Accelerating Fault-Triggered Page Table Walks Using Software Translation Caches

Wu, Michael
Rutgers University
Department of Computer Science
New Brunswick, NJ, 08901, USA
mw811@rutgers.edu

Patel, Parth A.
Rutgers University
Department of Electrical and Computer Engineering
New Brunswick, NJ, 08901, USA
pap223@rutgers.edu

Abstract

One of the greatest bottlenecks in modern operating system is the cost of performing a page table walk. This has been identified as such a large performance problem that multiple hardware structures (TLBs, hardware page walkers and page walk caches) have been developed with the sole purpose of accelerating this process. While these structures benefit greatly for pages that have already been mapped into the virtual address space, they are left unused when a page walk is forced to occur in software due to a page fault. In these cases, the operating system is still forced to walk all levels of the page table (5 in the most recent versions of Linux), making a memory access for each level, all of which may require going to DRAM.

To reduce the number of memory references performed in a page walk on a page fault, we introduce a software page walk cache. Our page walk cache stores the virtual memory addresses of page table entries from all levels and, on hit, allows the operating system to skip one or more levels of the page table walk. We test the performance benefits of our page walk cache with sequential, random, and semi-random accesses, as well as a matrix multiplication program. On average, with the cache enabled, the total time spent walking the page table in software is reduced by 20%, with some cases seeing a 35+% reduction.

1 Introduction

A major bottleneck in modern operating systems is virtual address translations for virtual memory. The result of a Translation Lookaside Buffer (TLB) miss incurs a penalty of making 4 memory references to locate a page table entry by doing a page table walk. Even with the advance hardware of modern MMU's to make page table walks efficient in hardware, TLB misses may still account for upto 50% of an application's run time, despite high TLB hit rates of 95+% [3].

When a translation is not found or is invalid, a page fault will be triggered and the operating system's page fault handler which resolves the issue through software. Page table walks with hyper specialized MMU hardware are a performance concern. This process is every costlier through software in the operating system page fault handler, which performs software page table entry lookups and software page table walks.

In this paper, we examined the overhead of page walks on page faults. A page table walk is the look up of a single page table entry. We also measured the overhead of processing page faults after the walk has been completed. Page faults are a common occurrence in modern operating systems. By analyzing the performance of the Linux page fault handler, we were able to recognize the lack of scalability. These issues are increasingly impacting large data centers dedicated to massive parallel data processing, where page faults and page walks are frequent due to the nature of the workload.

Along these findings, in this paper, we propose a Software Translation Cache to accelerate software page table walks. The Software Translation Caches cache the most recently accessed translation in each level of the page table through a simple indexing/tagging schema based off the design as described in [1]. With a hit on any level of the cache, we can skip one or more levels of a page table walk. If a miss is returned for one level of the cache, all lower levels automatically return misses as well, retaining correctness for translations. The Software Translation Cache improves page fault handling performance, reducing the total time spent walking the page table in software by 20%-35+%, by adding less than 200 lines of code the Linux operating system.

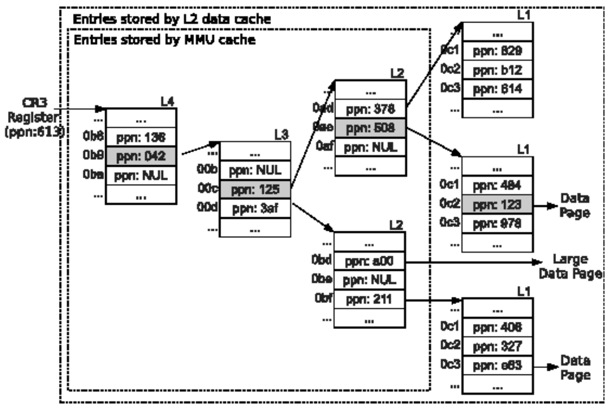

Figure 1: An example 4-level page walk for virtual address (0b9, 00c, 0ae, 0c2, 016). Each page table entry stores the physical page number for either the next lower level page table page (for L4, L3, and L2) or the data page (for L1). Only 12 bits of the 40-bit physical page number are shown in these figures for simplicity.

2 Background

Virtual memory is a critical part of most operating systems. It allows us to run more applications on the systems and give each of them more resources than that which is physically available. Among other things, memory virtualization helps with fragmentation, security, and physical limitations of storage. The cost of memory virtualization, however, is steep.

