Set-Associative Virtual Memory Regions for Near-Memory Processing

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

Abstract

With the slowdown in silicon efficiency and density scaling, near-memory accelerators are emerging as a promising solution to bridging the logic/memory gap in online analytic workloads and services. The key challenge in incorporating accelerators into a modern platform, however, is doing so 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 budgets for silicon resources and footprints. Moreover, recent proposals for accelerator VM support break the conventional VM abstractions with intrusive solutions to software stack to facilitate address translation.

In this paper, 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 in practice has little impact on page fault traffic but dramatically reduces both the VM hardware support requirements and preserves demand paging to minimize the impact on existing software stacks. We observe that address translation overheads arise not just because TLBs miss frequently, but also because page table walks must complete before data access can proceed. SpryVM mitigates this translation-data access serialization by dividing memory into set-associative accelerator-local partitions which can mapped and walked locally. By overlapping page walks and data fetch operations almost entirely, 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.

1. Introduction

Djordje *I think the main points that need to come across are: (1) Server workloads keep more and more data in memory –> server memory grows sharply in capacity (2) Both compute and memory are scaling out due to the diminished scaling. Compute gets specialized resulting in a huge number of small execution units/accelerators, often placed close to memory -> Maximum (and average) distance between logic and memory keeps increasing, while the minimum distance keeps decreasing. (3) The effectiveness of hardware translation gets worse as memory grows (TLB reach), and as the distance between memory and logic grows (page walk latency). More complex hardware needed to keep up with growing memory. (4) Small accelerators have *the same or worse* translation hardware requirements as large CPU cores –> a) the hardware overhead of translation is huge relative to a tiny accelerator, b) the overall translation hardware in the system scales linearly with the number of accelerators. (5) The overhead of TLB coherence gets worse as the system grows (I'd rather leave it for another section)*

The net results are that translation HW requirements grow with the number of accelerators AND with memory capacity, with its effectiveness getting worse with both.

We propose a scalable mechanism whose whose effectiveness does not depend on memory size or number of accelerators, and whose hardware requirements will not grow with the number of accelerators.

Javier *I don't like Djordje's story because all the points can be easily addressed by DVM. We need to explain the tradeoff between translation HW and SW flexility and a table comparing all the previous approaches. For example, DVM shows 4 variables: programmability, power/perf, flexibility, and safety. I think a simple way to drive the message home is that you want both (1) the programmability benefits of pointer-is-a-pointer AND (2) flexible VM system (e.g., demand paging, COW), which is essentially what conventional address translation gives you. Then, we explain why it doesn't work for accelerators and we continue with SpryVM.*

Large-scale IT services are increasingly run in memory to accommodate tight query latency requirements. Online services rely on larger memory capacity to keep data closer to processing logic and faster processing to improve cost/performance and return on investment. In light of the slowdown in silicon efficiency and density scaling in recent years, designers are resorting to hardware accelerators for a wide spectrum of applications ranging from deep learning [], to analytics [] and graph processing [].

A salient feature of such accelerators is the need for a host CPU and OS to manage the execution resources among applications and application phases narrowing the scope of accelerator functionality to efficient and parallel in-memory computation. While many designers opt for accelerators to have direct physical access to memory, providing protection, isolation and the lack demand paging significantly complicates the software stack. Others have argued that the conventional virtual memory abstraction and a global address space programming model is best for software development enabling "a pointer is a pointer everywhere" semantics [57, 59, 72], extending memory protection to accelerators, and obviating the need for manual inter-CPU-accelerator data marshalling.

Unfortunately, conventional virtual memory (VM) cannot

simply be extended to accelerators. While general-purpose cores rely on deeper cache and TLB hierarchies and data reuse to bridge the gap between the core speed and memory capacity, accelerators primarily exploit parallel access with proximity to memory [] forgoing reuse and deep hierarchies. The latter are also prohibitive in the required silicon resources relative to silicon-optimized accelerators. Moreover, accelerators are often designed to saturate the available memory bandwidth and are best placed near memory controllers, with physical memory partitioned among all accelerators to maximize parallelism [].

Prior approaches to accommodate for accelerators either forgo VM and give direct physical access to memory for accelerators [], require major modifications to DRAM and memory controllers, or map physically contiguous regions of memory directly in VM or using segments []. These techniques fall short of providing the flexibility of a demand-paged VM and limit the utility of accelerators and/or introduce complex changes to the software stack.

In contrast, we 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 Page Regions of Your Virtual Memory). SpryVM restricts VM associativity so that a virtual address uniquely identifies the memory chip and partition that holds the data. This feature allows MPUs to access the memory as soon as the virtual address is known. Each memory partition integrates an MMU, which includes a TLB hierarchy and page table, that translates the virtual address and fetches the data—both the translation and data are always located in the MMU's local memory partition. SpryVM achieves low-overhead page walks as the page walk and data fetch operations overlap almost entirely. More specifically, our contributions are:

• 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 [33]. 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. Our study covers memory-resident and larger-than-memory working sets for single- and multi-programmed workloads.

