Translation-Triggered Prefetching

Abhishek Bhattacharjee

Department of Computer Science, Rutgers University

abhib@cs.rutgers.edu

Abstract

We propose **translation-enabled memory prefetching optimizations** or **TEMPO**, a low-overhead hardware mechanism to boost memory performance by exploiting the operating system's (OS') virtual memory subsystem. We are the first to make the following observations: (1) a substantial fraction (20-40%) of DRAM references in modern big-data workloads are devoted to accessing page tables; and (2) when memory references require page table lookups in DRAM, the vast majority of them (98%+) also look up DRAM for the subsequent data access. TEMPO exploits these observations to enable DRAM row-buffer and on-chip cache prefetching of the data that page tables point to. TEMPO requires trivial changes to the memory controller (under 3% additional area), no OS or application changes, and improves performance by 10-30% and energy by 1-14%.

CCS Concepts **• Computer systems organization → Pipeline computing**; **Multicore architectures**

Keywords Virtual memory, cache prefetching, DRAM.

1. Introduction

Memory accesses represent a performance bottleneck [1, 4, 5]. Modern systems execute memory-intensive workloads with sparse data structures, big key-value stores, graph analytics, scientific processing algorithms, and multi-dimensional data sets, all with complex memory access patterns. Consequently, processor vendors and researchers are investigating DRAM optimizations to improve bandwidth and latency [6–14], boost row buffer management policies [15–18], make fairer and faster memory schedulers [19–24], and improve OS-DRAM interactions [25, 26]. Recent work has also proposed better on-chip cache replacement and prefetching to eliminate DRAM lookups for workloads with input-dependent and irregular memory accesses [27–29].

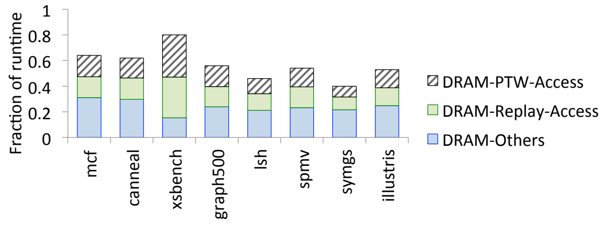

Figure 1. Fraction of total application runtime for page table accesses to DRAM (DRAM-PTW-Access), replayed accesses to DRAM (DRAM-Replay-Access), and other non-page table DRAM accesses (DRAM-Other).

While effective, these efforts have mostly ignored a crucial part of the system stack – virtual memory. Virtual memory automates memory and storage management, and also provides the benefits of memory protection. Processors supporting virtual memory translate program-visible virtual memory addresses to physical memory addresses. Virtual-to-physical translation mappings are maintained, in units of memory pages, by the OS in a software data structure called the page table. Like any software data structure, the page table resides in the on-chip cache hierarchy and DRAM.

Architects implement Translation Lookaside Buffers (TLBs) to cache virtual-to-physical translations. When a translation is absent in the TLB (a situation called a TLB miss), a hardware page table walker looks up or "walks" the page table. Memory references for page table walks are serviced from on-chip caches or off-chip DRAM. After the page table walk completes, the TLB is filled and the memory reference is *replayed*. The replayed access hits in the TLB and probes the memory hierarchy with the physical address.

One might expect that since address translation has been extensively optimized [2, 30–41] and as TLBs have been growing progressively bigger, virtual-to-physical translation requests usually hit in TLBs or on-chip caches, and rarely access DRAM. We find, however, that this is not the case in big-memory servers. Consider Figure 1, which uses the simulation-based methodology of Sec. 5 to quantify the fraction of application runtime expended on DRAM lookups. We model a 32-core Intel Skylake system with 4TBs of DRAM, running Ubuntu Linux with a v4.4 kernel and trans-

ASPLOS '17, April 08-12, 2017, Xi'an, China

© 2017 ACM. ISBN 978-1-4503-4465-4/17/04...$15.00

DOI: http://dx.doi.org/10.1145/3037697.3037705

parent support for superpages [42]. We split DRAM overheads into three categories: ① DRAM accesses to the page table (DRAM-PTW-Access); ② DRAM accesses for the replayed memory reference (DRAM-Replay-Access); and ③ other DRAM accesses (DRAM-Other).

Figure 1 shows that a large fraction of DRAM overheads arises from page table walk accesses. This is because our workloads are dominated by irregular memory accesses, following graph edges or non-zero elements in sparse matrices. These applications access their massive 4TB address spaces sparsely, frequently triggering TLB misses.