To get a physical page from its corresponding virtual page, hardware or software must translate the virtual memory address to a physical memory address. These translations are held in the Page Table. The page table is accessed by two very different entities: the memory management unit (MMU) and the operating system. The MMU is the hardware structure in place for performing the translation from physical to virtual addresses and page table management. The operating system on the other hand is also editing and accessing the page table through software (for example, fault handling or allocating/deallocating a Virtual Memory Area in Linux).

In Linux, the page table consists of a multiple level radix tree. A virtual address is split into a page number and an offset. The page number is then split further into 9-bit indices for each level of the page table. Each page in x86 architecture is 4KB, and each level of the page table is one page. At each level of the page table, the 512 entries are indexed using the corresponding 9-bit index, returning the address of the next level of the page table tree for the virtual address being translated.This process is repeated until either the page table entry (PTE) is found, which results in a successful translation, or is missing, resulting in a page fault.

There are two kinds of page faults: major and minor faults. Typically, a system will have many minor faults, where the page exists in memory, but the page table does not contain an entry for it, and fewer major faults, where the page you are looking for has been swapped to disk and must be loaded back to memory. On a page fault, execution is context switched to the kernel, where the OS page fault handler begins to assess the type of fault and take according actions. This process will result in a page table walk through software. In the next section, we will analyze the impact of page table walks on page fault, as well as the necessity for improvements.

3 Motivation

Page faults are a common occurrence in modern operating systems. To further examine the problem of today's multi-level radix tree page table walk, we measured the overhead of page walks for a single PTE on page faults. We also measured the overhead of processing page faults after the walk has been completed. To do so, three timer calls were placed within the function *__handle_mm_fault()* in mm/memory.c, one at the start of the function, one before the call to *handle_pte_fault()*, and one before the final return statement. Note that this method measures only the page fault overheads of pte level faults. All modifications were made to Linux version 4.19.80.

By perform either sequential or random accesses to a large, page-aligned array allocated using mmap(), the total execution time is measured. System calls are used to obtain the total time spent performing page walks, as well as the total time spent processing page faults after the walk is completed. The program outputs the fraction of total execution time spent walking the page table or processing page faults, where Multi-Alloc Parameter refers to the number of pages are allocated on a page fault.

The Figures 2 and 3 show the measurements of the percentage of total execution time spent processing page faults and performing page walks. It can be seen that for sequential, random, and semi-random accesses, page fault processing consumes significantly more time than just the page walk on a fault. For both types of accesses, the fraction of time spent processing faults increases as the multi-alloc parameter increases. This is expected, as many more memory accesses are required to actually allocate a new page, as the operating system must first find

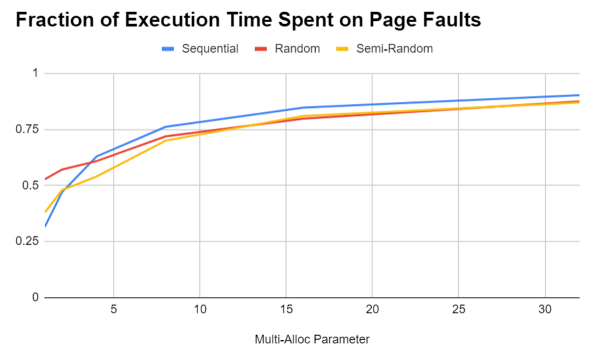

Figure 2: Fraction of execution time spent on page faults across sequential, random, and semi-random workloads.

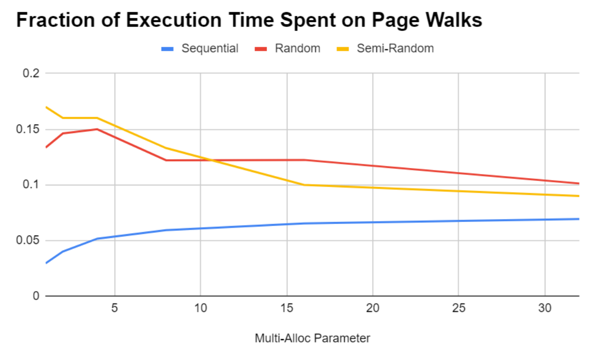

Figure 3: Fraction of execution time spent on page walks (for page faults) across sequential, random, and semi-random workloads. Significant difference for sequential (cache friendly) vs random workloads.

