across memory chip/partition counts.

• A comparison of spryVM with well-known translation mechanisms. SpryVM improves performance by up to 26% and 15% for in-memory and out-of-memory scenarios respectively, achieving almost-perfect zero-overhead address translation.

2. Background and Motivation

2.1. Goals for VM in Heterogeneous Systems

When buiding VM support for accelerators, it is important to identify design goals for the its operation. Like prior work [25], we identify the following as key goals in our design:

• **Programmability.** The widespread adoption of accelerators rests on the usability and familiarity of their programming models. Unified virtual memory between CPUs and accelerators are one way of achieving this in a manner that simplifies data sharing, eliminating the need for hand-managed data copying and marshaling. Ideally, we wish to preserve all the benefits typically associated with VM like memory protection and isolation, and flexibility of sharing parts of the address space among processes.

• **Flexibility.** Traditional VM imposes no restrictions on virtual-to-physical page mapping relationships. This is valuable to support any level of system memory fragmentation, application multi-tenancy, memory allocation strategies transparent to programmers, as well as the integration of features such as paging and copy-on-write (CoW).

• **Safety and Security.** Direct access to physical memory is generally not acceptable nor desirable. Such memory management approaches cannot prevent malicious or erroneous memory accesses and prohibits sharing accelerators across different processes with proper isolation [25]. Furthermore, the entropy in address mapping must be as high as possible

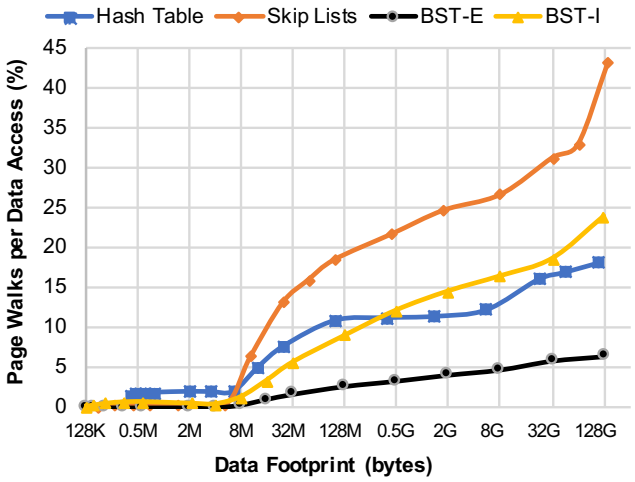

Figure 2: Frequency of page walks as a function of memory size.

Table 1: Comparison of SpryVM with previous approaches for reducing virtual memory overhead. For protection/security, we show tick/cross-marks for each of those cases. Coarse-grained protection is indicated with a cross, as is lower entropy in virtual address bits.

	Programmability	Performance and Efficiency	Flexibility	Protection/Security
Multi-page mappings [43, 44]	✓	✗	✓	✓ ✓
Transparent Huge Pages [29]	✓	✗	✓	✗ ✓
Libhugetlbfs [50]	✗	✗	✓	✗ ✓
Direct Segments [10]	✗	✓	✗	✗ ✗
Redundant Memory Mappings [31]	✓	✗	✗	✗ ✗
Direct-mapped Mappings [25, 47]	✓	✓	✗	✗ ✗
SpryVM (Our Approach)	✓	✓	✓	✓ ✓

to reduce vulnerability to security attacks.

- **Performance and Efficiency.** Crucially, all the goals listed thus far must be achievable without excessive performance or area overheads in hardware. In other words, address translation must provide near-zero performance overheads regardless of application working set and locality patterns, and must do so under tight area and power constraints (particularly for area-constrained accelerators).

2.2. Shortcomings of Modern MMU Hardware

The ever-increasing memory needs of modern software has given rise to scale-out systems with large memories with low-latency access to data [10, 19, 32, 58]. The advent of increasing memory sizes is problematic for both address translation reach and latency, as we next discuss.

2.2.1. TLB Reach. Several studies have established the difficulties of building TLBs with sufficient capacity or *reach* to cover increasing physical memory sizes [10, 25, 44]. Industry's approach has been to aggressively grow TLBs – e.g., Intel has been doubling CPU TLBs from Sandybridge to Skylake architectures – and pay the cost of increased area/power. Nevertheless, despite TLBs with thousands of entries, the poor locality of access in emerging server workloads make TLB miss rates problematic. Figure 2 captures this effect by quantifying TLB miss rates as we vary the memory footprint of several of our workloads on an Intel Broadwell chip with 1.5K-entry L2 TLBs (see Section 6 for details). Despite Broadwell's large thousand-entry TLBs, TLB miss rates increase dramatically with larger memory footprints, corroborating results from prior work [10]. Naturally, this poses problems for accelerator TLBs; for example, GPUs integrate massive multi-thousand-entry TLBs too [36, 56], but there is widespread consensus that this approach is not viable for other more area-constrained accelerators [25, 47]. Perhaps even more troublingly, while other techniques like large pages can offer partial relief in some cases, they present their own set of challenges because they offer only coarse-grained protection [45], they can be hard to form on fragmented systems [34], they have poor NUMA support [23], and they require complex TLB hardware for concurrent page size support [17]. For all these reasons, transparent support for large pages in OSes like Linux only apply to 2MB pages (and not other sizes like 1GB) even after decades of research [21]. Practically, vendors implement multi-thousand-entry TLBs for the worst-case scenario when base 4KB pages dominate. For the successful adoption, we concur with recent work [10, 25, 31, 44] that alternate approaches (complementary to large pages) are needed.

2.2.2. Increasing Page Table Walk Latency. In addition to TLB reach, the penalty of a TLB miss is critical to address translation performance. For this reason, CPUs are equipped with dedicated MMU caches to accelerate page table walks [8, 12]. Unfortunately, even the presence of MMU caches cannot mitigate high page table walk latencies in the context of accelerators. The key culprit is that heterogeneous sytems with accelerators are usually integrated with NUMA memories and require long-latency lookups across sockets/chips, etc. Even perfect MMU caches require at least one memory reference per page table walk and recent work shows that this single reference can dramatically exacerbate address translation costs for accelerators [46, 47].

2.3. Prior Approaches

In response to the problems detailed in the last section, several studies have proposed a range of techniques to improve address translation. Table 1 summarizes these techniques as well as their programmbility and flexibility attributes. We detail these techniques further but note that in general, all approaches change VM software and sacrifice aspects of traditional VM flexibility (to varying degrees) to achieve effective address translation. SpryVM's goal is to find a better compromise between retaining traditional VM benefits at the software level, while also enabling more efficient TLB hardware.

Multi-page mappings. Studies have exploited contiguity naturally generated by the buddy allocator and the memory compactor. COLT [44] and clustered [43] TLBs coalesce 4-8 page translations into a single TLB entry, as long as their physical locations are contiguous. Although TLB reach improves, they cannot cover the entirety of a large memory system of tens or hundreds of GBs [20]. Equally problematically, these techniques rely on contiguity that *might* be generated but does not have to be. Achieving such contiguity is challenging in highly-loaded cloud systems where memory can be fragmented [44].

