Breaking the Address Translation Wall by Accelerating Memory Replays

Abhishek Bhattacharjee
Rutgers University

Emerging software stacks continue to process ever-increasing amounts of data, posing a performance challenge to the virtual memory layer of modern computer systems. In particular, address translation is now an acute system performance bottleneck. In response, we propose a class of cache prefetchers triggered by page table walk (PTW) activity. Our scheme—translation-enabled memory prefetching optimizations (TEMPO)—hinges on two observations. First, a substantial fraction of DRAM references in modern big-data workloads are devoted to accessing page tables (PTs). Second, when memory references require PT lookups in DRAM, the majority of them also look up DRAM for the subsequent data access. TEMPO exploits these observations to enable cache prefetching of the data pointed to by the PT. TEMPO requires only modest changes to hardware and no OS or application-level changes. Overall, TEMPO improves performance by 10-30 percent and energy by 1-14 percent.

Practically all computer systems use address translation to realize the benefits of virtual memory. Without virtual memory, system programmability, code portability, memory protection, and system security would be compromised. Address translation is central to these features.

Unfortunately, however, address translation has become a performance bottleneck and can consume 20-50 percent of system runtime today.[1-12] Consequently, vendors are rapidly improving address translation hardware. For example, TLBs are becoming larger and more complex; Intel Skylake chips now complement Level 1 (L1) TLBs with 1,536-entry 12-way set-associative Level 2 (L2) TLBs. Academic studies have explored the benefits of shared TLB topologies,[7] TLB prefetching,[2,3] and compression strategies,[6] as well as speculation.[4,8] Similarly, PT walkers

Published by the IEEE Computer Society
0272-1732/18/$33.00 ©2018 IEEE

have evolved from being implemented with lightweight OS routines to becoming hardware structures that can service multiple TLB misses concurrently. Meanwhile, novel structures like MMU caches accelerate TLB misses by caching entries from the upper levels of the radix-tree PT (typical of x86-64 and ARM).[5,12] Even OS support for superpages has improved over decades of research.[3,5,13-15] Nevertheless, despite these efforts, address translation overheads continue to be a vexing problem.

OUR CONTRIBUTIONS

We present TEMPO to improve address translation performance. Prior work generally aims to reduce the frequency of TLB misses or their latency. However, there is another aspect to address translation that has been ignored: the cost of subsequently accessing the data that was originally requested by the memory reference that suffered the TLB miss. TEMPO accelerates this component of address translation, also known as a memory replay.

Many workloads suffer from such poor locality of reuse that they require PT lookups that miss in both TLBs and caches, thus requiring DRAM lookup. We find that in 98 percent of these cases, the replay also requires DRAM lookup. This is intuitive—if the PT entry has poor reuse (meaning it is absent from TLBs/caches), any data in the physical frame it points to has even poorer reuse. Because the replay does not begin until the page walk completes, replays are an important source of performance degradation. TEMPO reduces replay overheads by prefetching the replay's target cache line into the last-level cache (LLC), thereby converting replay LLC misses into LLC hits. TEMPO initiates prefetch when DRAM is accessed for a PT entry.

To motivate TEMPO, Figure 1 shows data from a 32-core Intel Skylake system running Ubuntu Linux 4.14 with 4 Tbytes of DRAM. Workload inputs are scaled so that their total memory footprint is close to 4 Tbytes. Using Tbytes of memory with irregular access patterns in this way showcases the operating point where address translation is challenging. To create a fair baseline, we instrument Linux to employ 2 Mbyte superpages transparently. In our experiments, we found that Linux used superpages to cover over half of the memory footprint of each workload. However, the applications are memory- and pointer-intensive and have such poor access locality that their TLB misses often require PT lookups in DRAM. In fact, 20-30 percent of workload runtime is expended on DRAM lookups for PTs (DRAM-PTW-Access). However, Figure 1 also reveals a hitherto ignored fact that the time spent on DRAM accesses for the replay (DRAM-Replay-Access) takes up another 20-30 percent of performance. TEMPO converts almost all of these replay DRAM accesses to LLC hits and improves performance by 10-30 percent. TEMPO requires modest hardware changes to the PT walker and memory controller and no OS or application changes. By shortening runtime, TEMPO also saves 1-14 percent of system energy.