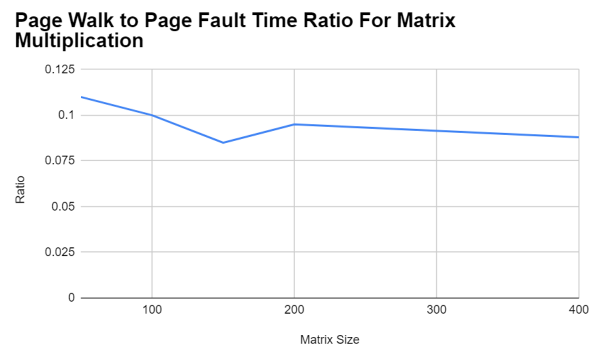

Figure 4: This graph shows the ratio of total time spent in page walks (for page faults), to the total time spent in all page faults. It is seen that for all tested matrix sizes, the time taken to walk the page table is between 8-10% of the time taken to process the entire page fault.

a free physical page, and then allocate a new page table entry for it. This process also requires significantly more computation.

Different trends are seen for partial walk times for sequential and random accesses. The partial walk fraction increases with the allocation parameter for sequential accesses, and decreases for random accesses. This is likely due to how allocating multiple pages at a time is beneficial for sequential accesses, decreasing running time. At the same time, the same number of page walks are performed, keeping the raw time the same, so the fraction increases. The opposite is true for random and semi-random accesses. Allocating extra pages is generally a waste, so overall running time increases, lowering the fraction of time spent on partial page walks. Total fault processing time increases with the allocation parameter for both types of accesses, as expected.

It is important to note that for random and semi-random accesses, the fraction of time spent on page walks is significantly larger than for sequential accesses. This is explained by regular caching of intermediate page table entries. When a page walk occurs, the page table entries are found in memory. Thus, they can possibly benefit from cache locality. This is the case for sequential accesses, where adjacent virtual pages are always accessed near each other, experiencing high spatial locality when the page table walk occurs. However, random pages are far less likely to experience cache locality. Thus, each level of the page table walk is more likely to miss in the caches and go to main memory. This takes more time, explaining how page walks make up a larger amount of execution time in the programs with random and semi-random accesses.

By analyzing the performance of the Linux page fault handler, we were able to recognize the necessity for performance improvement. Additionally, in the Linux Kernel, page fault handling is mostly serial, becoming a serious scalability hazard. Due to a lock on the *mm_struct*, this greatly reduces parallel processing capabilities on fault heavy workloads. This issue is surfacing most notably in Big Data workloads [2], which require parallel processing based computation at large data centers with nearly random memory access across multiple sockets of processors. Therefore, improving the time of a software page table walk can have significant improvements for large computationally heavy data centers, as well as other devices using the Linux operating system.

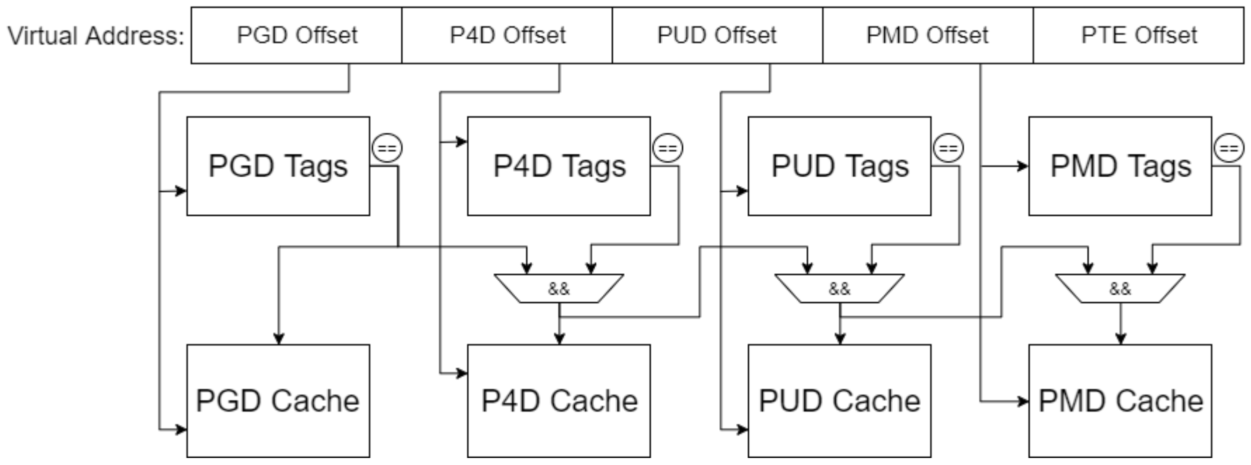

Figure 5: High-level design of a software split translation cache. Both a tag array and page cache exist for all levels of the page table except the last. For each level of the page table in the cache, the bottom bits of the corresponding offset are used to index both the tag array and page cache of that level. To access any given page cache, the tags for that level and all higher levels must match.