Segments. These approaches use variable-size segments instead of fixed page-based translations [10, 31, 41]. Unfortunately, the effectiveness of these techniques relies on heavy changes to the OS's allocation path with at-allocation contiguity generation (i.e., eager paging). Furthermore, direct segments require applications to explicitly allocate a segment at startup, while redundant memory mappings (RMMs) [31]. While effective in many scenarios, their reliance on at-allocation contiguity generation means that there are situations on highly fragmented systems where segments/ranges may be difficult to generate. Additionally, because they are flexible in the number of contiguous translations they can coalesce in a TLB entry, they require highly associative TLBs, which poses power/area overheads.

Direct-mapped segments. Recent work on devirtualized virtual memory extends the previous line of work on segments specifically in the context of accelerators [25]. The approach is to identity map virtual and physical pages for large swaths of the application footprint; this enables just one TLB entry to essentially cover an entire segment. While address translation overheads are largely eliminated with this approach, identity mappings may be optimistic in highly-loaded multi-tenant cloud scenarios with memory fragmentation. Similar restrictions apply to other "direct-mapped" approaches propsed for near-memory accelerators [47]. Moreover, these approaches preclude mechanisms like copy-on-write (COW), which are widely used in fork system calls, and their impact on schemes that require address bit entropy (like ASLR) require further examination before widespread adoption.

3. Our Approach

Fundamentally, all prior techniques present the following maxim – one can either implement large power-hungry TLBs to accommodate all the benefits of fully-flexible VM, or one can get away with smaller TLBs as long as software flexibility is sacrificed. In the first scenario, large TLBs are essential because VM is fully associative. In the second scenario, software flexibility is sacrificed because VM imposes restrictions reminiscent of direct mapping. SpryVM's contribution is to showcase, however, that there is a third regime of operation between these two extremes, where we implement set-associative VM to continue enjoying the benefits of flexible VM without requiring large TLBs.

The core idea behind SpryVM is (partly) captured in Figure 1. With SpryVM, VM associativity is restricted in that a virtual address is guaranteed to map to a physical address in a specific memory "region". This memory region could be a socket or a memory channel. This effective set-associativity provides two benefits. First, it enables us to replace large per-CPU/accelerator TLB that need to address the entire physical address space with memory-side TLBs responsible for mapping only their specific memory regions. Because memory regions represent just a subset of the total physical address space, memory-side TLBs can achieve high hit rates despite being smaller in size. Second, set-associativity also permits us to dramatically reduce TLB miss penalty by allowing co-location of data and the corresponding translations in the same memory regions, overlapping their lookup. The confluence of these two benefits is a combination of better performance and reduction in TLB resources.

A factor in our approach is that implementing set-associativity in a commodity OS requires only modest changes to existing memory allocation/management code-paths. Furthermore, we have found that in practice, we can achieve good address translation performance by only modestly reducing associativity from the default fully-associative VM approach. Consequently, SpryVM implements unified CPU-accelerator address spaces without forcing coarser-grained memory protection, compromising on entropy in virtual address bits, or requiring power/area-hungry TLBs. The fact that SpryVM does not rely on contiguity also means that it can continue to operate efficiently in fragmented memory settings where translation contiguity or identity mappings may be hard to generate. Furthemore, having only a modest drop from fully-associative designs means that optimizations like copy-on-write can, for all practical purposes, be supported (we will show that the number of CoW pages is limited only by the size of the memory region, which can be thousands of pages in practice).

4. SpryVM: Hardware Support

We compare the hardware necessary for SpryVM with conventional TLBs. We first walk the reader through a translation operation in a conventional memory management mechanism and then we present the same operation in SpryVM.

4.1. Baseline Address Translation

Figure 3 illustrates the timeline of conventional address translation. In the conventional case, a TLB is first probed by the acclerator it resides in. Consider the case of a TLB miss on an accelerator without caches residing close to memory, as considered in recent work [25, 47]. The accelerator's hardware page table walker identifies the physical address of the page table and initiates a lookup of it. Page tables can be implemented in many ways, e.g., radix tree, inverted, hashed, etc. Consider the scenario where either because of the page table structure or the presence of MMU caches [8, 12], the entire page table walk amounts to a single memory reference lookup. This lookup can result in a probe of any location in memory. Consequently, we may need to traverse the NoC and off-chip interconnect to locate the memory channel hosting the desired page table entry. Once the memory channel is identified, its memory controller accesses the page table entry's location in DRAM, and then page table entry, which holds the virtual-to-physical translation information is found. The page table entry has to return to the accelerator, again prompting NoC and off-chip interconnect traversal. Once the page table entry

is returned to the accelerator, TLB fill occurs. This part of the timeline constitutes th translation path.

Subsequently, the accelerator uses the page table entry to calculate the physical address of the data fetch. A set of NoC and off-chip interconnects are traversed to locate the memory channel that holds the piece of data, and its memory controller access DRAM. Once the memory controller receives the data, the data has to traverse the same NoC and off-chip interconnects to return to the accelerator. This part of the timeline constitutes the data fetch.

Figure 3 shows that conventional address translation is expensive. First, the TLB's accelerator-side integration means that it must address all of physical memory, which can elevate its miss rate. Second, when these misses occur, translation and data paths may not overlap. Hence, translation walk latency is on the critical path of the memory access.

4.2. SpryVM Address Translation

SpryVM's set-associative organization means that a virtual page can map to a physical frame from specific memory regions only. This permits us to relieve the accelerator from having to translate virtual pages to physical frames and instead, requires that accelerators only calculate the physical memory region that a virtual page corresponds to. This can be accomplished using a simple hash function (e.g., implementable using bit masking combinational logic) and the request can be routed to the particular region. This means that rather than placing per-accelerator TLBs, we only need per-region TLBs mapping the physical frames within that region. Similarly, we can also place a page table within the memory region for physical pages only belonging to that region. Per-region TLBs can be made much smaller than per-accelerator TLBs, which have to cover all of physical memory. Per-region page tables reduce page table walk latency as they are co-located in the same region as the data.

Figuer 4 shows the timeline of events with a TLB hit in SpryVM. The virtual address uniquely identifies a memory region and traverses the NoC and off-chip interconnect to locate the appropriate memory channel. Then, a simple multiplexer (not shown in the figure) distinguishes virtual and physical address requiests, and sends the former to the memory channel's TLB. The TLB holds the translation. From this page frame, we can calculate the physical address of the data, completing the translation path. Once this completes, data fetch commences. Unlike the baseline case, note that spryVM can overlap translation with data fetch completely because both the TLB and data are co-located in the same region. This means that the translation overhead is exactly zero, equal to an ideal translation mechanism.