• A design space exploration for a hardware translation mechanism that leverages VM's modest associativity requirements. While such mechanisms do require OS changes, we show that these changes are readily-implementable as they can leverage existing and ongoing work on page coloring [46], multi-page table synchronization for heterogeneous memories [29], and more. 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 and partition, allowing MPUs to access the memory as soon as the virtual address is known. Each memory partition features an MMU, which includes a page table and TLB hierarchy to localize address translation and data fetch, thus minimizing the overhead. Each MMU's TLB hierarchy serves only its memory partition, makings its TLB performance robust across any memory chip and 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.

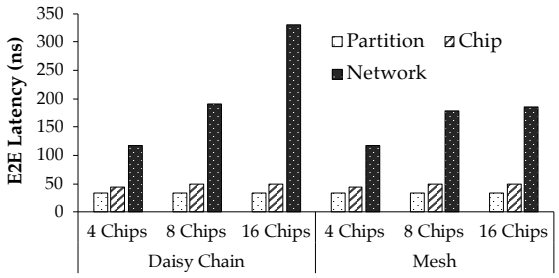


Figure 1: End-to-end latency of memory accesses.

While we propose using set-associative VM to tackle the address translation challenges posed by MPUs, our key observations are general and likely useful for broader classes of multi-chip systems. Set-associativity is a general concept applicable to any system with a significant memory access latency due to expensive chip-to-chip traffic. For instance, one could also envision the ideas of set-associativity and co-location of data and translation within the same memory partition/chip being applied to classic multi-socket NUMA machines [35, 49, 50]. We hope our study will inspire researchers to leverage set-associativity in other multi-chip systems.

2. Background

Javier *Points to come across:*

- *SW trend: Servers workloads are keeping their datasets memory resident; HW trend: Due to slowdown in silicon density and efficiency, computer system are integrating custom logic (accelerators).*
- *Explain the programmability benefits of pointer-is-a-pointer AND flexible VM system (e.g., demand paging, COW) of a conventional translation mechanism.*
- *TLB Reach Problem: Explain that a conventional translation mechanism is not effective for accelerators because (1) it relies on deep cache and TLB hierarchies and (2) data reuse, to bridge the gap between computation speed and memory capacity. However, accelerators primarily exploit parallel access with proximity to memory, and not reuse and deep cache hierarchies. Furthermore, accelerator are custom and hence silicon optimized, hence the available budget for translation hardware is limited. I think we can just cite prior work on TLB miss rates for in-memory workloads.*
- *TLB Penalty Problem: Explain that compute and memory*

Table 1: Comparison of SpryVM with previous approaches for reducing virtual memory overhead.

	Programmability	Area and Power	Flexibility	Safety
Multi-page mappings [55, 56]	✓	✗	✓	✓
Transparent Huge Pages [37]	✓	✗	✓	✓
libhugetlbfs [61]	✗	✗	✓	✓
Direct Segments [12]	✗	✓	✗	✓
Redundant Memory Mappings [40]	✓	✗	✓	✓
Direct-mapped Mappings [32, 58]	✓	✓	✗	✓
SpryVM	✓	✓	✓	✓

are scaling-out due to the slowdown in silicon scaling and efficiency, hence TLB misses (page walks) become more costly. We can show something similar to Figure 1 here.

No prior proposal for virtually addressed MPUs is both general and efficient. The simplest approaches offload the address translation task to the CPU cores or an IOMMU [28, 51, 72, 76]. Though simple, the fact that the MPUs and CPUs are on different chips means that sending translation requests to the CPU results in expensive chip-to-chip communication. Alternatively, one can integrate a traditional MMU with a TLB hierarchy per MPU. However, due to the limited reach of modern TLBs, page walks occur frequently, resulting in costly cross-chip traffic, as the multiple levels of the page table [27] are arbitrarily scattered across the memory chips. All of these existing techniques create excessive cross-chip traffic, either to reach the CPU or to perform page walks, which could account for latencies of hundreds of nanoseconds, dramatically increasing the translation overhead.

Several recent proposals seek to improve the TLB reach. The most common approach is the introduction of larger page sizes [37, 61]. Similarly, CoLT [56] and Clustered TLBs [55] coalesce 4-8 page translations into a single TLB entry, as long as their physical locations are contiguously in memory. Although the TLB reach improves, it is still unable to cover the entirety of a large memory system with tens or hundreds of GBs [26]. More innovative ways of improving TLB coverage employ variable-size segments instead of fixed page-based translations [12, 40, 53]. Unfortunately, the effectiveness of these techniques relies on the OS to allocate contiguous chunks of physical memory, which is not possible when the system is under memory pressure. Hence, page walks are still common in some scenarios, accounting for translation overheads of hundreds of nanoseconds.

To summarize, there is no single approach capable of providing a unified VM that is suitable for all use cases. Prior work attempts to exploit contiguity in the physical memory, which is not always available. In contrast, our study focuses on sidestepping the need for fewer TLB misses by overlapping them with data fetch.