Figure 1 reveals, however, an opportunity to improve DRAM performance. Past work has focused on reducing a memory reference's probability of suffering a TLB miss, or reducing its page table walk latency. We have, however, ignored the overhead of replaying the memory reference *after* the page table walk. Figure 1 shows that *DRAM accesses for replayed data (in green) constitute 10-30% of runtime*, almost as much as the time taken for the prior DRAM page table accesses. The intuition for this is as follows. Suppose a memory reference looks up the TLB for its translation and misses. This translation points to a physical page hosting the data ultimately needed. A TLB miss indicates that the translation is cold. Even further, if the subsequent page table walk accesses DRAM – because the translation is absent from the on-chip caches – the translation is likely very cold. Hence, the data pointed to by the translation is likely even colder, and most probably resides in DRAM itself. In other words, a single memory reference generates back-to-back DRAM accesses, once for the page table lookup, and once for the data. We observe that this occurs often (see Figure 1) and that for every workload, over 98% of DRAM page table lookups are followed by DRAM lookups for replayed accesses.

In response, we propose **translation-enabled memory prefetching optimizations** or **TEMPO**. TEMPO leverages DRAM page table accesses to virtually eliminate the overheads of replayed accesses to DRAM. TEMPO improves performance with two optimizations. First, TEMPO performs DRAM row buffer prefetching. We add simple hardware in the memory controller to identify DRAM accesses for page tables, and deduce the physical memory address the page table entry points to. TEMPO prefetches this data into the DRAM row buffer. This allows DRAM accesses for replays to enjoy row buffer hits, boosting performance. Second, TEMPO prefetches this data into the last-level cache (LLC). Together, these approaches improve performance by 10-30%, energy by 1-14%, add negligible area (only 3% to the memory controller), and require no OS or application changes. Overall, our contributions are:

- We characterize the impact of address translation on main memory performance. We discover that DRAM accesses for post-page table walk replays constitute a significant fraction of system runtime.

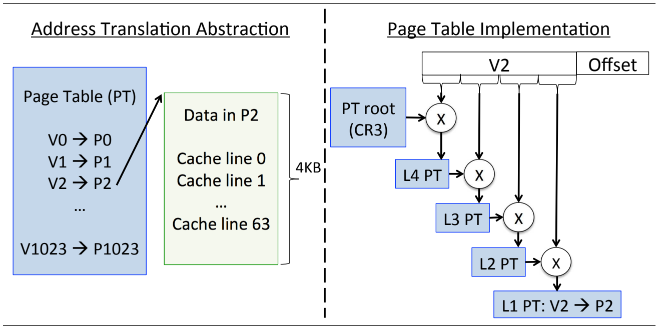

Figure 2. (Left) Page tables maintain virtual to physical page translations. The post-translation physical page consists of several cache lines (e.g., 64 cache lines for 4KB x86-64 pages); and (Right) multi-level radix tree implementations of x86-64 page tables.

- We propose TEMPO to attack replay DRAM accesses, eliminating their overheads almost entirely by using row buffer and LLC prefetching.
- We assess TEMPO's benefits on: ⓐ several state-of-the-art memory schedulers for performance [43] and fairness [23, 24]; ⓑ open, closed, and adaptive row buffer management policies [16, 17]; ⓒ alternative row buffer organizations; and ⓓ recent cache prefetching schemes for irregular memory access patterns [44]. TEMPO remains effective – and often boosts performance even more – when processors employ these optimizations.

Since TEMPO requires modest hardware support, we expect it to be readily-implementable in upcoming processors.

2. Motivation and Background

We begin this section by detailing OS and hardware support for address translation, and our goals. Though TEMPO benefits all architectures, we focus on x86-64 systems.

2.1 Address Translation and Data Interactions

Figure 2 shows that all memory references require two steps: first, a virtual-to-physical translation and second, a post-translation lookup. We now detail both:

Virtual-to-physical address translation: The diagram on the left of Figure 2 shows that all virtual addresses are grouped into virtual pages. Abstractly, a page table maps each virtual page to a physical page. For example, virtual page 2 maps to physical page 2. The physical page stores cache lines. For x86-64 systems using a base page size of 4KB, a page stores 64 distinct 64-byte cache lines.

The diagram on the right of Figure 2 shows an example page table implementation. x86-64 systems use a multi-level forward-mapped radix tree to represent the page table. We refer readers to prior work for details on x86-64 page tables [30, 45–49], but briefly discuss how page table walkers traverse them. At a high-level, there are four page table levels, which we refer to as L4-L1 page tables (PTs), similar to prior work [45, 46]. x86-64 systems use them to support 48-bit virtual addresses today. The first step in a walk is to concatenate the root physical address of the L4 PT (stored in the CR3 register) with the upper-most 9 bits of the virtual page number. Hardware page table walkers look up the L4 PT with this concatenated address, to extract the physical address of the base of the L3 PT. This address is concatenated with the next 9 bits of the virtual page number to look up the L3 PT. This process repeats until the L1 PT is probed to identify the desired virtual-to-physical translation. In Figure 2, we thus discover that V2 is mapped to P2 in the L1 PT.