With SpryVM, TLBs misses are rarer but can occur. When they do, as shown in Figure 5, the virtual address identifies the appropriate memory controller, which can be reached by traversing the NoC and off-chip interconnects. Suppose there is a TLB miss; the page table is also resident in the same memory region. Therefore, unlike conventional translation, there is no need to traverse the NoC and off-chip interconnects for the page table lookup. Once the page table entry arrives from the DRAM arrays and the TLB fill operation completes, translation finishes. Then, the memory controller issues the complete physical address to locate the target piece of data. When data returns from the DRAM array, it is sent back to the accelerator by traversing the networks, finishing the data path.

With SpryVM, page walks are cheaper than in conventional translation as the time to locate the appropriate memory channel is part of both the translation path and data path (essentially removing this part of the cost of address translation). Page table walks do not involve network communication as the page table entries are all located in the same memory channel. The longer the latency of the network is with respect to the DRAM accesses, the lower overhead of page table walks in SpryVM. This behavior particularly benefits large memory systems, as the need for scaling out the memory for large-memory machines increase the average distance to the memory channels, as well as systems with lower DRAM latencies (e.g., 3D-stacked memories).

5. SpryVM: Operating System Support

One of SpryVM's advantages is that is is readily-implementable and compatible with commodity OSes. To demonstrate this, we have prototyped SpryVM in stock Linux (4.10) by implementing set-associativity in the memory management sections of the kernel. Additionally, we have implemented SpryVM's page table in the kernel as an inverted page table, although other implementations are also just as readily possible. Overall, these and other ancillary modifications were compatible with the existing kernel. A measure of this is that when use `git diff` to track our changes, we find the following: `20 files changed, 266 insertions(+), 43 deletions(-)`. Key aspects of our changes involved:

Set-associativity using NUMA page allocation. We were able to greatly simplify our set-associativity implementation by leveraging existing Linux code built to bind processes to NUMA nodes (i.e., *membind*). In other words, we observed that the notion of mapping only a fixed set of physical frames to a virtual page is analogous to the idea of assigning physical frames from only select NUMA nodes. Consequently, to build a system with only slightly reduced associativity, we conceptually treat subsets of the virtual address space as being bound to specific NUMA nodes. Therefore SpryVM can be implemented in a manner that reuses *membind* code paths such that each memory set is treated as a NUMA node. Note that this implementation is only possible because SpryVM reduces associativity only *by a small factor*, as opposed to reducing it *to a small number* (e.g., dirrect mapped or 4-way associative [47]).

Anonymous and file-backed pages. Our SpryVM implementation has to treat anonymous and file-backed pages differently

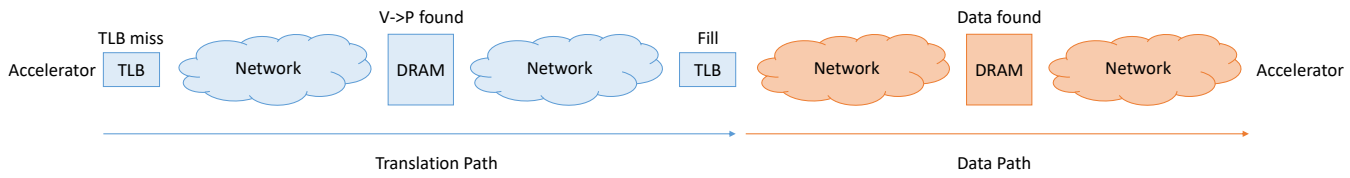

Figure 3: Timeline of events of an accelerator's memory reference that misses in the TLB on conventional translation.

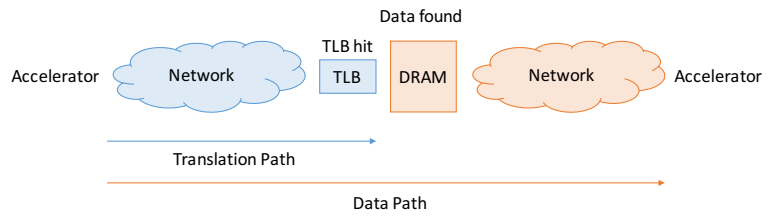

Figure 4: Timeline of events of an accelerator's memory reference that hits in the TLB on SpryVM.

because the former is generally private to a process, while the latter can (and is) often shared among processes.

Consider the case of anonymous pages. On page faults, the OS must calculate a corresponding NUMA node ID when allocating physical pages. To do this, we embed logic in the kernel to apply our hash function on the faulting virtual page before obtaining a free page from the NUMA node identified by the hash function. We found it most efficient to embed this logic in Linux's `alloc_pages_vma()`, which serves as the entry point of all user space allocations. Within this function, we calculate a memory set index (stored in the NUMA node mask), derive the desired zone list, and pass this information to the page allocation function, (`__alloc_pages_nodemask()`). From this point onwards, we recycle existing Linux code paths for anonymous pages. The basic operations are shown in Algorithm 1.

File-backed pages, on the other hand, present more subtle implementation challenges because they can be shared among different processes. Because of this, there is no guarantee that multiple processes use the same virtual addresses to access these pages, meaning that the output of our hash function may differ depending on the faulting process. Consequently, we use the page offset in a file as the memory set ID for each page. The rationale is that page offsets indicate the position of the page residing in the file to which it maps. Therefore, using page offsets enables us to allocate file-backed pages from different memory sets via our index scheme. We do need to adjust virtual address assignment in the kernel (via `get_unmapped_area()`) to accomplish this however. We do this by aligning the beginning of the given virtual address to match memory set 0's index. Thus, each process maps files starting from memory set 0. These algorithms are shown in Algorithm 2.

Inverted page table design. We adopt the same inverted page design as the hashed page tables in IBM Power systems. We reserve memory for each inverted page table in each individual NUMA node and insert/invalidate/update inverted page table entries whenever CPU page tables change in the same hook functions (e.g. `update_mmu_cache()`). To guarantee coherence in translation information between CPUs and accelerators, each inverted page table entry has a valid bit atomically accessed by both CPUs (for inverted page table modification) and accelerators (for reading).

Algorithm 1 Anonymous page allocation scheme

```
1: procedure ALLOC_PAGES_VMA(addr_t vaddr)
2:     mem_set_id ← MEM_SET_INDEX_HASH(vaddr)
3:     page ← alloc_pages_node(vaddr, mem_set_id)
4:     return page
```

Algorithm 2 File-backed page allocation scheme

```
1: procedure ALLOC_PAGE_CACHE(pgoff_t page_offset)
2:     mem_set_id ← page_offset
3:     page ← alloc_pages_node(vaddr, mem_set_id)
4:     return page
5: procedure GET_UNMAPPED_AREA(size_t mmap_size,
   pgoff_t page_offset)
6:     vaddr ← FIND_HOLE_IN_VSPACE(mmap_size)
7:     vaddr ← ALIGN_TO_OFFSET(vaddr, page_offset)
8:     return vaddr
```

6. Experimental Methodology

Like most recent work on VM [9, 10, 12, 40, 43, 44, 52], we use a combination of real-hardware measurements, trace-driven functional simulation, analytical models, and linux kernel prototypes.