3. Set-associative Virtual Memory

Javier *Points to come across:*

- *We can have a first subsection to talk about the 4 points of the intro's table (i.e., programmability, power/perf, flexibility, safety). What we would like to have, the goal.*
- *We can have second subsection to talk about the problems of full associativity, and the idea of set associativity, which is essentially partitioning the memory and translation information and knowing to which memory set to go, and how that can help at the aforesaid 4 points. We can have a high-level figure on how the memory and the translation information is partitioned. NOTE: This section has to be short, maximum one page, and zero results! It has to be high level!!!*

Table 2: Workload description.

Workload	Description
Cassandra	NoSQL data store running Yahoo's YCSB.
Memcached	Cache store running Twitter-like workload [43].
TPC-H	TPC-H on MonetDB column store (Q1-Q21).
TPC-DS	TPC-DS on MonetDB column store (Queries of [41]).
MySQL	SQL DBMS running Facebook's LinkBench [1].
RocksDB	Store engine running Facebook benchmarks [5].

In modern page-based VM, a virtual page is mapped on demand into a page frame. Modern systems place no restrictions on the virtual-to-physical mapping; a virtual page can potentially map to any page frame. Though flexible, this fully associative approach places translation on the critical path of every memory access, because a memory access cannot begin until the translation operation finishes.

One potential way to break translation-memory access serialization is to completely eliminate associativity and enforce a direct mapping between virtual pages and page frames. Since a virtual page can now only map to a single page frame, address translation and data fetch are independent and can proceed in parallel. Conventional wisdom dictates that direct mapping creates an excess of page faults, due to conflicts from multiple virtual pages mapping to the same page frame. However, this is merely intuition, as 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 [34].

To fill this research void, we employ the 3C model—initially developed for caches [34]—to study VM associativity. In this context, associativity means the number of possible locations

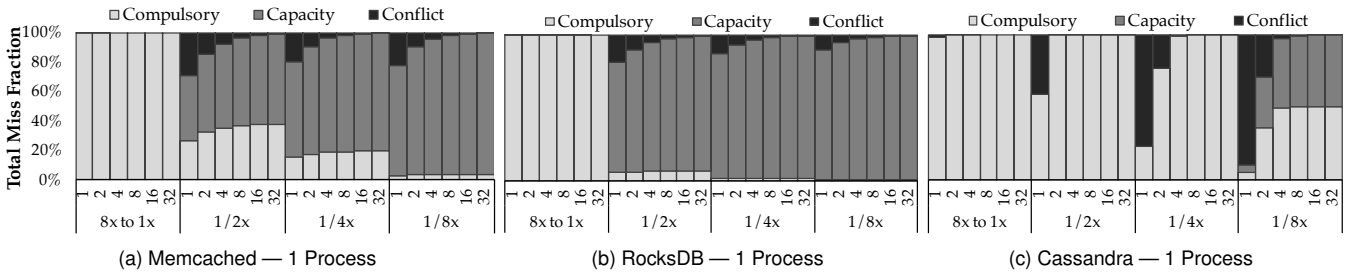


Figure 2: Overall miss ratio broken down into compulsory, capacity, and conflict misses.

(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).

To simulate real-world scenarios, we select a set of representative server workloads, summarized in Table 2. We include two cloud workloads from CloudSuite [25], Cassandra and Memcached, an online transaction processing (OLTP) workload [1], MySQL, two online analytical processing (OLAP) workloads [18], TPC-H and TPC-DS, and a widely-used storage system workload [24], RocksDB. We collect long memory traces of several 10s of billions of instructions of the server workloads using Pin [44]. We extract the virtual address and address space identifier (ASID) of each memory reference and use it to probe a set-associative memory structure, varying the associativity to observe and classify the misses. A detailed description of our methodology is found in Section 6.

3.1. Single-Process In-Memory

Fig. 2 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 all the conflicts for the $8\times$, $4\times$, and $2\times$ cases, while 4 ways are required for the $1\times$ 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 $8\times$–$2\times$ cases and the $1\times$ case respectively. However, conflict misses drop rapidly as associativity increases and using 4 ways removes all conflicts for the $4\times$ and $2\times$ cases, while virtually eliminating conflicts for the $1\times$ case. The observation that conflict misses drop rapidly with associativity has also been shown for caches [19, 34]. We elide the results for TPC-H, TPC-DS, and MySQL as they follow identical trends.

3.2. Single-Process Out-of-Memory

In contrast to in-memory scenarios, when the working sets do not fit in memory ($1/2\times$, $1/4\times$, $1/8\times$ 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 $1/4\times$ the data size, and capacity misses start to rise at the $1/8\times$ case and beyond. Fundamentally, all these results corroborate the seminal work on caches [19, 34].