4 Design

The high-level design of our page walk cache is shown in Figure 5. Our design resembles a software implementation of the hardware split translation cache (STC) as described in [1], and adds less than 200 lines of code the Linux operating system. Caches are intialized for each individual *vm_area_struct*, and involve two primary structures, tag arrays and page caches. The tag arrays hold offsets used to walk the page table, while the page caches store the addresses of actual page table entries which have been accessed in the past. Both the tag arrays and page caches are designed as direct-mapped caches, as adding associativity in software (where memory accesses cannot explicitly be performed in parallel) increases the number of memory accesses just to check the cache, defeating the purpose of skipping memory accesses in the page walk. The following subsections describe how this page walk cache is accessed and used.

4.1 Accessing and Updating the Cache

In this subsection we describe in detail the mechanisms used to determine cache hits and misses, as well as how the cache is updated. We assume an adequate method for indexing the cache. This issue is explored, and our solution is presented, in Section 4.2.

Before examining how cache hits and misses are determined, we first describe how cache hits and misses affect cache behavior in the first place. When an address used for a page walk misses in all levels of the cache, the operating system proceeds to perform a regular page walk. After the page walk completes, all of the tag arrays are indexed, and their entries filled with the page walk offsets of the address used in the walk. Additionally, all page caches are indexed, and their entries filled with the virtual addresses of the pages accessed during the walk. More formally, after a walk using the address A with pgd, p4d, pud, and pmd offsets O_G, O_4, O_U, and O_m, respectively, which accesses the pgd, p4d, pud, and pmd page table entries P_G, P_4, P_U, and P_m, respectively, O_G, O_4, O_U, and O_m will be written to entries in the corresponding tag arrays, while P_G, P_4, P_U, and P_m will be written to entries in the corresponding page caches.

When an address hits in any level of page walk cache, the lowest level page cache for which there was a hit is accessed. The operating system then starts the page walk from that level of the page table, skipping the previous levels. After the page walk completes, the tag arrays and page caches for levels lower than the accessed page cache are updated as described before. For example, if an address hits in the pgd, p4d, and pud caches, the pud cache will be accessed, obtaining the memory address of a previously used pud page table entry. The walk will then progress as normal from the pud level, accessing the pmd and (possibly) pte levels of the page table. After the walk completes, the pmd tag array and page cache will be updated with the pmd offset and the pmd page that was accessed in the walk.

To determine whether or not an address hits in each level of the page walk cache, the OS compares the page walk offsets of the address to those stored in the indexed tag array entries. For any level of the page walk cache, if the offset does not match with the data stored in the tag array, that level of the cache returns a miss. Additionally, if a miss is returned for one level of the cache, all lower levels automatically return misses as well, regardless as to whether there was a match or not. Thus, a hit occurs in one level of the page walk cache if and only if the tags match for that level and all levels above it. This avoids cases where false positives occur due to two addresses sharing a lower offset, while differing in higher order bits.

4.2 Indexing the Cache

Because our page walk cache stores entries for all levels of the page table, the hashing scheme used to index each tag array and cache should also involve each page walk index in the virtual address. If this were not the case, the page caches for one or more levels would either become useless or grossly inefficient. As a demonstration, consider a scheme that indexes all tag arrays and page caches using some arbitrary (but consistent) hash function of the pud offset. This indexing method works efficiently for accessing the pud page cache (where locality exists in the pgd, p4d, and pud offsets, but not the pmd offset), but performs poorly for all other levels. We provide two examples to show why such a scheme is inefficient.

For the first example, let there be two addresses that share pgd and p4d offsets, but differ in the pud offsets. Clearly, for an properly working cache, the access for the second address should hit in the p4d page cache. However, for this indexing scheme, the pud offsets for both addresses differ, so (disregarding hash collisions), the second address will check the tag caches with a different index, resulting in a false miss. Note that for these cases, not only are false misses induced, but the entire pgd and p4d caches are rendered useless, as indexes are determined only from the pud offset.