6.1. Workloads

To study the impact of associativity on general-purpose software, we use a suite of popular server workloads, shown in Table 2, in a variety of setups. First, we use them to study

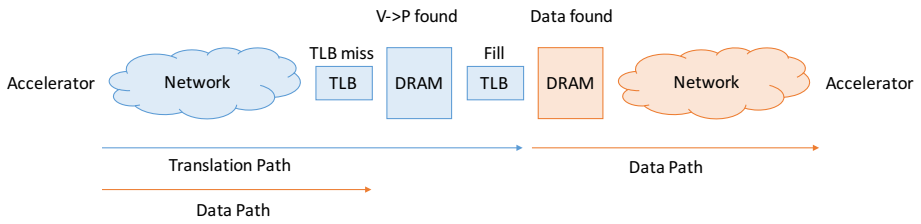

Figure 5: Timeline of events of an accelerator's memory reference that misses in the TLB on SpryVM.

Table 2: Server Workload description.

Workload	Description
Cassandra	NoSQL data store running Yahoo's YCSB.
Memcached	Cache store running Twitter-like workload [35].
TPC-H	TPC-H on MonetDB column store (Q1-Q21).
TPC-DS	TPC-DS on MonetDB column store (Queries of [33]).
MySQL	SQL DBMS running Facebook's LinkBench [1].
RocksDB	Store engine running Facebook benchmarks [4].

the very limits of reducing associativity using a 3C page fault model and Pin traces. For practical reasons these workloads are tuned to fit in 8GB memory for this experiment. Then we study the behavior of the server workloads on real hardware with our modified linux kernel (v4.10) with set-associative memory (which we plan to open-source), using more realistic dataset sizes (16-32GB), in both in-memory and out-of-memory setups.

To study the TLB behavior at very large datasets (128GB), we use data traversal applications present in the ASCYLIB [18] suite, which contains state-of-the-art implementations of hash tables, extarnal and internal binary trees, and skip lists, which are the core of many of the server workloads, such as Memcached and RocksDB. We choose these workloads because of their minimal data locality and instruction-level parallelism, and the fact that they are stressing conventional general-purpose CPU architectures and hence favor custom hardware [25,28,33,47]. Similar to prior work [47], we present the results for four representative implementations for space reasons.

6.2. TLB studies

To study the impact on memory size on TLB performance (Figure 2), we perform measurements on real hardware using the *perf* tool during a 10min execution window. To study the impact of TLB size, we collect memory traces for 32GB and 128GB datasets using Pin [37], with each trace containing 1B instrutions, which is more than enough for the TLB sensitivity experiments up to a few thousand entries. We use the traces to probe a set-associative TLB structure while varying its size and organization, after validating the baseline TLB model against the measurements on real hardware.

6.3. Limits of Associativity Reduction

As our approach intends to find a middle-ground between fully associative (i.e., conventional translation) and direct mapping [25, 47], we aim to fill the research void of what is the level of associativity needed. Interestingly enough, there exists no study on VM associativity, unlike caches, which are similar in some organizational aspects and for which such a study has existed for three decades [27]. To study the VM associativity, we employ the well-established 3C model—initially developed for caches [27]. In this context, associativity means the number of possible locations (page frames) a given virtual page can map to. This model classifies misses (i.e., page faults) into three categories: conflict misses (when too many active pages map to a fraction of the sets), capacity misses (due to limited memory size), and compulsory misses (upon the first access to a page).

We collect memory traces using Pin [37]. For workloads with fine-grained operations (i.e., Memcached, RocksDB, MySQL, and Cassandra), the traces contain the same number of instructions as the application executes in 60 seconds without Pin. For analytics workloads (i.e., TPC-H & TPC-DS), we instrument the entire execution. We extract the ASID bits and virtual address of each memory access, concatenate both [11, 59], and use it to probe a set-associative memory structure. For the associativity experiments, we tune all the workloads to have a resident set size (RSS) of 8GB. In other words, the allocated physical memory for all the processes of a given workload is 8GB. For single-process runs, a single process has an RSS of 8GBs. For two-process runs, each of the processes has an RSS of 4GB. The same scaling applies for the runs with four and eight processes. To vary the memory size to working set size ratio, we vary the size of the set-associative memory structure.

We collect the traces on a dual-socket server CPU (Intel Xeon E5-2680 v3) with 256GB of memory, using the Linux 3.10 kernel and Google's TCMalloc [2]. Address space randomization (ASLR) is enabled in all experiments.

6.4. Performance

Full-system simulation for TLB misses and page faults is not practical, as these events occur less frequently than other micro-architectural events (e.g., branch mispredictions). Hence, we resort to the CPI models often used in VM research [13,40,52] to sketch the performance gains. These prior

studies report performance as the reduction in the translation-related cycles per instruction. As CPI components are additive, this metric is valid irrespective of the workload's baseline CPI. We further strengthen this methodology by studying the CPI savings on all memory operations, not only on translation (as we overlap translation and data fetch operations). Our model thus captures both the translation and data fetch cycles, which together constitute the largest fraction of the total CPI in data structure traversals [47]. The CPI is measured by applying fixed penalties to the TLB miss rates obtained with the PIN traces.

7. Evaluation

In this section, we evaluate SpryVM with conventional translation using 4KB and 2MB pages, and qualitatively with the other prior work.

7.1. TLB Sensitivity

7.1.1. 32GB working set. Figure 6 shows the TLB miss ratio as we increase the number of entries, of the four data structure traversals, hash table, skip list, binary search tree internal, and binary search tree external, with a working set size of 32GBs. Each graph plots three lines. The blue line represents the conventional translation mechanism using 4KB pages. The orange line the same mechanism but using 2MB pages. The gray bar represent the TLB miss ratio of SpryVM, which uses 4KB pages.

4KB pages: For all four data structures, SpryVM systematically beats conventional translation with 4KB pages given the same TLB size. For example, for the hash table, conventional translation requires a 128-entry TLB to match the miss ratio of SpryVM with 8 entries, a 16× difference. Furthermore, we see that the gap between the efficiency of TLBs worsens with the absence of data locality. For the skip list, which is the workload with the lowest data locality, conventional translation requires around 2048 entries to match the miss ratio of SpryVM with 16 entries.

2MB pages: Using 2MB pages improves the TLB behavior of conventional translation. For example, for the hash table, with 2MB pages, conventional translation only need 8 entries to match the miss ratio of 4 entries in SpryVM, a $2 \times difference$. In contrast, in workloads where there exist data locality, like in the trees, in which there is data reuse for the higher levels of the tree, the per-region $4KB$ pages are not able to beat 2MB pages. However, note that 2MB are not always available (as explained in the background section) and we could also employ $2MB$ pages for SpryVM; we defer that approach for future work. Furthermore, in workloads where there exists no locality, such as in the skip list data structure, 2MB deliver an almost negligible improvement with respect to $4KB$ pages (also shown in prior work for other data structure with low data locality like graphs [25]). Hence, for the skip list, the number of TLB entries still needs to be 16× bigger to match SpryVM's TLB area efficiency.