3.3. Multi-programming In-Memory

Fig. 3 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 $1\times$ case in Fig. 2, the working set of a single process equals the size of the memory. In Fig. 3, 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 ($8\times$–$1\times$ cases), once the associativity equals the number of processes, compulsory misses represent 99.9% of all the misses for all the workloads. Increasing

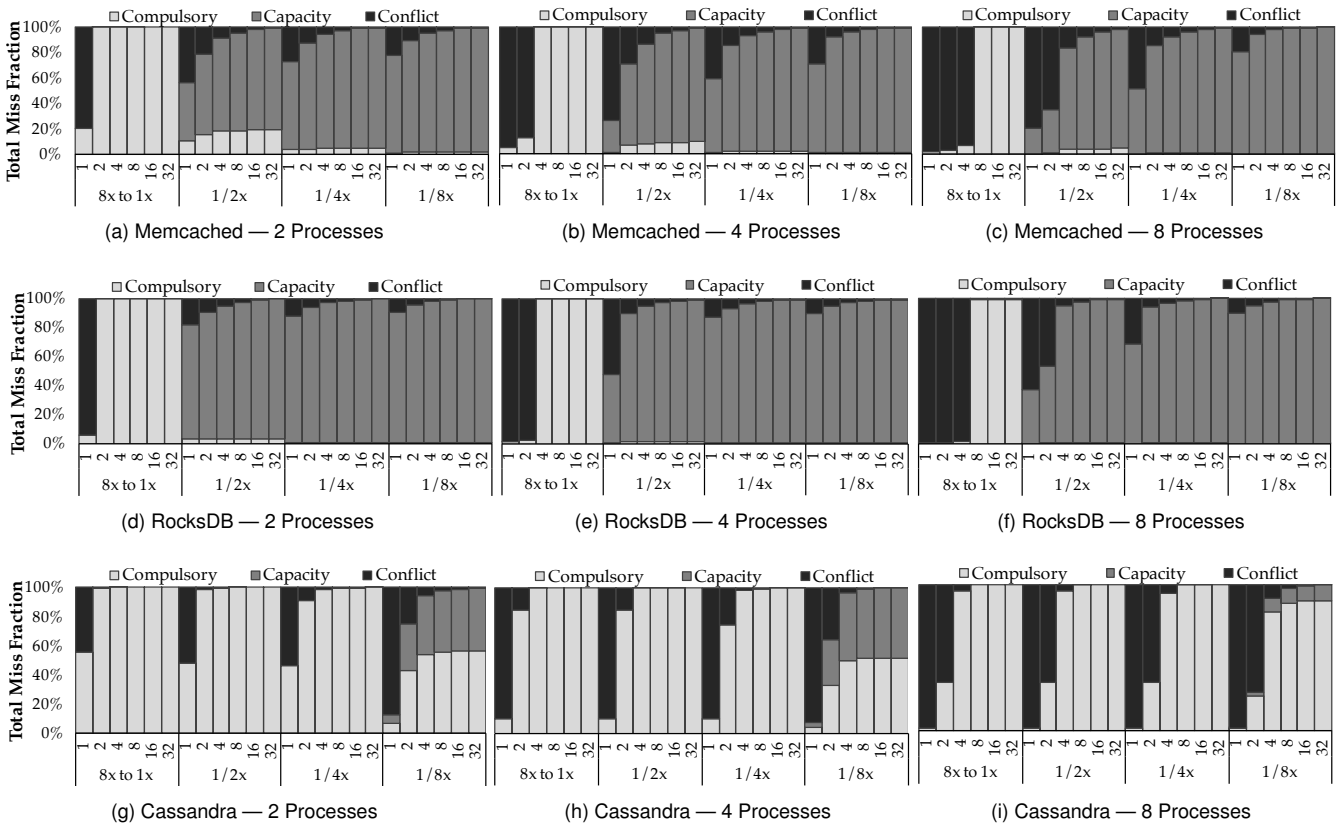


Figure 3: Overall miss ratio broken down into compulsory, capacity, and conflict misses.

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.

3.4. 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.

3.5. Observations & Implications

Our results show that fully associative VM is unnecessary for many modern server workloads. This observation holds across scenarios that span single process, multi-programming, in-memory and out-of-memory working sets. Compulsory and capacity misses dominate, and conflict misses, which associativity alleviates, drop rapidly as the associativity increases. Specifically, for in-memory scenarios, compulsory misses dominate when associativity equals the number of processes. For out-of-the-memory scenarios, capacity misses dominate and become more important as working set sizes increase. In this case, conflict misses become scarce once associativity matches the number of processes, and virtually disappear after a few extra ways. These trends match prior literature on caches [19, 33, 34]—just as set-associative (or direct-mapped) caches can often provide nearly all the benefits of full associativity with faster access times, set-associative VM can provide nearly all the benefits of full associativity with faster translation.

To put the associativity requirements into perspective, a commodity server today integrates 256GB of memory [49]. Assuming 4KB pages, fully associative VM provides $64M$ ways. As there are around 100 processes concurrently running after booting [6], even when assuming a system with 128 processes and an associativity of 16 per process—which is extreme as we have seen that an associativity of 2–4 per process is sufficient—the total associativity requirements does not exceed $2K$ ways. Even for a commodity server, the $2K$

requirement is $32K\times$ less than what full associativity provides. A server with a larger memory capacity (e.g., large NUMA machines [35, 50]) or with emerging non-volatile memories (e.g., 3D XPoint [4]) would supply even more ways, widening the gap between the required and provided associativity.