Since page table walks require 4 sequential (and hence expensive) memory references, CPUs use two types of hardware structures to accelerate translation lookup. The first is the TLB, which stores virtual-to-physical translations; in other words, it caches frequently-used entries from the L1 PT. The second is a family of MMU caches, which accelerate page table walks when TLBs miss [45,46,50]. MMU caches store frequently-used entries from the L4, L3, and L2 PTs. They are generally smaller (by 32× in Skylake processors) than TLBs since L4, L3, and L3 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 better hit rates than TLBs [46].

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 page table walker, which generates memory references for L4, L3, L2, and L1 PTs. Each of these references may hit in the MMU cache. However, if they miss, they are sent to the on-chip cache hierarchy, made up of L1 caches to the LLC. LLC misses result in DRAM page table accesses.

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 page table walk; and ⓑ 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.

2.2 Our Goal

TEMPO's goal is to decrease the latency of DRAM accesses for replays. Figure 3 illustrates this. The x-axis plots the latency of the page table access. TEMPO is triggered when DRAM needs to be accessed for the page table lookup. By prefetching into the row-buffer alone, TEMPO expedites the replay access latency since row buffer hits generally cut DRAM access time by as much as 66% (see Sec. 2.3). By further prefetching data into the LLC, access times for replays are reduced to the latency of an LLC hit.

Figure 4 quantifies the fraction of DRAM references amenable to TEMPO. Figure 4 shows that DRAM page table accesses (DRAM-PTW-Access) constitute as much

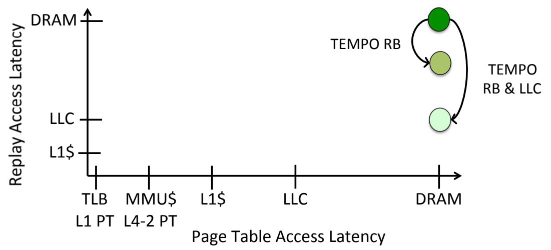

Figure 3. TEMPO prefetches into the row buffer and LLC to decrease replay access latency.

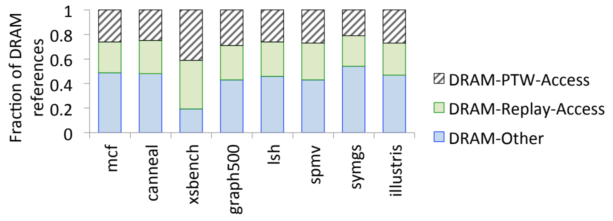

Figure 4. Fraction of total DRAM references devoted to page table walk accesses to DRAM (DRAM-PTW-access), replay accesses to DRAM (DRAM-Replay-Access), and other non-page table DRAM accesses (DRAM-Other).

as 20-40% of total DRAM references. This means that TEMPO's prefetching will be triggered often. Further, Figure 4 shows that almost an equivalent number of DRAM references are devoted to replay accesses (DRAM-Replay-Access). This means that each prefetch will likely be useful.

Now, consider Figure 4 more closely. One might expect DRAM-PTW-Access to be a bigger fraction of the total DRAM references than DRAM-Replay-Access. After all, each page table walk requires four memory references while the replay data access requires merely one (see Sec. 2.1). However, we find that leaf PT accesses (e.g., L1 PTs for 4KB pages) suffer much worse reuse than upper-level PT accesses. Consequently, 96%+ of all DRAM page table walk accesses (or the striped bars in Figure 4) are for leaf PTs. And since poor reuse for leaf PTs implies that the physical pages they point to have poor reuse too, the replay (the green bars in Figure 4) usually results in a DRAM access too.

One might also expect superpages to reduce the frequency of DRAM page table accesses, and hence TEMPO's benefits. However, Figure 4 is collected on a system where Linux support for transparent 2MB superpages is enabled [42]. Our experiments show that the Linux allocates more than half the memory footprint with 2MB superpages (see Sec. 6), for every workload. Still, our workloads are so memory-intensive (using most of the 4TB physical memory) and look up DRAM with such irregular access patterns and poor locality, that DRAM page table accesses remain common. Sec. 6 studies this further, showing TEMPO's benefits when 2MB and 1GB superpages are used.