7.1.2. 128GB working set. Figure 7 shows the TLB miss ratio as we increase the number of entries of the four data structure traversals with a working set size of 128GBs. Similarly to the previous set of graphs, each graph plots three lines: conventional translation with 4KB pages, the same mechanism but using 2MB pages, and SryVM, which uses 4KB pages.

4KB pages: For all four data structures, SpryVM systematically beats conventional translation with 4KB pages given the same TLB size. Furthermore, the area-efficiency gap is more pronounced as the working set is larger.

2MB pages: Although 2MB pages helps to reduce the area-efficiency gap with respect to 4KB and SpryVM, the increase in the working set worses the benefits. For example, in the hash table, 2MB pages now requires 8× the number of TLB entries to match the miss ratio of 4 TLB entries in SpryVM. Furthermore, in the cases where there is more data locality, the situation also worsens, and in fact, for the binary search tree external, now 2MB pages and SpryVM, behave almost identically.

7.2. TLB Miss Penalty

SpryVM does not only improve the area-efficiency of TLBs but also reduces the TLB miss penalty. Figure 8 shows the TLB miss penalty of a memory reference, including the data access to memory, normalized to the ideal translation mechanism where only the data access of the memory references is required. We decided to be conservative and assume perfect MMU caches, and therefore page walks requires only one memory reference. The reason is because we want to decouple the TLB miss penalty benefits from the page table implementation, since in our case, with an inverted page table of $1/4x$ of load factor, we are able to guarantee one memory reference per page walk, whereas the hierarchical four-level page table requires a very significant MMU cache hierarchy. Furthermore, we consider two systems, a conventional two-socket machine and a large-memory eight-socket machine.

As the figure shows, page walks in a conventional VM mechanism requires a memory reference to locate the page table entry, and hence a memory reference requires two memory accesses. In contrast, and as explain in Figure 5, the virtual address uniquely identifies the memory region, and hence the time it takes to locate the region, traversing the NoC and off-chip interconnects is overlapped with the data path. Furthermore, the larger the memory systems, the larger the contribution of the network time in the page walk time, and hence the large the reduction in page walk time with respect with conventional translation.

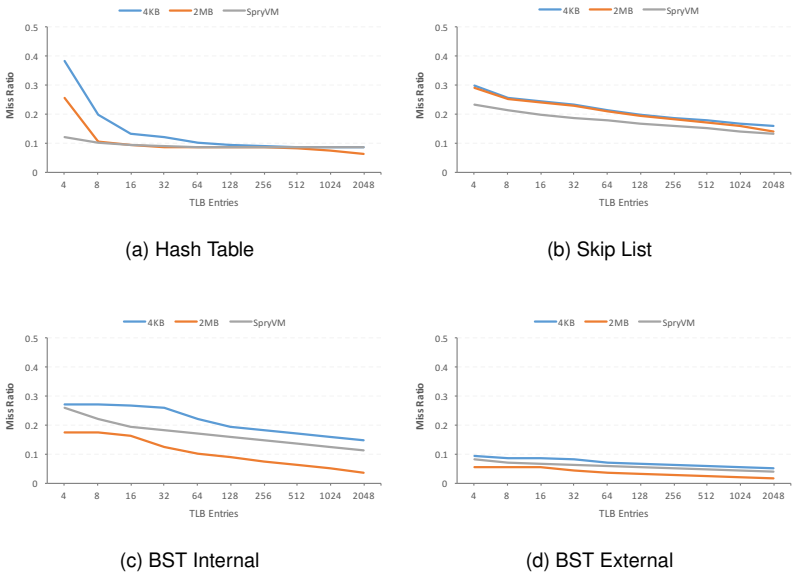

Figure 6: TLB sensitivity study for 32GB working set for conventional translation with 4KB and 2MB pages and SpryVM.

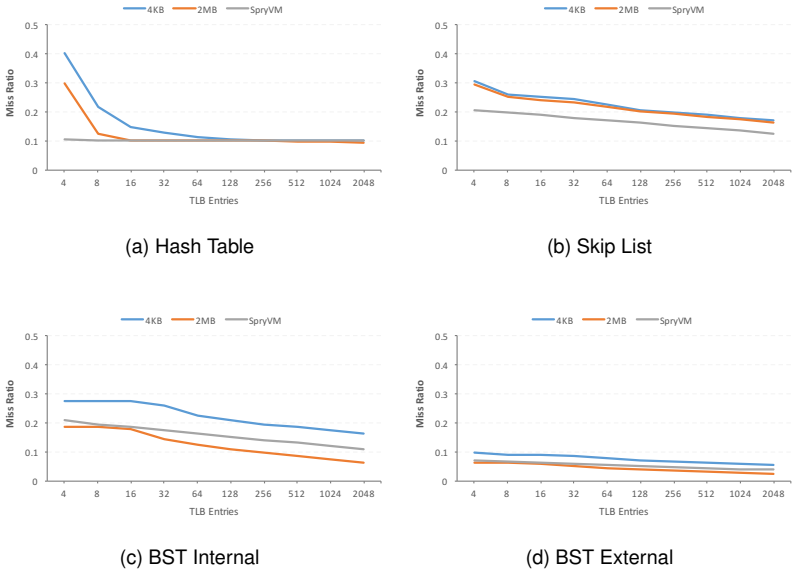

Figure 7: TLB sensitivity study for 128GB working set for conventional translation with 4KB and 2MB pages and SpryVM.

7.3. Associativity study of VM

7.4. Single-Process In-Memory

Fig. 9 shows results for three single-process workloads: Memcached, RocksDB, and Cassandra. The y-axis breaks down the total misses into the three distinct miss classes. Each category on the x-axis corresponds to the ratio between the size of the physical memory and the size of the application's working set. For example, $8\times$ indicates that the memory is eight times larger than the application's working set. Similarly, $1/2\times$ means that the application's working set is twice the size of the memory. We collapse results for $8\times$ to $1\times$ because they are similar and visually identical; each case represents a fully in-memory scenario. Furthermore, within each working set category, the x-axis sweeps through different VM associativities, from direct-mapped to 32-way associative.