4. Operating System Support

Javier *Points to come across:*

- *We need to remove the table. We need to write in a positive way that our changes are very simple and to prove it we managed to build a prototype. We need to explain the modifications with maybe pseudocode. Then, we can have a paragraph on the limitations of the prototype and talk about the memory-mapped files and explain what we would need to make it work, and why it is feasable. Maybe we can also talk about the COWs in SpryVM.*

SpryVM requires a number of operating system (OS) modifications. The OS must assign page frames to virtual pages so that both pages map to the same memory set. Furthermore, the OS must create and manage spryVM's inverted page table, and synchronize it with the native page table.

4.1. Support spryVM with NUMA page allocation policy

In Linux, mapping only a subset of physical pages to given virtual addresses is very similar to a scenario existing in NUMA systems — binding the memory of processes to a specific NUMA nodes. With NUMA memory bind policy (i.e. *membind*), a process is only able to get physical pages coming from the NUMA node assigned by the policy, where, with spryVM, each subset of a process's virtual address space can be seen as being bound to a certain NUMA node. As a result, to implement spryVM in operating systems, we are able to reuse most of existing NUMA-related code. In a spryVM system, each memory set can be treated as a NUMA node and all processes have `membind` as their default memory allocation policy.

In Linux, there are two types of pages, anonymous pages and file-backed pages. The former is generally private to each process and the latter can be shared between different processes. Thus, they need to handled differently.

Anonymous pages Whenever a page fault happens, during the process of physical page allocation, the operating system calculates the corresponding NUMA node id from the faulting virtual address with our hash function, then obtains a free page from that NUMA node and finishes the page fault process. Linux uses `alloc_pages_vma()` as the entry point of each user space page allocation, thus, we simply calculate a memory set index (stored in a NUMA node mask variable) out of the faulting address, derive the corresponding zone list, and pass them to the page allocation function (`__alloc_pages_nodemask()`). Linux is able to handle the rest of page fault work without any problem. The basic operations are shown in Algorithm 1.

File-backed pages File-backed pages can be shared among different processes but allocating file-backed pages based on faulting virtual addresses only works when they belong to only one process. Because there is no guarantee that other processes will map a file-backed page with the same virtual address or an virtual address having the same memory index as the first virtual address mapping to the page. To solve this, we use page index in a file as the memory set id for each page. A page index tells the position of the page residing in the file it maps to. For example, for a 64KB file, there will be 8 pages mapping to it and first page gets page index 0, second gets page index 1, and so on. Thus, all file-backed pages are allocated from different memory sets based on our index scheme. On the other hand, we need to adjust virtual address assignment in the kernel, which comes from `get_unmapped_area()`. All we need to do is aligning the beginning of the given virtual address to match memory set 0's index. Thus, each process maps a file all starting from memory set 0.

Inverted page table Not all architectures use inverted page tables, but they are commonly used in IBM Power systems as hashed page tables. Thus, we adopt the same operation model as hashed page tables in IBM Power systems: reserving memory for each inverted page table in individual NUMA node and inserting/invalidating/updating inverted page table entries whenever CPU page tables change in the same hook functions (e.g. `update_mmu_cache()`). To guarantee the translation information is coherent between CPUs and accelerators, each inverted page table entry has a valid bit, which is atomically accessed by both CPUs (for inverted page table modification) and accelerators (for reading). Alternatively, if accelerators, e.g. GPUs, want to maintain their own page tables instead of using the inverted page table, additional hook functions, like `mmu_notifier` mechanism in Linux, could be used to keep the page tables of accelerators coherent with CPU page tables.

Both modifications are very simple and not intrusive to operating systems. We only add around 270 lines of code in Linux to achieve the required functionality for spryVM.

Algorithm 1 Memory set indexing algorithm

```
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
```

5. Discussion

Javier *I think this should be after evaluation.*

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. How-

ever, to avoid TLBs and TLB shootdowns [73], we could use virtual caches. Recent practical designs for virtual cache hierarchies [54, 77] 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 [7, 8, 28, 60], memory can be organized as a hierarchy, with a die-stacked cache [62, 74] 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 [62], MPUs rely on a software API for explicit migration of pages into the die-stacked memories.

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 [45, 46] and memory resident. The TLB's tag matching logic would only require to skip the ASID bits for kernel accesses.

6. Methodology

Like most recent work on VM [11, 12, 14, 52, 55, 56, 63], we use trace-driven functional and cycle-accurate simulation.

6.1. Traces

We collect memory traces using Pin [44]. 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 [13, 77], and use it to probe a set-associative memory structure.

For the associativity experiments in Section 3, 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 use the same traces for the page table experiments of Section **??** and feed them into our inverted page table modeling tool. For the TLB experiments in Section **??** and the performance experiments in Section 7, we tune the workloads to employ working sets of size 32GB and 64GB, depending on the size of the network. We use 32GB and 64GB for 4- and 8-chip, and 16-chip configurations respectively.

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.