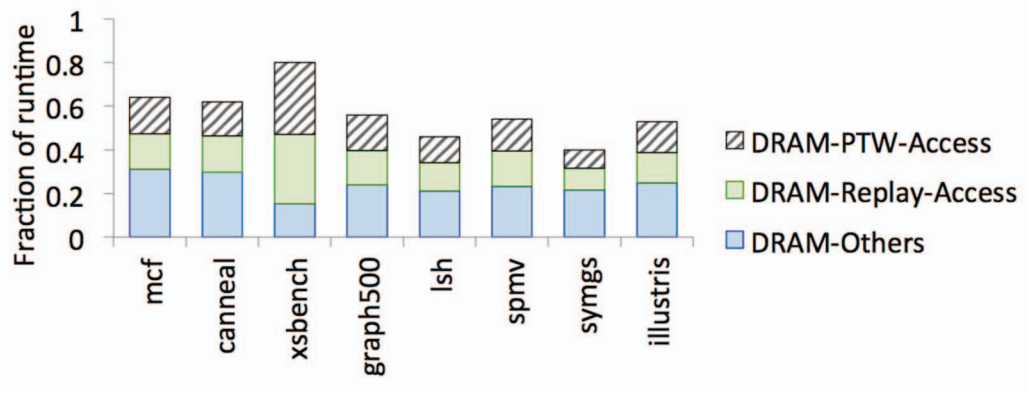

Figure 1. Fraction of total application runtime for PT accesses to DRAM (DRAM-PTW-Access), replay accesses to DRAM (DRAM-Replay-Access), and other non-PT DRAM accesses (DRAM-Others).

MOTIVATION AND BACKGROUND

All memory references require two steps: a virtual-to-physical translation and a post-translation lookup.

Virtual-to-Physical Address Translation

Virtual addresses are grouped into page-sized chunks, called virtual pages. A PT maps each virtual page to a physical page, where data actually resides in system memory. For example, consider the scenario where virtual Page 2 maps to physical Page 2. The physical page stores cache lines. For x86-64 systems using a base page size of 4 Kbytes, a page stores 64 distinct 64-byte cache lines. Naturally, this view of virtual-to-physical page mappings can take many real implementations. For example, x86-64 systems use a multi-level forward-mapped radix tree to represent the PT. We refer readers to prior work for details on x86-64 PTs.[3,8,13,16,17]

Because PTWs require four sequential (and hence expensive) memory references, CPUs use two classes of dedicated hardware structures to accelerate translation lookup. The first is the TLB, which stores virtual-to-physical translations and caches frequently used entries from the L1 PT. The second is a family of MMU caches, which accelerate PTWs when TLBs miss.[5,12,16] MMU caches store frequently used entries from the Level 4 (L4), Level 3 (L3), and L2 PT levels. They are generally smaller (by 32X in Skylake processors) than TLBs, because L4, L3, and L2 PT entries map much larger chunks of the address space than L1 PT entries. For the same reason, despite their smaller size, MMU caches tend to enjoy higher hit rates than TLBs.[5]

Overall, a memory reference first probes the TLB. If there is a TLB hit, the CPU can continue with the post-translation memory reference. If there is a TLB miss, the CPU invokes the PT walker, which generates memory references for L4, L3, L2, and L1 PTs. Each of these references might hit in the MMU cache. However, if they miss, they are sent to the on-chip cache hierarchy from L1 caches to the LLC. Misses in these levels result in a DRAM PT access.

Post-Translation Lookup

Once the translation is found, the post-translation data is looked up. All post-translation memory accesses can be classified into:

- Replay accesses, which correspond to memory references after a TLB miss and subsequent PTW; or
- Regular accesses, which correspond to memory references after a TLB hit.

Both types of accesses can be satisfied from the on-chip caches or DRAM.

ANATOMY OF A MEMORY REFERENCE

Before presenting how TEMPO works, we describe the anatomy of a memory reference that requires a DRAM PTW. We separately discuss DRAM PT lookup and replay.

DRAM PT Access

Figure 2 illustrates the events corresponding to PTWs in blue, and those for data accesses in green. The events are time-ordered from left to right.