Even with a simple direct-mapped translation, compulsory misses represent 99.9% of all the misses. There are no capacity misses as the working set fully fits in memory. Conflict misses are scarce. For example, Memcached using direct-mapped translation achieves a conflict miss rate in the order of one miss per 10^8 memory accesses. Using 2 ways removes

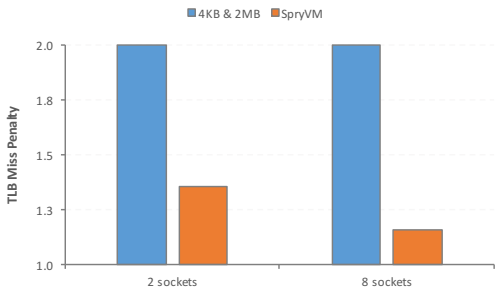

Figure 8: TLB miss penalty for conventional and SpryVM.

all the conflicts for the 8×, 4×, and 2× cases, while 4 ways are required for the 1× case (where the memory size is equal to the size of the working set). For in-memory scenarios, page conflicts arise because the virtual address space of server applications is particularly sparse; there are many virtual segments scattered all over the address space. For example, Java processes exhibit many virtual segments due to the dynamic nature of the JVM. This is best observed in the case of Cassandra, which exhibits direct-mapped conflict miss rates in the order of one miss per 10^7 and 10^6 memory accesses for the 8×–2× cases and the 1× case respectively. However, conflict misses drop rapidly as associativity increases and using 4 ways removes all conflicts for the 4× and 2× cases, while virtually eliminating conflicts for the 1× case. The observation that conflict misses drop rapidly with associativity has also been shown for caches [15, 27]. We elide the results for TPC-H, TPC-DS, and MySQL as they follow identical trends.

7.5. Single-Process Out-of-Memory

In contrast to in-memory scenarios, when the working sets do not fit in memory (1/2×, 1/4×, 1/8× cases), capacity misses grow with the working set size, while the fraction of compulsory misses drops. Although conflict misses are more significant than in the in-memory scenarios, conflicts drop sharply after 2-4 ways. In the worst case, 16 ways are required to drive conflict misses down to $\sim 1\%$ of all the misses. Note that Cassandra has an active working set that fits in a memory of 1/4× the data size, and capacity misses start to rise at the 1/8× case and beyond. Fundamentally, all these results corroborate the seminal work on caches [15, 27].

7.6. Multi-programming In-Memory

Fig. 10 presents results for multi-programming. We take each of the three workloads, Memcached, RocksDB, and Cassandra, and increase the number of processes, keeping the overall working set size the same with respect to the single process scenarios. For example, for the 1× case in Fig. 9, the working set of a single process equals the size of the memory. In Fig. 10, for two processes sharing the same physical memory, the per-process working set is half the size of the physical memory. Similarly, with four processes, per-process working sets consume a quarter of the physical memory. We thus guarantee that only the increase in the number of processes has an impact on the associativity.

For in-memory scenarios (8×–1× cases), once the associativity equals the number of processes, compulsory misses represent 99.9% of all the misses for all the workloads. Increasing the associativity further makes conflicts virtually disappear after a few additional ways. For instance, for Memcached, with 8, 16, and 32 ways, the conflicts are in the order of a single miss per 10^9 accesses for 2, 4, and 8 processes, respectively. Cassandra requires 4, 8, and 8 ways to achieve a miss conflict rate of one miss per 10^9 accesses for 2, 4, and 8 processes, respectively. Again, the trends are identical for TPC-H, TPC-DS, and MySQL. Overall, conflict misses become virtually zero with a few ways (i.e., 2-4) per process.

7.7. Multi-programming Out-of-Memory

For the cases where the working sets do not fit in memory, the trends are similar to the single-process scenarios. Capacity misses become more significant as the working set sizes grow, making conflict and compulsory misses less important. Similar to the in-memory case, once the associativity equals the number of processes, the fraction of conflict misses matches the single-process results. Conflicts drop rapidly as in the other cases and with 4 ways per process, conflict misses remain within 1% of the total misses for all the workloads. The results for TPC-H, TPC-DS, and MySQL are identical.

8. Discussion

We now discuss possible concerns and further considerations.

Cache hierarchy. We assume MPUs, much like conventional cores, integrate physical caches and an execution-side MMU with TLBs. Upon a page walk, the memory-side MMUs reply with the page table entry and cache block, which are stored in the MMU's TLB and data cache respectively. However, to avoid TLBs and TLB shootdowns [57], we could use virtual caches. Recent practical designs for virtual cache hierarchies [42, 59] would be a perfect fit for spryVM. In this approach, MPUs access the cache with virtual addresses, and upon a cache miss, the request is propagated to the memory-side MMUs to translate and fetch the corresponding block. The memory-side MMU only replies with the data cache block, simplifying our design.

Multi-level memories. Though prior work on MPUs assumes a single level [5, 6, 22, 49], memory can be organized as a hierarchy, with a die-stacked cache [51, 58] backed up by planar memory. For hardware-managed caches, the memory-side MMU performs the translation and accesses the partition, and in case of a cache miss, the page is fetched from planar memory as part of the standard cache miss operation. Once the page arrives into the partition, the data is sent back to the MPU. Note that moving the page from planar memory to the 3D memory does not affect the page table entry. In software-managed caches [51], MPUs rely on a software API for explicit migration of pages into the die-stacked memories.

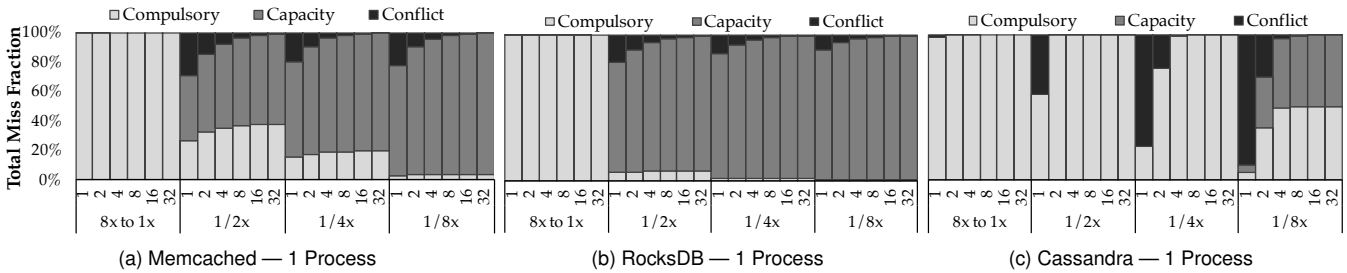

Figure 9: Overall miss ratio broken down into compulsory, capacity, and conflict misses.

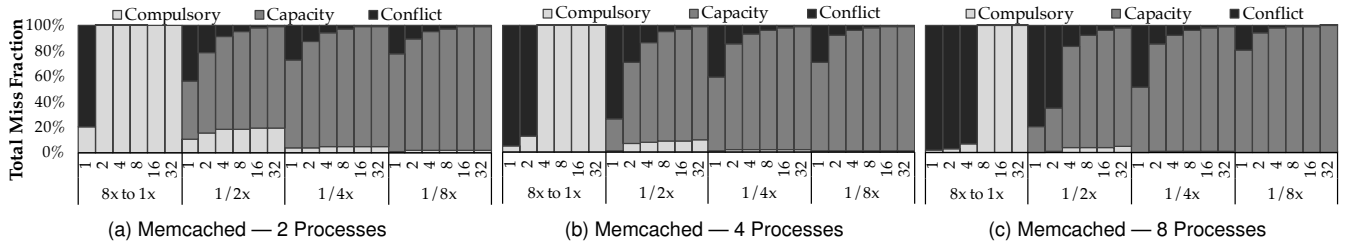

Figure 10: Overall miss ratio broken down into compulsory, capacity, and conflict misses.