6.2. 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 [15, 52, 63] to evaluate 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 server workloads [25]. The CPI is measured by feeding the memory traces into our cycle-accurate simulator.

6.3. Simulation Parameters

We use the Flexus cycle-accurate simulator [75], with detailed core, MMU, memory hierarchy, and interconnect models. Following prior work on MPUs [7, 28, 60], we model the MPU cores after ARM Cortex A7 [9]. We provision the baseline with a high-end MMU similar to Intel Xeon Haswell [23, 31], with multi-level TLBs and MMU caches [10, 14]. For the baseline, we consider a 4-level hierarchical page table [36] (as in ARMv8 and x86_64) and an inverted page table identical to the one spryVM employs, but covering the whole memory. We assume 48-bit virtual and physical addresses. The MMU supports 4KB, 2MB, and 1GB pages. Page table entries are transparently allocated in the L1-D cache like in commercial systems [3]. For simplicity, we probe the cache with physical addresses for the baseline and with virtual addresses when using spryVM. Both behave identically as we verify that TLB misses never reference a cache-resident block.

We model our 3D memory stack following the organization of the Micron's Hybrid Memory Cube with eight 8Gb DRAM layers and 16 vaults [47]. We conservatively estimate DRAM timing parameters from publicly available information and research literature [28]. For spryVM, we employ a conventional two-level TLB hierarchy. The SRAM overhead per memory chip is less than 256KB (which is partitioned across vaults) for an area of $0.3mm^2$ in 22nm, corresponding to less than 0.2% of the area of an 8Gb DRAM die (i.e., $226mm^2$ [64]). As there are 16 memory partitions (or vaults), each of 512MB, spryVM provides a VM associativity of 128K ways, making the number of page faults virtually identical to full associativity.

Table 3: System parameters.

MPU logic	Description
Cores	Single-issue, in-order, 2GHz
L1-I/D	32KB, 2-way, 64B block, 2-cycle load-to-use
MMU	**Description**
TLB	4KB pages: 64-entry, 4-way associative 2MB pages: 32-entry, 4-way associative 1GB pages: 4-entry, fully associative
STLB	4KB/2MB pages: 1024-entry, 8-way associative
Caches	L4: 2-entry, fully associative [14] L3: 4-entry, fully associative [14] L2: 32-entry, 4-way associative [14]
Memory	**Description**
MPU chip	8GB chips, 8 DRAM layers x 16 vaults
Networks	4, 8, and 16 chips in daisy chain and mesh
DRAM	$t_{CK} = 1.6$ns, $t_{RAS} = 22.4$ns, $t_{RCD} = 11.2$ns $t_{CAS} = 11.2$ns, $t_{WR} = 14.4$ns, $t_{RP} = 11.2$ns
Serial links	2B bidirectional, 10GHz, 30ns per hop [39, 71]
NoC	Mesh, 128-bit links, 3 cycles per hop
spryVM	**Description**
TLB	4KB pages: 64-entry, 4-way associative
STLB	4KB pages: 1024-entry, 8-way associative

We conservatively assume that page faults always involve SSD IO, taking $32\mu s$ to resolve [22]. This parameter is only important for out-of-memory workloads. Memory-resident workloads only suffer a page fault the first time a page is accessed. Table 3 summarizes all system parameters.

7. Evaluation

Javier *Points to come across:*

- *Set-associativity doesn't increase page faults: 3C model + Memcached and RocksDB on real HW*
- *TLB miss rate vs. working set for microkernels (hash table, skip list, bst internal, bst external)*
- *TLB miss penalty vs. working set (or sockets)*
- *Fragmentation*
- *$IPC = IPC_{b}ase + Penalty_{t}lb + Penalty_{p}age_{f}aults$*

We now perform a quantitative and qualitative study on VM.

7.1. Performance Analysis

The paper discusses many different workload scenarios and system configurations. As there are way too many performance points, we will show only the scenarios that have significant performance differences. First, we will show two workload scenarios, in-memory, where the memory size is equal to the working set size (i.e., the 1x ratio), and out-of-memory, where the working set size is eight times larger than the memory size (i.e., the 1/8x ratio). Second, as the performance across different process counts does not vary significantly, we are showing the average across runs. Third, the two topologies, daisy chain and mesh, behave similarly in terms performance, and hence we also present the average of both. Last, we present the results for different memory chip counts.

Fig. 4a shows the speedup of five translation mechanisms over the conventional MMU using 4KB pages and the hierarchical page table, labeled 4KB-H, for the in-memory scenario. The techniques are the conventional MMU using 4KB pages and the inverted page table, 2MB pages using the hierarchical and inverted page tables, spryVM, and an ideal translation mechanism with zero translation overhead. We label the techniques as 4KB-I, 2MB-H, 2MB-I, spryVM, and Ideal respectively. In the interest of space, we omit the results with 1GB pages, which perform better than 4KB pages but always worse than 2MB pages. The reason is that the number of entries in the MMU for 1GB pages is significantly limited, and server workloads have a large number of virtual segments which are accessed concurrently. First, we see that spryVM clearly outperforms 4KB-H, 4KB-I, 2MB-H, and 2MB-I. In some cases by a large margin, up to 25% for TPC-H. In other cases more moderately, 2.1%, 2%, 1.3%, and 0.5% over 4KB-H, 4KB-I, 2MB-H, and 2MB-I respectively for MySQL. Most importantly, spryVM is consistently on par with the ideal translation that incurs zero overhead for translation. Overall, spryVM improves performance by 17.2%. 14%, 12.3%, and 10.5% over 4KB-H, 4KB-I, 2MB-H, and 2MB-I respectively, and stays within 1.2% of the ideal translation mechanism on average.