TLB and cache lookup

The CPU first accesses the TLB and L1 cache in parallel, as per usual for virtually indexed and physically tagged caches. Suppose that there is a TLB miss. The PT walker (not shown) responds by initiating a multi-level PT lookup. The walker injects memory references for the L4,

L3, and L2 PT entries. Although we omit showing them to simplify Figure 2 assume that these lookups result in MMU cache hits. Subsequently, the PT walker accesses the L1 PT entry. This access prompts the L1 cache and LLC lookups. Assuming misses in both, DRAM lookup commences.

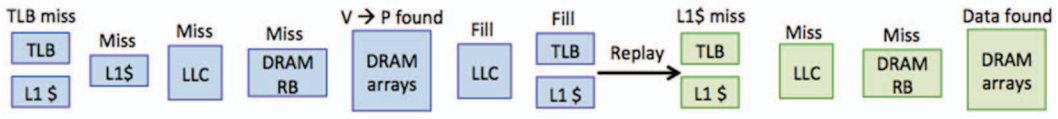

Figure 2. Timeline of events for a memory reference that misses in the TLB. PTWs are shown in blue while the memory replay is shown in green.

DRAM lookup

The processor accesses off-chip DRAM through one or more memory controllers. The controllers orchestrate DRAM device operation using one or more memory channels. DRAM devices are organized as banks of arrays. Arrays are 2D structures of bit-cells identified by row and column number. DRAM array accesses occur at row granularity, using activation, or ACT, commands. DRAM hardware reads the activated row, in units of 4-16 Kbytes, into a row buffer. If the memory controller injects further requests to this row, they are served promptly from the row buffer (a row buffer hit), without DRAM array access delays.

Alternately, the memory controller may request addresses from a different row. Two situations are possible. In the first case, the open row buffer contains array contents from a different row. This is called a row buffer conflict. In response, the memory controller issues a precharge command to write the open row back to the DRAM array, and an ACT command to latch the desired row into the row buffer. This approach places the expensive precharge operation on the critical path of DRAM access. Hence, in the second case, the DRAM logic preemptively closes the open row contents, taking the precharge operation off the DRAM access' critical path. This is called a row buffer miss. Finally, the DRAM controller issues read and write commands to identify the desired column or word from the row buffer. While their latencies vary with process technology and several timing parameters, DDR3 DRAM row buffer hits are generally 10-15 ns, while conflicts and misses are 30-50 ns. Hence, row buffer hits improve access latency by as much as 66 percent.

Figure 2 shows that the L1 PT lookup first checks the row buffer. Unfortunately, PT accesses usually suffer row buffer conflicts or misses. This is because PT accesses are usually interleaved with more frequent accesses to non-PT data. PTs are therefore unlikely to be open in row buffers. On row buffer conflicts or misses, DRAM logic reads the array, where the desired translation (V→P) is found. The translation is relayed to the CPU and is filled into the caches and TLB.

Replay Access to DRAM

On PTW completion, the memory reference is replayed.

TLB and cache lookup

This time, the translation is found in the TLB. However, the replay data is unlikely to be found in any of the caches. Hence, DRAM is again accessed.

DRAM lookup

Unfortunately, more than 98 percent of replays suffer row buffer conflicts or misses. Naturally, if a memory reference suffers a DRAM access for a PTW, it is cold and unlikely to have been accessed sufficiently recently to be open in a DRAM row. Therefore, not only are replays expensive because they usually look up DRAM, but they also often suffer DRAM array latencies.

HIGH-LEVEL APPROACH

Mechanism

Figure 3 shows that TEMPO converts DRAM accesses for replays to LLC hits or row buffer hits. This is accomplished by adding hardware to the memory controller to identify DRAM PT accesses. We also add combinational logic to identify the physical page stored in this translation. This is combined with information about the desired cache line—sent to the memory controller by the PT walker—to identify the post-PT memory address, before the memory replay. Figure 3 shows how this allows two optimizations. First, the 4-16-Kbyte row holding the data needed by the replay is prefetched from the DRAM array into the row buffer. Second, the cache line storing this data is prefetched into the LLC.