2.3 Anatomy of a Memory Reference

Before presenting TEMPO's hardware, we describe the anatomy of a memory reference that requires a DRAM page table walk. We separately discuss the DRAM page table lookup and replay.

DRAM page table access: Figure 5 illustrates the events corresponding to page table walks in blue, and those for the subsequent data access in green. The events are time-ordered from the 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 [34, 51]. Suppose that there is a TLB miss. The page table walker (not shown) responds by initiating a multi-level page table lookup. The walker injects memory references for the L4, L3, and L2 page table entries. Although we omit showing them to simplify Figure 5, let us assume that these lookups result in MMU cache hits. Subsequently the page table walker accesses the L1 page table entry. This looks up the L1 cache and LLC. Assuming misses in both, DRAM lookup commences.

ⓑ DRAM lookup: The processor accesses off-chip DRAM via one or more on-chip memory controllers. The controllers orchestrate DRAM device operation using one or more channels. DRAM devices are organized as banks of arrays (see recent work for details [6–18]). Arrays are two-dimensional 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 4-16KB, into a *row buffer* [17, 18]. 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 desire 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 pre-emptively 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 10-15ns, while conflicts and misses are 30-50ns [7, 15]. Hence, row buffer hits improve acces latency by as much as 66%.

Figure 5 shows that the L1 page table lookup first checks the row buffer. Unfortunately, page table accesses usually suffer row buffer conflicts or misses. This is because page table accesses are usually interleaved with more frequent accesses to non-page table data. Page tables are therefore unlikely to be open in row buffers. On row buffer conflicts or misses, DRAM logic reads the array, where the desired translation ($V \rightarrow P$) is found. The translation is relayed to the CPU, and is filled the caches and TLB.

Replay access to DRAM: On page table walk completion, the memory reference is replayed.

ⓐ TLB and cache lookup: This time, the translation is found in the TLB. However, for the reasons described in Sec. 2.2, the replay data is unlikely to be found in any of the caches. Hence, DRAM is again accessed.

ⓑ DRAM lookup: Unfortunately, 98%+ of replays suffer row buffer conflicts or misses. Naturally, if a memory reference suffers a DRAM access for a page table walk, it is cold and is unlikely to have been accessed sufficiently recently to be open in a DRAM row. Therefore, not only are these replays expensive because they look up DRAM, they also usually suffer DRAM array lookup latencies.

3. High-Level Approach

We now explain, at a high-level, TEMPO's mechanism and how it aids performance.

Mechanism: Figure 6 explains how TEMPO works. The goal is to convert DRAM accesses for replays to LLC hits or row buffer hits. Consequently, we add hardware to the memory controller to identify DRAM page table 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 page table walker – to identify the post-page table memory address, *before* the memory replay. Figure 6 shows that this allows two optimizations. First, the 4-16KB 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.

Benefits: Ideally, TEMPO ensures that replays enjoy LLC hits, eliminating: ⓐ on-chip network traversal from the LLC to the memory controller; ⓑ memory controller queueing delays; ⓒ DRAM row buffer lookup; ⓓ DRAM row buffer close with PRECHARGE; ⓔ row activation ACT; ⓕ column READ/WRITE; and ⓖ cache fill activities. This can translate to a savings of 100-150+ cycles. Naturally, it is possible (though rare, as we show in Sec. 6) for the LLC line to be evicted before use; in these cases, row buffer hits may still occur, eliminating ⓓ-ⓕ.

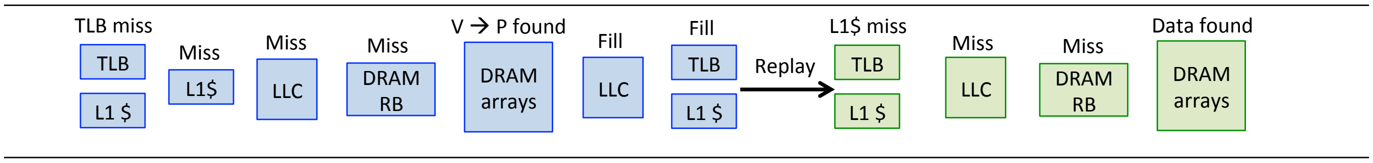

Figure 5. Timeline of events for a memory reference that misses in the TLB. Page table walks are shown in blue the memory replay is shown in green.