Kernel memory. We consider user instructions only as we argue that MPUs, like all prior custom hardware (e.g., GPU, FPGA), should execute only user code. Nevertheless, spryVM is a great fit for the kernel, as Linux and FreeBSD's memory usage is almost entirely direct-mapped [38, 39] and memory resident. The TLB's tag matching logic would only require to skip the ASID bits for kernel accesses.

9. Related Work

Improving TLB performance. The techniques of Section **??** improve the TLB reach. In contrast, other techniques target reducing the TLB miss penalty. Commercial processors store page table entries in data caches to accelerate page walks [3]. Some architectures use page table caches (e.g., TSBs in SPARC [54]). MMU caches are employed to skip walking intermediate page table levels [8, 12]. Other techniques translate speculatively [9] or prefetch TLB entries [14].

Reducing VM associativity. We are not the first to restrict VM associativity. Several degrees of page coloring—fixing a few bits from the virtual-to-physical map—were proposed in the past. The MIPS R6000 used page coloring coupled with a small TLB to index the cache under tight latency constraints [55]. Page coloring has also been used for virtually indexed physically tagged caches [16] as an alternative to large associativities [24] or page sizes [30]. Alan Jay Smith [53] advocated the usage of set-associative mappings for main memory to simplify page placement and replacement.

10. Conclusion

Extending VM to MPUs is a crucial step in easing programmability, simplifying CPU-MPU sharing, and enabling transparent memory allocation and protection. In this work, we show that the full associativity of VM is largely unnecessary, as the majority of misses are either compulsory or capacity, and hence insensitive to associativity. By restricting the associativity to identify a memory chip and partition uniquely, memory can be accessed as soon as the virtual address is known, while a memory-side MMU translates and fetches the data, overlapping both operations almost entirely.

References

[1] Facebook LinkBench Benchmark. https://github.com/facebook/linkbench.

[2] Google Performance Tools. https://code.google.com/p/gperftools/.

[3] Intel 64 and IA-32 Architectures Software Developer's Manual. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-system-programming-manual-325384.pdf.

[4] RocksDB In Memory Workload Performance Benchmarks. https://github.com/facebook/rocksdb/wiki/RocksDB-In-Memory-Workload-Performance-Benchmarks.

[5] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 2015 International Symposium on Computer Architecture*, 2015.

[6] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. PIM-enabled instructions: a low-overhead, locality-aware processing-in-memory architecture. In *Proceedings of the 2015 International Symposium on Computer Architecture*, 2015.

[7] Rachata Ausavarungnirun, Joshua Landgraf, Vance Miller, Saugata Ghose, Jayneel Gandhi, and Christopher Rossbach. Mosaic: A gpu memory management with application-transparent support for multiple page sizes. In *Proceedings of the 2017 International Symposium on Microarchitecture*, 2017.

[8] Thomas W. Barr, Alan L. Cox, and Scott Rixner. Translation caching: skip, don't walk (the page table). In *Proceedings of the 2010 International Symposium on Computer Architecture*, 2010.

[9] Thomas W. Barr, Alan L. Cox, and Scott Rixner. SpecTLB: A mechanism for speculative address translation. In *Proceedings of the 2011 International Symposium on Computer Architecture*, 2011.

[10] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. Efficient virtual memory for big memory servers. In *Proceedings of the 2013 International Symposium on Computer Architecture*, 2013.

[11] Arkaprava Basu, Mark D. Hill, and Michael M. Swift. Reducing memory reference energy with opportunistic virtual caching. In *Proceedings of the 2012 International Symposium on Computer Architecture*, 2012.

[12] Abhishek Bhattacharjee. Large-reach memory management unit caches. In *Proceedings of the 2013 International Symposium on Microarchitecture*, 2013.

[13] Abhishek Bhattacharjee, Daniel Lustig, and Margaret Martonosi. Shared last-level TLBs for chip multiprocessors. In *Proceedings of the 2011 International Conference on High-Performance Computer Architecture*, 2011.

[14] Abhishek Bhattacharjee and Margaret Martonosi. Characterizing the TLB behavior of emerging parallel workloads on chip multiprocessors. In *Proceedings of the 2009 International Conference on Parallel Architectures and Compilation Techniques*, 2009.

[15] Jason F. Cantin. Cache Performance for SPEC CPU2000 Benchmarks, 2003.

[16] Tzi-cker Chiueh and Randy H. Katz. Eliminating the address translation bottleneck for physical address cache. In *Proceedings of the 1992 International Conference on Architectural Support for Programming Languages and Operating Systems*, 1992.

[17] Guilherme Cox and Abhishek Bhattacharjee. Efficient address translation for architectures with multiple page sizes. In *Proceedings of the 2017 International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[18] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized concurrency: The secret to scaling concurrent search data structures. In *Proceedings of the 2015 International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015.

[19] Michael Ferdman, Almutaz Adileh, Yusuf Onur Koçberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *Proceedings of the 2012 International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

[20] Jayneel Gandhi, Vasileios Karakostas, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman S. Unsal. Range translations for fast virtual memory. *IEEE Micro*, 36(3):118–126, 2016.

[21] John Gantz and David Reinsel. Transparent Hugepage Support, 2010.

[22] Mingyu Gao, Grant Ayers, and Christos Kozyrakis. Practical near-data processing for in-memory analytics frameworks. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation*, 2015.

[23] Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin R. Funston, Alexandra Fedorova, and Vivien Quéma. Large pages may be harmful on NUMA systems. In *Proceedings of the 2014 USENIX Annual Technical Conference*, 2014.

[24] R. N. Gustafson and Frank J. Sparacio. IBM 3081 processor unit: Design considerations and design process. *IBM Journal of Research and Development*, 26(1):12–21, 1982.

[25] Swapnil Haria, Mark D. Hill, and Michael M. Swift. Devirtualizing memory in heterogeneous systems. In *Proceedings of the 2018 International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018.

[26] Mark D. Hill. A case for direct-mapped caches. *Computer*, 21(12), December 1988.

[27] Mark Donald Hill. *Aspects of Cache Memory and Instruction Buffer Performance*. PhD thesis, University of California, Berkeley, 1987. AAI8813907.