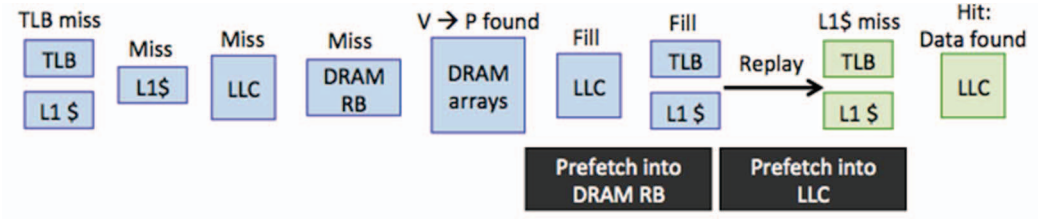

Figure 3. Timeline of events when TEMPO prefetches the data that the replayed instruction will use into the DRAM row buffer and LLC. Subsequent LLC and row buffer hits improve performance.

Benefits

Ideally, TEMPO ensures that replays enjoy LLC hits, eliminating the following time-consuming steps from the critical path of operation:

1. On-chip network traversal from the LLC to the memory controller
2. Memory controller queuing delays
3. DRAM row buffer lookup
4. DRAM row buffer close with precharge
5. Row activation ACT
6. Column read/write
7. Cache fill activities

This can translate to a savings of 100-150+ cycles. Naturally, it is possible (though rare) for the LLC line to be evicted before use. In these cases, row buffer hits still eliminate Steps 4-6.

Prefetching Timeliness

Figure 3 shows that TEMPO's prefetches are overlapped within a window of time where the translation is filled into the caches and TLB, and the replay proceeds until LLC lookup. We make two observations. First, this "slack window" is usually long enough to perform row buffer and LLC prefetching. For example, Intel Haswell and Skylake processors usually take 120+ cycles for these events. In contrast, prefetching from the DRAM array into the row buffer takes 60-100 cycles, while prefetching to the LLC adds another 20-30+ cycles. Second, even when prefetching time exceeds the slack window, prefetching can be partially overlapped, boosting performance. In practice, we find that in scenarios with partial overlap, LLC hits occur less often but DRAM row buffer hits remain prevalent.

Prefetching Accuracy

Classical cache prefetching predicts the addresses of future memory references. Naturally, predictions can be incorrect. Incorrect speculations waste energy, degrade performance, and pollute

caches. TEMPO suffers from none of these problems because it is non-speculative. The memory controller always calculates the replay address correctly.

Hardware Augmentations

PT walker

After the walker finds the upper PT entries, it emits a request for the leaf (such as L1 PT for 4-Kbyte pages) PT entry. We modify the PT walker and tag memory requests for the leaf PT entry with an identifier bit. This bit identifies DRAM PT accesses that should trigger prefetches.

Memory controllers need two pieces of information to determine the prefetch address: the physical page where the replay's requested data resides and the target cache line within the page. The memory controller deduces the former from the L1 translation entry it reads. However, ordinarily, the controller only knows about the latter when the replay request arrives, which is too late to perform prefetching.

In response, we modify the PT walker and append the replay's desired cache line to the memory address of the desired L1 PT entry. In our example, the PT walker appends information about cache line 1, transmitting an additional 6 bits with the memory request for the L1 PT. We discuss these overheads in subsequent sections. (See Figure 4.)

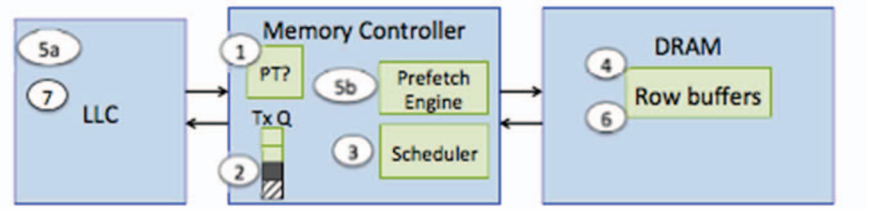

Figure 4. TEMPO detects PT accesses and prefetches post-translation replay data into the row buffer and the LLC.