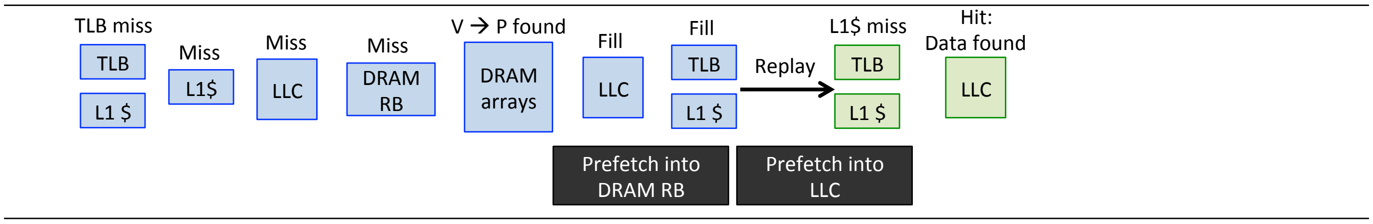

Figure 6. Timeline of events when TEMPO prefetches the data that the replayed instruction will use into the DRAM row buffer and LLC. Subsequent LLC and row buffer hits improve performance.

Prefetching timeliness: Figure 6 shows that TEMPO's prefetches are overlapped within a *slack window* where the translation is filled into the caches and TLB, and the replay proceeds till LLC lookup. We make two observations. First, the slack window is usually long enough to perform row buffer and LLC prefetching. For example, Intel Haswell and Skylake processors usually take 120+ cycles [52,53] 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 future memory references and hence *speculates* on their target addresses, possibly incorrectly. Incorrect speculations waste energy, degrade performance, and needlessly pollute caches. TEMPO suffers from none of these problems because it is *non-speculative*. The replay's memory address is always calculated correctly by the memory controller.

4. Hardware Design

TEMPO requires hardware enhancements to page table walkers and the on-chip memory controller. No changes are needed to the OS or application.

4.1 Hardware Augmentations

We separate our discussion of hardware changes to the page table walker and the memory controller.

Page table walker: Consider a page table walk. After the walker finds the upper PT entries, it emits a request for the leaf (e.g., L1 PT for 4KB pages) PT entry. We modify the page table walker and tag memory requests for the leaf PT entry with an identifier bit. This bit identifies DRAM page table accesses that should trigger prefetches.

Memory controllers need two pieces of information to determine which address to prefetch from: ⓐ the physical page where the replay's requested data resides, ⓑ and the target cache line within the page. The memory controller deduces ⓐ from the L1 translation entry it reads (with hardware that we present in the next subsection). However, ordinarily, the controller only knows about ⓑ when the replay request arrives, which is too late to perform prefetching.

Suppose, for example, that a program using the page table in Figure 2 accesses virtual address 0x2001. In other words, the application requests cache line 0 from virtual page 2. Further, suppose that the mapping V2→P2 is absent from the TLB. The page table walker accesses the L4-L2 levels, and discovers the base physical address of the L1 page table. It then sends this physical address, concatenated with the least significant 9 bits of the virtual page number (0x02) to the memory hierarchy. If this access reaches DRAM, the memory controller can use it to look up the L1 page table and determine that V2 maps to P2. However, it has no way to know that the initial memory reference – and hence the replay – is to 0x2001 and hence accesses cache line 0 within P2. Without this knowledge, TEMPO cannot prefetch.

In response, we modify the page table walker and append the replay's desired cache line to the memory address of the desired L1 PT entry. In our example, the page table walker appends information about cache line 0, transmitting an additional 6 bits with the memory request for the L1 PT. We discuss these overheads in subsequent sections.

Memory controller: We extend the memory controller to support two basic functions:

ⓐ Page table access detection and trigger: Figure 7 shows our modifications of the memory controller. Suppose the

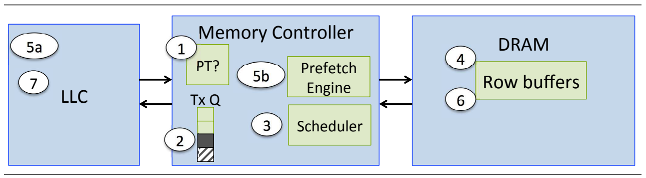

Figure 7. TEMPO detects page table accesses and prefetches post-translation replay-data into the row buffer and LLC.

memory controller receives a page table request in ①. We add a comparator (indicated by the PT? block) to identify leaf page table accesses, using the bit identifier set by the page table 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-page table requests are inserted into the queue as per usual. Page table accesses, however, must be handled differently as they have a bigger bit-width than Tx Q entries. This is because we have modified page table walkers to append information about the replay's desired cache line to the page table entry's address.

One solution may be to widen each Tx Q's bitwidth. Unfortunately, this increases the size of the queue by roughly 25%, according to our RTL modeling (see Sec. 5). Instead, we break the page table access into two Tx Q transactions in ②. The first one (in striped black) represents the page table 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 page table access is scheduled in ③. The controller sends ACT commands to read the DRAM array row containing the desired page table entry into the row buffer ④. The requested cache line is filled in the LLC ⑤ⓐ.