As another example, consider three addresses used in a page walk, the first two sharing all but the same pmd offset, and the third having all the same offsets as the first address. In this case, after the first two walks complete, the third walk should match with all levels of tag arrays, leading to a hit in the pmd page cache for the pmd entry used in the first walk. However, since the first two addresses share the same pud offset, the second page walk will overwrite the pmd tag array and page cache entries, leading to the third address hitting only in the pud cache. In this case, because all address with the same pud offset index to the same pmd tag array and page cache entry, regardless of their pmd offset, the size of the pmd page cache is effectively limited to a single entry. Therefore, even though no incorrect behavior or false misses are exhibited, the efficiency is still greatly reduced.

In light of this issue, we present a simple solution to maximizing the efficiency of each level of the page walk cache, which we use in our implementation. Instead of using the same hash to index each level of the page walk cache, our indexing scheme takes hashes of each page walk offset separately, and uses them to access only the corresponding page level's tag array and page cache. In other words, the pgd tag array and page cache are indexed by a hash of the pgd offset, the p4d tag array and page cache are indexed by a hash of the p4d offset, and so on. Because this indexing method uses any page walk offset to index only that page level's cache, it removes indexing-related false misses, and also allows for all entries in each page cache to be used. Note that although this results in many cases where a different index is used for each level of the page walk cache, correctness is still maintained, as two addresses with the same offsets for a given level will necessarily index to the same entry for that level's tag array and page cache.

5 Evaluation

We quantify the benefits of adding our software page walk cache by measuring the total time spent walking the page table in software when the cache is either enabled or disabled. We test this using a program that performs sequential, random, and semi-random accesses over a large region of memory, as well as a single-threaded matrix multiplication program. For the first program, we also vary the number of threads performing accesses to see the effect our page walk cache may have on synchronization costs in the OS (due to acquiring a lock on the *mm_struct* before walking the page table). All tests are performed in an x86_64 QEMU virtual machine running with KVM enabled. The host machine uses an Intel® Core™ i7-7700HQ Quad-Core processor and has 8GB of DRAM. We find that in our tests, the addition of a software page walk cache reduces the total time traversing the page table in software by ∼25% on average, with speed up being as high as 37% in some cases.

5.1 Testing of Different Types of Accesses

To test the general effectiveness of our page walk cache we varied the type of access made on a region of memory between sequential, random, and semi-random accesses. Our sequential access test steps through the re-

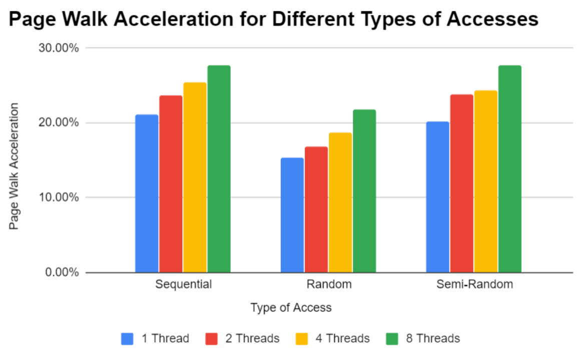

Figure 6: Percentage speedup of software page walks when using a software page walk cache for sequential, random, and semi-random accesses. The greatest differences are seen for the sequential and semi-random accesses, with improvements up to 28%. Still, an 18% average speedup is still exhibited in tests with fully random accesses.

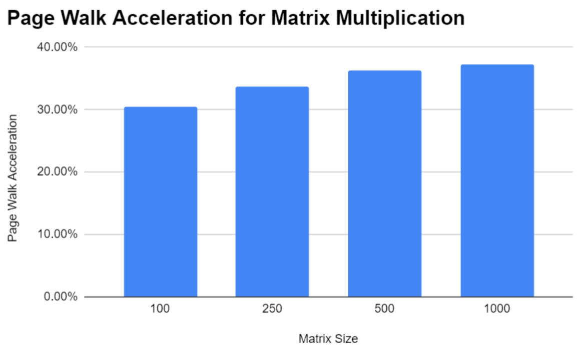

Figure 7: Percentage speedup of software page walks when using a software page walk cache for a single-threaded matrix multiplication program. Tests are performed with varying matrix sizes. The average observed speedup is 34%, and the maximum is 37%.

gion of memory one page at a time, triggering a page fault on adjacent pages in virtual memory with every access. For random accesses, the program makes completely randomized accesses across a region of memory, while for semi-random accesses, the program makes a random access, followed by four sequential page-sized steps. All tests are performed in a 1GB page-aligned region of memory allocated using mmap(). The results are shown in Figure 6.