Memory controller

We minimally extend the memory controller to support two functions:

PT Access Detection and Trigger. Figure 4 shows our modifications of the memory controller. Suppose the memory controller receives a PT request in Step 1. We add a comparator (indicated by the "PT?" block) to identify leaf PT accesses, using the bit identifier set by the PT walker. When such a memory access is detected, the controller inserts this message into the transaction queue (Tx Q). We modify the standard Tx Q to operate as follows. Non-PT requests are inserted into the queue as usual. PT accesses, however, must be handled differently as they have a bigger bit-width than Tx Q entries. This is because we have modified PT walkers to append information about the replay's desired cache line to the PT entry's address. One solution might be to widen each Tx Q's bit-width. Unfortunately, this increases the size of the queue by roughly 25 percent, according to our circuit-level modeling. Instead, we break the PT access into two Tx Q transactions in Step 2. The first one (in striped black) represents the PT access, while the second one (in solid black) temporarily buffers information about the replay cache line, to be used shortly to construct the prefetch target. Next, in Step 3, the DRAM schedules PT access. The controller sends ACT commands to read the DRAM array row containing the desired PT entry into the row buffer in Step 4. Finally, the requested cache line is filled in the LLC in Step 5(a).

Prefetch of Replay Data. TEMPO prefetches post-translation replay data in parallel with LLC fill. We add a finite state machine (the prefetch engine) to accomplish this in Step 5(b). The prefetch engine identifies the desired 8-byte PT entry from the requested PTW access and extracts the physical page number residing in it. This corresponds to the physical page number of the replay's memory access. The prefetch engine logic then concatenates this physical page

number with the replay access's cache line information stored in the Tx Q (the solid black entry in Figure 4). The result is the replay's memory reference address. The memory controller sends a read request for this address to the DRAM device. In response, in Step 6, the row containing the prefetch target is latched into the row buffer. Further, the cache line containing the prefetch target is sent to the LLC in Step 7.

Hardware overheads

We have synthesized the additional PT walker and memory controller hardware and modeled the impact of the PT walker's larger message sizes. We see that PT walkers become 0.5 percent bigger, memory controllers become 3 percent bigger, and there is a negligible increase in on-chip network bandwidth usage from the larger messages.

Interactions with Traditional Cache Prefetching

TEMPO is orthogonal to classical cache prefetchers**.** Specifically, we have studied TEMPO's interactions with the recently proposed IMP prefetcher.[18] IMP is designed to prefetch irregular memory accesses from indirect patterns of the form A[B[i]]. We find that TEMPO's performance benefits become even more pronounced with IMP for two reasons. First, IMP generates many DRAM PT accesses as it prefetches across page boundaries; workloads with irregular memory accesses therefore easily thrash TLBs. TEMPO mitigates the post-translation replay access bottlenecks for these workloads. Second, IMP successfully prefetches many non-PT cache lines, leaving DRAM PT accesses and replay accesses as performance bottlenecks. Overall, TEMPO improves the performance of systems using IMP by as much as 40 percent, going beyond the10-30-percent performance improvements of systems without prefetching.

Interactions with Memory and Row Buffer Scheduling

Vendors and researchers have proposed hardware support for many memory scheduling and row buffer management policies in recent years.[19] TEMPO is complementary to all of them. We omit a discussion of these techniques in this manuscript but point readers to the original article[17] for more details.

EVALUATION

We point interested readers to our original article for a detailed discussion of the software simulation-based methodology that we use to evaluate TEMPO.[17] We summarize some key results from our analysis in this section.

Performance and Energy Improvements

Figure 5 (upper left) summarizes the performance (blue) and energy (green) benefits of TEMPO. Our results are measured as a fraction of the baseline execution without TEMPO, with higher numbers being better. Figure 5 (upper right) shows the fraction of memory footprint devoted to 2-Mbyte superpages, measured on the real system from which we collect traces.

Performance benefits

Figure 5 (upper left) shows that TEMPO consistently improves our big-data workloads. Workloads with frequent DRAM PT accesses (such as xsbench) enjoy close to 30 percent performance boosts. Generally, the poorer the access locality of the workload, the more useful TEMPO is. These benefits exist even in the presence of fairly aggressive superpage use.