ⓑ Prefetch of replay data: TEMPO prefetches post-translation replay data in parallel with LLC fill. We add a simple finite state machine (the Prefetch Engine) to accomplish this in ⑤ⓑ. The Prefetch Engine identifies the desired 8-byte page table entry from the requested page table walk 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 concatenates this physical page number with the the replay access' cache line information stored in the Tx Q (the solid black entry in Figure 7). The result is the replay's memory reference address. The controller sends a read request for this address to the DRAM device. In response 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 ⑦.

Hardware overheads: We have synthesized the additional page table walker and memory controller hardware (see Sec. 5 for details), and modeled the impact of the page table walker's larger message sizes. Overall, we see that page table walkers become 0.5% bigger, memory controllers become 3% bigger, and there is a negligible increase in on-chip network bandwidth usage from the larger messages. Therefore, TEMPO's benefits far outweigh its modest overheads.

4.2 Interactions with Traditional Cache Prefetching

TEMPO operates orthogonally to classical cache prefetchers [3, 27–29, 44]. Some of these studies have focused on prefetching for pointer-intensive programs [3], which shares some conceptual similarities to the notion of prefetching the replay data pointed to by the page table. However, none of these studies showcase the impact of virtual memory on cache prefetching techniques.

We have studied TEMPO's interactions with the IMP prefetcher [44]. IMP is designed to prefetch irregular memory accesses from indirect patterns of the form *A[B[i]]* [44]. We find that TEMPO's performance benefits become even more pronounced with IMP for two reasons. First, IMP generates many DRAM page table 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-page table cache lines, leaving DRAM page table accesses and replay accesses as performance bottlenecks. Overall, TEMPO improves the performance of systems using IMP by as much as 40%, going beyond its 10-30% performance improvements of systems without prefetching.

4.3 Interactions with Memory and Row Buffer Scheduling

Researchers have recently proposed hardware support for several memory schedulers [19–24, 43] and row buffer management policies [15, 17, 54] in recent years. TEMPO functions efficiently with any of them. These policies use either *closed* or *open/adaptive* row buffer management.

Closed row buffer management: In this approach, once a row is opened and read, it is immediately written back to the DRAM array [17]. Therefore, all row buffer lookups result in misses. However, by pre-emptively writing back the row to the DRAM array, the PRECHARGE latency is taken off the critical path of future memory references. Closed row policies are beneficial when memory accesses have poor locality (e.g., in many-core systems when multiple memory access streams are interleaved in DRAM, destroying intra-stream locality). TEMPO boosts performance in these cases. Sec. 6 shows these performance benefits.

Open row buffer management: Instead of immediately closing a row's contents, an alternative is to leave the row open for subsequent DRAM requests. If these DRAM requests access the same row, row buffer hits boost performance. Past work has proposed several optimizations to schedule DRAM requests such that row buffer hits are promoted. For example, the classic FR-FCFS scheduler reorders DRAM requests to strike a balance between the request age

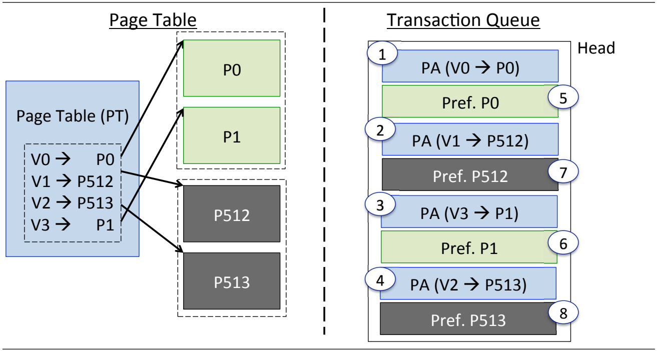

Figure 8. Memory scheduling policies in the presence of multiple page table accesses and replay prefetches.

and its likelihood of enjoying a row buffer hit [43]. Similarly, other schedulers promote row buffer hits for parallel workloads [20].