[28] Kevin Hsieh, Samira Manabi Khan, Nandita Vijaykumar, Kevin K. Chang, Amirali Boroumand, Saugata Ghose, and Onur Mutlu. Accelerating pointer chasing in 3d-stacked memory: Challenges, mechanisms, evaluation. In *Proceedings of the 2016 IEEE International Conference on Computer Design*, 2016.

[29] Jonathan Corbet. Transparent huge pages in 2.6.38. https://lwn.net/Articles/423584/.

[30] Norman P. Jouppi. Architectural and organizational tradeoffs in the design of the multititan cpu. In *Proceedings of the 1989 International Symposium on Computer Architecture*, 1989.

[31] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman Ünsal. Redundant memory mappings for fast access to large memories. In *Proceedings of the 2015 International Symposium on Computer Architecture*, 2015.

[32] Vasileios Karakostas, Osman S. Unsal, Mario Nemirovsky, Adrián Cristal, and Michael M. Swift. Performance analysis of the memory management unit under scale-out workloads. In *Proceedings of the 2015 International Symposium on Workload Characterization*, 2014.

[33] Yusuf Onur Koçberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin T. Lim, and Parthasarathy Ranganathan. Meet the walkers: accelerating index traversals for in-memory databases. In *Proceedings of the 2013 International Symposium on Microarchitecture*, 2013.

[34] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. Coordinated and efficient huge page management with ingens. In *Proceedings of the 2016 USENIX Symposium on Operating Systems Design and Implementation*, 2016.

[35] Kevin T. Lim, David Meisner, Ali G. Saidi, Parthasarathy Ranganathan, and Thomas F. Wenisch. Thin servers with smart pipes: designing soc accelerators for memcached. In *Proceedings of the 2013 International Symposium on Computer Architecture*, 2013.

[36] Jason Lowe-Power. Inferring kaveri's shared virtual memory implementation. In *http://www.lowepower.com/jason/inferring-kaveris-shared-virtual-memory-implementation.html*, 2017.

[37] Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim M. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 Conference on Programming Language Design and Implementation*, 2005.

[38] Wolfgang Mauerer. *Professional Linux Kernel Architecture*. Wrox Press Ltd., Birmingham, UK, UK, 2008.

[39] Marshall Kirk McKusick, George Neville-Neil, and Robert N.M. Watson. *The Design and Implementation of the FreeBSD Operating System*. Addison-Wesley Professional, 2nd edition, 2014.

[40] Misel-Myrto Papadopoulou, Xin Tong, André Seznec, and Andreas Moshovos. Prediction-based superpage-friendly TLB designs. In *Proceedings of the 2015 International Symposium on High Performance Computer Architecture*, 2015.

[41] Chang Hyun Park, Taekyung Heo, Jungi Jeong, and Jaehyuk Huh. Hybrid tlb coalescing: Improving tlb translation coverage under diverse fragmented memory allocations. In *Proceedings of the 2017 International Symposium on Computer Architecture*, 2017.

[42] Hang H. Park, Heo Taekyung, and Jaehyuk Huh. Efficient synonym filtering and scalable delayed translation for hybrid virtual caching. In *Proceedings of the 2016 International Symposium on Computer Architecture*, 2016.

[43] Binh Pham, Abhishek Bhattacharjee, Yasuko Eckert, and Gabriel H. Loh. Increasing TLB reach by exploiting clustering in page translations. In *Proceedings of the 2014 International Symposium on High Performance Computer Architecture*, 2014.

[44] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. Colt: Coalesced large-reach TLBs. In *Proceedings of the 2012 International Symposium on Microarchitecture*, 2012.

[45] Binh Pham, Jan Vesely, Gabriel Loh, and Abhishek Bhattacharjee. Large pages and lightweight memory management in virtualized environments: Can you have it both ways? In *Proceedings of the 2015 International Symposium on Microarchitecture*, 2015.

[46] Bharath Pichai, Lisa Hsu, and Abhishek Bhattacharjee. Architectural support for address translation on gpus: designing memory management units for cpu/gpus with unified address spaces. In *Proceedings of the 2014 International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.

[47] Javier Picorel, Djordje Jevdjic, and Babak Falsafi. Near-memory address translation. In *Proceedings of the 2017 International Conference on Parallel Architectures and Compilation Techniques*, 2017.

[48] Jason Power, Mark D. Hill, and David A. Wood. Supporting x86-64 address translation for 100s of GPU lanes. In *Proceedings of the 2014 International Symposium on High Performance Computer Architecture*, 2014.

[49] Seth H. Pugsley, Jeffrey Jestes, Huihui Zhang, Rajeev Balasubramonian, Vijayalakshmi Srinivasan, Alper Buyuktosunoglu, Al Davis, and Feifei Li. NDC: analyzing the impact of 3d-stacked memory+logic devices on mapreduce workloads. In *Proceedings of the 2014 International Symposium on Performance Analysis of Systems and Software*, 2014.

[50] Red Hat Inc. *libHugeTLBFS*.

[51] James Reinders. Knights Corner: Your Path to Knights Landing. https://software.intel.com/sites/default/files/managed/e9/b5/Knights-Corner-is-your-path-to-Knights-Landing.pdf, 2014.

[52] Ashley Saulsbury, Fredrik Dahlgren, and Per Stenström. Recency-based TLB preloading. In *Proceedings of the 2000 International Symposium on Computer Architecture*, 2000.

[53] Alan Jay Smith. A comparative study of set associative memory mapping algorithms and their use for cache and main memory. *IEEE Trans. Software Eng.*, 4(2):121–130, 1978.

[54] Sun Microsystems. UltraSPARC T2 Supplement to the UltraSPARC Architecture 2007. http://www.oracle.com/technetwork/systems/opensparc/t2-13-ust2-uasuppl-draft-p-ext-1537760.html, 2007.

[55] George Taylor, Peter Davies, and Michael Farmwald. The tlb slice—a low-cost high-speed address translation mechanism. In *Proceedings of the 1990 International Symposium on Computer Architecture*, 1990.

[56] Ján Veselý, Arkaprava Basu, Mark Oskin, Gabriel H. Loh, and Abhishek Bhattacharjee. Observations and opportunities in architecting shared virtual memory for heterogeneous systems. In *Proceedings of the 2016 International Symposium on Performance Analysis of Systems and Software*, 2016.

[57] Carlos Villavieja, Vasileios Karakostas, Lluís Vilanova, Yoav Etsion, Alex Ramírez, Avi Mendelson, Nacho Navarro, Adrián Cristal, and Osman S. Unsal. Didi: Mitigating the performance impact of TLB shootdowns using a shared TLB directory. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, 2011.

[58] Stavros Volos, Djordje Jevdjic, Babak Falsafi, and Boris Grot. Fat caches for scale-out servers. *IEEE Micro*, 37(2):90–103, 2017.

[59] Hongil Yoon and Gurindar S. Sohi. Revisiting virtual L1 caches: A practical design using dynamic synonym remapping. In *Proceedings of the 2016 International Symposium on High Performance Computer Architecture*, 2016.