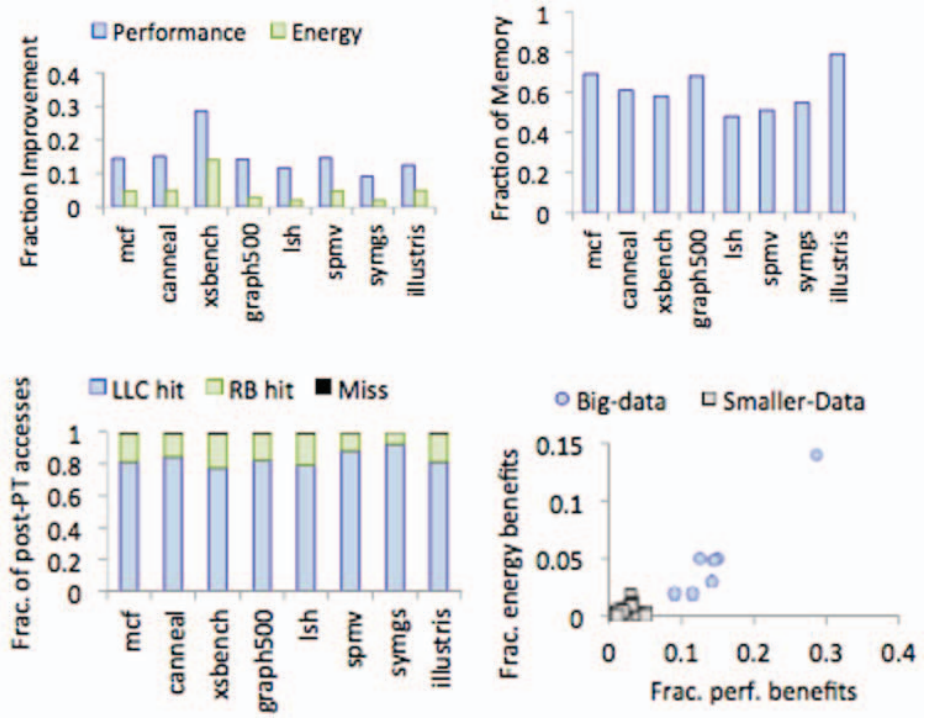

Figure 5. (Upper left) improvements in performance and energy using TEMPO as a fraction of the baseline execution; (upper right) fraction of memory footprint devoted to 2 Mbyte superpages; (bottom left) fraction of replays serviced from the LLC and row buffer; and (bottom right) energy/performance graphs comparing big-data workloads with Spec/Parsec workloads with smaller memory footprints.

Energy benefits

Figure 5 (upper left) shows that TEMPO saves energy, despite its area overheads, by speeding up execution and hence reducing static energy. We see 1-14-percent energy savings, trending similarly with performance improvements.

Breakdown of benefits

Figure 5 (bottom left) distinguishes the benefits of row buffer and LLC prefetching. For each workload, we separate the fraction of post-PT accesses that see hits in the LLC (blue) or the row buffer (green) due to prefetching. Note the tiny presence of a third category where TEMPO cannot aid replay accesses. These typically occur only during pathological cases when there are just too many PT accesses queued at the memory controller for the prefetches at the Tx Q tail to complete in a timely fashion. The bulk of the replays' accesses (75 percent and higher) see LLC hits, and most LLC misses become row buffer hits.

Workloads with small memory footprints

Our results thus far have focused on big-data workloads with poor locality of memory access. This is because TEMPO initiates prefetches only when workloads use memory sufficiently aggressively to initiate many DRAM PT accesses. We also, however, need to ensure that TEMPO does not harm workloads with smaller memory footprints (in other words, that the additional hardware does not significantly compromise system energy). Figure 5 (bottom right) presents the results of this study, where we separate the energy and performance characteristics of the big-data workloads (blue) with those of the remaining smaller-footprint Spec and Parsec workloads. As expected, big-data workloads benefit most; however, not a single smaller-footprint workload

becomes slower or consumes more energy. Instead, performance improves by 1-2 percent and energy by roughly 1 percent.

Interactions with cache prefetching

Our original article quantifies TEMPO's benefits in the presence of IMP prefetchers, which prefetch data into the L1 cache. We find that workloads with especially poor memory access locality (such as xsbench and smpv) are particularly aided, seeing almost 10-percent performance improvements from the no-prefetching case. These energy savings track our performance benefits; because application runtime is further reduced, static energy is mitigated.

Interactions with superpages and memory organizations

In our original article,[17] we also quantify how TEMPO's benefits evolve as superpages are used more aggressively. As one might expect, the more frequently the OS generates superpages, the less pronounced the performance improvements. Nevertheless, our workloads use so much data that even when superpages are used to back more than 90 percent of the application's data, TEMPO can offer 5-10-percent performance benefits. Similarly, our original article also assesses how TEMPO improves performance and energy as the memory organization varies.[17] We find that, regardless of row/sub-row buffer organization open/closed/adaptive row policies, TEMPO achieves performance improvements.