Although the TEMPO design presented thus far operates seamlessly with open row buffer policies, we can further enhance it to balance row buffer hits for page table accesses, while also ensuring that the prefetches they trigger occur in a timely manner. There are two situations of interest:

ⓐ Single page table access queued: Suppose that the Tx Q holds several memory access requests, but only one of them is a page table access. Suppose further that the DRAM scheduler loads the row holding the desired translation into the row buffer. At this point, TEMPO can immediately initiate the prefetch, closing the current row buffer contents. However, we have found that instead of prefetching immediately, it is beneficial to leave the row buffer contents open for a few cycles. This approach *anticipates* the arrival of additional page table requests; if these requests are to page table entries that map to the same 4-16KB row contents, delaying the prefetch by a few cycles potentially boosts row buffer hits. Naturally, the delay ought to be judicious – excessive waiting times overly delay prefetches, counteracting TEMPO's benefits. We have found that a 10-cycle delay boosts performance by 1-4% over baseline TEMPO, across all applications, but particularly for those that suffer from frequent DRAM page table accesses (e.g., xsbench).

ⓑ Multiple page table accesses queued: Multiple page table walk requests – and their associated prefetch requests – may be present in the Tx Q. Our goal is to schedule the requests to boost row buffer hits overall. Consider, for example, Figure 8. On the left, we show a page table mapping V0-3 to several physical pages. Further, we show, using dashed boxes, data that maps to the same DRAM row. For example, since the translations for V0-3 are 8 bytes each, they map to one row, shown in blue. However, the physical pages they point to map to separate rows. Suppose that row buffers store 8KB of data. In this case, spatially-adjacent physical pages P0-1 share a row, while P512-513 share a row. Further, on the right, Figure 8 shows how requests for these translations are queued in the Tx Q. Suppose that the page table walker sends DRAM requests for the translations of V0, V1, V3, and V2 by requesting their physical addresses (PA). The memory controller detects that these are page table requests and hence also queues prefetches (i.e., Pref. P0, etc.). As usual, these prefetch requests store only the target cache line from the physical page, until the translation is looked up from the DRAM array.

A naive scheduling policy schedules transactions in order from the queue's head to the tail. However, this needlessly exacerbates row buffer conflicts. For example, if PA(V0-P0) is followed by the prefetch of the cache line from P0, the subsequent PA(V1-P512) request experiences a row conflict, even though it maps to the same row as PA(V0-P0). Instead, TEMPO scans the Tx Q for page table access requests. Since page table requests usually lie on the critical path of execution, Tx Q entries for translations residing on *the same row* are first scheduled. Figure 8 shows this in ①-④.

Once the page table requests are scheduled, TEMPO schedules the prefetch requests. Again, row buffer hits are promoted by scheduling groups of prefetches that map to the same row. Since we assume 8KB rows and 4KB base pages, prefetch requests to cache lines in P0 and P1 are first handled ⑤-⑥, followed by P512 and P513 ⑦-⑧.

A potential concern is that handling multiple page table requests before initiating prefetches jeapardizes the latter's timeliness. We find that this is not a problem for several reasons. First, the additional page table requests are row buffer hits and can hence be handled quickly, without excessively delaying prefetches. Second, page table walks lie on the critical path of program execution and must hence be handled fast to enhance overall performance. Third, the slack window within which to prefetch is usually long enough to tolerate some prefetch delay. And finally, even if prefetches are somewhat delayed, they still provide *some* performance boost; in the worst cases, we find that replays may not enjoy LLC hits, but they still usually enjoy row buffer hits.

Adaptive row buffer management: Adaptive row policies combine the best of open and closed row policies, boosting row buffer hit rates and on misses, converting potential row buffer conflicts to merely misses (where the PRECHARGE operation sits off the critical path of execution). These policies use hardware predictors with saturating counters to predict the length of time a row should be left open for good performance. TEMPO boosts the performance of adaptive row policies from prior work [17] in Sec. 6.

Scheduling for fairness: TEMPO also improves the performance of memory schedulers that balance application fairness with performance, for workloads made up of multiple applications. Without loss of generality, we study the blacklisting memory scheduler or BLISS [23, 24].

BLISS mitigates inter-application interference by separating applications into two groups: first, a group of applications vulnerable to interference; and second, a group of applications that cause interference. BLISS counts the number of consecutive DRAM requests from each application to classify each application into one of these groups. Applications with more consecutive requests are classified as interference-causing. We rethink two design decisions to operate TEMPO efficiently atop BLISS:

ⓐ Counter increments: TEMPO performs two transactions on DRAM page table lookups: one for the translation lookup, and one for the post-translation prefetch. An important question is whether prefetches should be included in BLISS' per-CPU counter increments. On one hand, if a CPU performs DRAM page table walks that initiate prefetches, these prefetches constitute additional memory traffic generated by that CPU. On the other hand, prefetches are not technically on-demand, they are performance optimizations that are used to reduce the latency of future memory references. Therefore, it is not clear that they should be counted as *consecutive* memory references. Overall, we find that the best balance is to indeed increment counters on prefetches, but with a half the weight as a non-prefetch access. In other words, we increment per-CPU counters by 2 for all non-prefetch references, and by 1 for prefetches. Sec. 6 shows that this generally improves performance most consistently.