From the tests we see that using the page walk cache greatly benefits both the sequential and semi-random accesses, with the cache accelerating single threaded page walks of both types of accesses by approximately 20%. Between the two, sequential accesses experience a greater speedup ($\sim$1%) than the semi-random accesses, as expected. The least improvement is seen for random accesses, falling in line with intuition. Surprisingly, though, a significant 15% speedup is still seen for random accesses. This is likely a result of the region of memory being only 1GB in size, causing all virtual pages in the region would share the same p4d offset. We expect that this speedup would decrease greatly in a significantly larger (512 or more GB) memory region, though we lack the physical resources to perform such an experiment.

5.2 Testing of Multiple Threads

Another concern of the current page table structure is how on a fault, a thread must acquire a lock on the *mm_struct* before processing a page fault, adding synchronization cost. By reducing the amount of time spent walking the page table, the time each thread spends in the critical section is reduced. Consequently, accelerating the page walk of a single thread indirectly accelerates the page walks of all other threads of a process.

We test the effects of our page walk cache on this by performing the same tests as in Section 5.1, this time with multiple threads. For each test, all threads allocate their own 1GB region of memory and perform the same type of access on it. We see that indeed, the page walk acceleration increases with the number of threads. For both sequential and semi-random accesses, page walk acceleration increases from 20% up to 28% when increasing the number of executing threads from one to eight. Even for random accesses, the speed up increases from 15% to 22% when eight threads are executed, although this could once again be a result of the small testing region.

5.3 Matrix Multiplication Test

For our final test, we examine the benefits of our cache for a single-threaded matrix multiplication program. In this experiment, we vary the size of the matrix from 100x100 up to 1000x1000, effectively testing how our cache scales with increasing working sets. It is expected that any benefits would increase or at least remain constant for a matrix multiplication program, as the accesses would generally be sequential in nature. Our results confirm that this is the case. As shown in Figure 7, even for a relatively small 100x100 matrix, the use of the software page walk cache reduces fault-triggered page walk time by 30%. For a larger 1000x1000 matrix, this figure increased to 37%, suggesting that the benefits increase with the working set size.

An interesting question arises as to why page walk acceleration is even greater in a matrix multiplication program than in the basic sequential access program from Section 5.1. The explanation is that in the basic program, faults are triggered on every memory access, with nothing happening in between. This allows for better locality in the hardware caches, leading to improved page walk performance even without the software cache. For the matrix multiplication, a significant number of memory accesses are performed in between faults to perform the program's function. This causes thrashing of page table entries with userspace data in the hardware cache, reducing page walk performance. Consequently, when the software page walk cache is enabled, an even greater performance boost is seen.

6 Limitations, Conclusions, and Future Directions

Page table walks pose a significant performance problem for modern operating systems. In the single-threaded case, they impose a large overhead due to five memory accesses (one for each level of the page table) being required just to perform address translation. In the multi-threaded case, this overhead is compounded by synchronization costs, as threads in the same process must lock the page table before walking it. For non-faulting page walks, hardware structures such as TLBs and MMU caches allow for significant acceleration. Memory accesses that result in page faults, however, are forced to walk the page table in software, and thus do not benefit from such hardware acceleration.

To accelerate fault-triggered software page walks, this paper introduces a software page walk cache implemented in the Linux operating system. The design of this page walk cache involves maintaining separate direct mapped caches in software for each level of page table entries except the last. When a software page walk begins, the cache is indexed using the different page walk offsets in the virtual address. The operating system uses the lowest-level cache hit, and continues the page walk from that level, effectively skipping part of the page walk.

We find that the inclusion of this cache significantly accelerates software page walks, with page walks being accelerated by as much as 37% in some use cases. This serves to suggest that it is worthwhile to further investigate software translation caches. As of yet, a bug still exists in our implementation where when multiple threads perform accesses at the same time, the rss-counter will sometimes be updated incorrectly. In the future, it is worth testing software page walk caches more rigorously and comprehensively, possibly experimenting with varying cache sizes, associativity, and granularity (e.g. per-*mm_struct* instead of per-VMA caches). In the case that impressive performance benefits continue to be seen, it may even be worthwhile to look into integrating fault-triggered software page walks better with existing translation hardware.

References

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

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

[3] Idan Yaniv and Dan Tsafrir. Hash, Don't Cache (the Page Table). In *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science*, SIGMETRICS '16, pages 337–350, New York, NY, USA, 2016. ACM.