CONCLUSION

This work introduces TEMPO, a hardware-only augmentation of the PT walker and memory controller to DRAM accesses from replayed instructions off the critical path of execution. TEMPO prefetches this data into the row buffer and the LLC. We show that this approach improves performance and energy considerably for workloads with sparse memory access patterns, without compromising smaller-data workloads.

ACKNOWLEDGMENTS

We thank Guilherme Cox, Jan Vesely, and Zi Yan for discussions on the technical content of this work. We also thank Daniel Lustig, Tushar Krishna, Carole-Jean Wu, and Mikko Lipasti for their feedback on drafts of this work. Most importantly, we thank Shampa Sanyal, who made this work possible.

REFERENCES

1. Z. Yan et al., "Hardware Translation Coherence for Virtualized Systems," *International Symposium on Computer Architecture* (ISCA), 2017.
2. A. Bhattacharjee and M. Martonosi, "Inter-Core Cooperative TLB Prefetchers for Chip Multiprocessors," *International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS), 2010.
3. G. Cox and A. Bhattacharjee, "Efficient Address Translation for Architectures with Multiple Page Sizes," *International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS), 2017.
4. B. Pham et al., "Large Pages and Lightweight Memory Management in Virtualized Environments: Can You Have It Both Ways?," *International Symposium on Microarchitecture* (MICRO), 2015.
5. A. Bhattacharjee, "Large-Reach Memory Management Unit Caches," *International Symposium on Microarchitecture* (MICRO), 2013.
6. B. Pham et al., "CoLT: Coalesced Large-Reach TLBs," *International Symposium on Microarchitecture* (MICRO), 2012.

7. A. Bhattacharjee, D. Lustig, and M. Martonosi, "Shared Last-Level TLBs for Chip Multiprocessors," *International Symposium on High Performance Computer Architecture* (HPCA), 2011.
8. T. Barr, A. Cox, and S. Rixner, "SpecTLB: A Mechanism for Speculative Address Translation," *International Symposium on Computer Architecture* (ISCA), 2011.
9. V. Karakostas et al., "Redundant Memory Mappings for Fast Access to Large Memories," *International Symposium on Computer Architecture* (ISCA), 2015.
10. A. Basu et al., "Efficient Virtual Memory for Big Memory Servers," *International Symposium on Computer Architecture* (ISCA), 2013.
11. J. Gandhi et al., "Efficient Memory Virtualization," *International Symposium on Microarchitecture* (MICRO), 2014.
12. T. Barr, A. Cox, and S. Rixner, "Translation Caching: Skip, Don't Walk (the PT)," *International Symposium on Computer Architecture* (ISCA), 2010.
13. J. Navarro et al., "Practical, Transparent Operating System Support for Superpages," *Symposium on Operating Systems Design and Implementation* (OSDI), 2002.
14. M. Talluri et al., "Tradeoffs in Supporting Two Page Sizes," *International Symposium on Computer Architecture* (ISCA), 1992.
15. M. Talluri and M. Hill, "Surpassing the TLB Performance of Superpages with Less Operating System Support," *International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS), 1994.
16. R. Bhargava et al., "Accelerating Two-Dimensional Page Walks for Virtualized Systems," *International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS), 2008.
17. A. Bhattacharjee, "Translation-Triggered Prefetching," *International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS), 2017.
18. X. Xu et al., "IMP: Indirect Memory Prefetcher," *International Symposium on Microarchitecture* (MICRO), 2015.
19. O. Mutlu, "Memory Scaling: A Systems Architecture Perspective," *MemCon*, 2015.
20. V. Seshadri et al., "Page Overlays: An Enhanced Virtual Memory Framework to Enable Fine-Grained Memory Management," *International Symposium on Computer Architecture* (ISCA), 2015.

ABOUT THE AUTHOR

Abhishek Bhattacharjee is a professor of computer science at Rutgers University. His primary research interest is the hardware/software interface, with a focus on architectural support for operating systems abstractions. He has a PhD in electrical engineering from Princeton University. Contact him at abhib@cs.rutgers.edu.