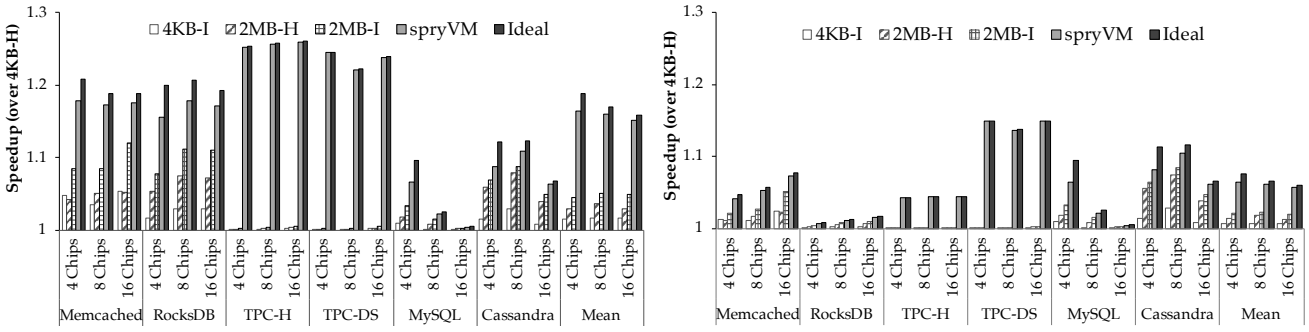
Fig. 4b presents the speedups for the out-of-memory scenario. As expected, the speedups are less significant that in the in-memory case. The reason is that all the cases incur the slowdown of resolving page faults, and therefore there are fewer accesses that can be accelerated. Still, spryVM systematically performs better than the conventional MMU, 6.1%, 5.4%, 4.5%, and 4% on average over 4KB-H, 4KB-I, 2MB-H, and 2MB-I respectively, and stays within 0.6% of Ideal.

Importantly, we model a greatly optimistic case when using 2MB and 1GB pages as we assume all pages are huge, with no generation overhead or fragmentation in any scenario. Additionally, we do not account for the excess in IO traffic generated by writing back dirty huge pages. Therefore, we expect further improvements with a realistic overhead.

7.2. Comparison with Other Proposals

All the recent proposals on address translation for CPUs presented in Section **??** aim to enhance the reach of TLBs by exploiting the contiguity available in the virtual and physical address spaces. Hence, these techniques are orthogonal to spryVM as we reduce the TLB miss penalty.

However, these techniques would be only effective for in-memory scenarios, and perform poorly for larger-than-memory working sets. In out-of-memory scenarios, direct segments [12] would not be able to allocate the most frequently used pages in memory. Similarly, Redundant Memory Mappings and Hybrid TLB Coalescing [40, 53] would not be able to allocate contiguous chunks of physical memory and fall back to base pages. CoLT [56] and Clustered TLBs [55] would also perform similar to the conventional translation



(a) In-memory scenario.

(b) Out-of-memory scenario.

Figure 4: Speedup over 4KB-H for 4KB-I, 2MB-H, 2MB-I, spryVM, and an ideal translation.

when there is limited physical contiguity available.

8. 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 [66]). MMU caches are employed to skip walking intermediate page table levels [10, 14]. Other techniques translate speculatively [11] or prefetch TLB entries [16].

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 [70]. Page coloring has also been used for virtually indexed physically tagged caches [21] as an alternative to large associativities [30] or page sizes [38]. Alan Jay Smith [65] advocated the usage of set-associative mappings for main memory to simplify page placement and replacement.

9. 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] Intel, Micron reveal Xpoint, a new memory architecture that could outclass DDR4 and NAND. http://www.extremetech.com/extreme/211087-intel-micron-reveal-xpoint-a-new-memory-architecture-that-claims-to-outclass-both-ddr4-and-nand.

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

[6] Jeongseob Ahn, Seongwook Jin, and Jaehyuk Huh. Revisiting hardware-assisted page walks for virtualized systems. In *Proceedings of the 2012 International Symposium on Computer Architecture*, 2012.

[7] 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.

[8] 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.

[9] ARM. Cortex-A7 Processor. http://www.arm.com/products/processors/cortex-a/cortex-a7.php.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

[17] David L. Black, Richard F. Rashid, David B. Golub, Charles R. Hill, and Robert V. Baron. Translation lookaside buffer consistency: A software approach. In *Proceedings of the 1989 International Conference on Architectural Support for Programming Languages and Operating Systems*, 1989.

[18] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. Breaking the memory wall in monetdb. *Commun. ACM*, 51(12), December 2008.

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

[20] Ray Cheng. Virtual address cache in unix. In *Proceedings of the Summer 1987 USENIX Technical Conf.*, 1987.

[21] 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.

[22] Chia-Chen Chou, Aamer Jaleel, and Moinuddin K. Qureshi. CAMEO: A two-level memory organization with capacity of main memory and flexibility of hardware-managed cache. In *Proceedings of the 2014 International Symposium on Microarchitecture*, 2014.

[23] Intel Corporation. Tlbs, paging-structure caches and their invalidation, 2008.

[24] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. Optimizing space amplification in rocksdb. In *Proceedings of the 8th Conference on Innovative Data Systems Research*, 2017.

[25] 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.

[26] 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.

[27] John Gantz and David Reinsel. White Paper: 5-Level Paging and 5-Level EPT, 2016.

[28] 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.

[29] Jerome Glisse. HMM (Heterogeneous Memory Management) v22, 2017.

[30] 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.

[31] Per Hammarlund. 4th Generation Intel ® Core ™ Processor, codenamed Haswell. http://www.hotchips.org/wp-content/uploads/hc_archives/hc25/HC25.80-Processors2-epub/HC25.27.820-Haswell-Hammarlund-Intel.pdf, 2013.

[32] 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.

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

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

[35] HP. HP to Transform Server Market with Single Platform for Mission-critical Computing. http://www8.hp.com/us/en/hp-news/press-release.html?id=1147777#.WRs0sbyGOL4.

[36] Bruce L. Jacob and Trevor N. Mudge. A look at several memory management units, tlb-refill mechanisms, and page table organizations. In *Proceedings of the 1998 International Conference on Architectural Support for Programming Languages and Operating Systems*, 1998.

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

[38] 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.

[39] David Kanter. Cavium Thunders Into Servers: Specialized Silicon Rivals Xeon for Specific Workloads, 2016.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

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

[46] 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.

[47] Micron. Hybrid Memory Cube Specification 2.1, 2014.

[48] Juan Navarro, Sitaram Iyer, Peter Druschel, and Alan L. Cox. Practical, transparent operating system support for superpages. In *Proceedings of the 2002 Symposium on Operating System Design and Implementation*, 2002.

[49] Jia Ning. Open Compute Project: Facebook Server Intel Motherboard V3.1 Rev 1.00. https://www.circleb.eu/wp-content/uploads/2016/04/Open_Compute_Project_FB_Server_Intel_Motherboard_v3.1_rev1.00.pdf, 2016.

[50] Oracle. SPARC M7-16 Server. https://www.oracle.com/servers/sparc/m7-16/index.html.

[51] Mark Oskin, Frederic T. Chong, and Timothy Sherwood. Active pages: A computation model for intelligent memory. In *Proceedings of the 1998 International Symposium on Computer Architecture*, 1998.

[52] 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.

[53] 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.

[54] 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.

[55] 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.

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

[57] 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.

[58] 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.

[59] 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.

[60] 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.

[61] Red Hat Inc. *libHugeTLBFS*.

[62] 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.

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

[64] Manjunath Shevgoor, Jung-Sik Kim, Niladrish Chatterjee, Rajeev Balasubramonian, Al Davis, and Aniruddha N. Udipi. Quantifying the relationship between the power delivery network and architectural policies in a 3d-stacked memory device. In *Proceedings of the 2013 International Symposium on Microarchitecture*, 2013.

[65] 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.

[66] 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.

[67] Madhusudhan Talluri and Mark D. Hill. Surpassing the TLB performance of superpages with less operating system support. In *Proceedings of the 1994 International Conference on Architectural Support for Programming Languages and Operating Systems*, 1994.

[68] Madhusudhan Talluri, Mark D. Hill, and Yousef A. Khalidi. A new page table for 64-bit address spaces. In *Proceedings of the 1995 Symposium on Operating Systems Principles*, 1995.

[69] Madhusudhan Talluri, Shing Kong, Mark D. Hill, and David A. Patterson. Tradeoffs in supporting two page sizes. In *Proceedings of the 1992 Annual International Symposium on Computer Architecture*, 1992.

[70] 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.

[71] Brian Towles, J. P. Grossman, Brian Greskamp, and David E. Shaw. Unifying on-chip and inter-node switching within the anton 2 network. In *Proceedings of the 2014 International Symposium on Computer Architecture*, 2014.

[72] 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.

[73] 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.

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

[75] Thomas F. Wenisch, Roland E. Wunderlich, Michael Ferdman, Anastassia Ailamaki, Babak Falsafi, and James C. Hoe. Simflex: Statistical sampling of computer system simulation. *IEEE Micro*, 26(4):18–31, 2006.

[76] Sam (Likun) Xi, Oreoluwa Babarinsa, Manos Athanassoulis, and Stratos Idreos. Beyond the wall: Near-data processing for databases. In *Proceedings of the 2015 International Workshop on Data Management on New Hardware*, 2015.

[77] 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.