ⓑ Switching between application reference streams: BLISS switches between streams of references from different CPUs to balance good performance and fairness. TEMPO modifies the process of switching slightly. After a page table access, its subsequent prefetch is also scheduled before references from a different application are scheduled. This ensures that prefetches are handled in a timely manner. Furthermore, we have found that it is best to wait for a grace period after performing the prefetch before references from a competing application are scheduled. Keeping the prefetched row buffer contents open for longer allows incoming references from replayed instructions to enjoy row buffer hits. We find that 15-cycle grace periods provide good performance.

4.4 Interactions with Sub-row Buffers

Past work has proposed replacing per-bank row buffers with multiple smaller sub-row buffers [18]. Sub-row buffers are useful in many ways. For example, they improve fairness and performance with *Fairness Oriented Allocation (FOA)* and *Performance Oriented Allocation (POA)* [18]. FOA allocates dedicated row buffers to cores that suffer the most interference. POA takes into account the differing memory bandwidth requirements of CPUs and allocates row buffers to cores in line with their demands. TEMPO can be applied to these approaches with no changes. Also, we find that dedicating 2 sub-rows for an architecture with 8 sub-rows improves performance (proposed in prior work [18]) as it permits multiple simultaneous post-translation prefetches.

Cores	LLC	Memory controller	DRAM
32 core, 4GHz, 3-wide issue, 8 MSHRs/core, 128-entry inst. win., 32KB L1$, 64-entry L1 TLB, 1024-entry L2 TLB	64B cache lines, 32-way set associative, 32MB	64-entry read/ write queue FR-FCFS, BLISS, adaptive polices	4TB, DDR3-1600, 800MHz, 1 rank/ channel, 8 banks/rank, 64K rows/bank, 8KB row buffers, tRCD/tRAS 12/30 cycles

Figure 9. Simulated machine. Parameters on row buffer policies, sub-row buffers, and cache prefetching are detailed in the text.

4.5 System-Level Interactions

Superpages: TEMPO is applicable to any page size (or distribution of page sizes) that the OS allocates. For example, for 2MB superpages, the page table walker simply tags the L2 PT access – now the leaf PT – to trigger the memory controller into initiating row buffer and LLC prefetches. Sec. 6 shows how well TEMPO and superpages operate in tandem.

Page faults: Occasionally, DRAM page table accesses may refer to a translation with an unallocated virtual page. TEMPO's Prefetch Engine identifies these unallocated translations and ensures that they do not trigger prefetches.

5. Methodology

5.1 Evaluation Workloads

TEMPO is primarily useful for big-memory workloads on big-memory servers. Therefore, like past work [39, 40, 44, 55], we use several workloads from the machine learning, graph analytics, and sparse linear algebra domains. These consist of memory-intensive applications mcf from Spec, and canneal from Parsec; locality-sensitive hashing from nearest neighbor workloads in machine learning (lsh); sparse vector matrix multiplication (spmv); and symmetric gauss-siedel smoother (sgms), an important high-performance computing application which performs a forward and back triangular solve. We also use graph500, xsbench, a Monte Carlo neutron transport [55], and the illustris cosmoglical framework [56], a popular state-of-art simulation of the physics of the universe. All use 3-4TB of memory.

We have also evaluated TEMPO on all the remaining Spec and Parsec workloads. We expect TEMPO to be modestly useful for workloads with smaller memory footprint; nevertheless, we evaluate them to ensure that TEMPO does not *harm* performance or energy if DRAM page table accesses are rare.

5.2 Simulation Infrastructure

Our simulation approach is a two-step process. First, to capture accurate real-system virtual memory behavior, we use a modified version of Pin (logging both virtual and physical memory addresses) to generate memory traces of program behavior [34, 35]. Our test system has an Intel Skylake processor with 4TBs of memory, running stock Linux.

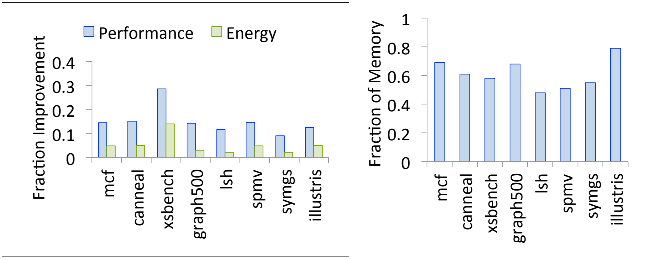

Figure 10. (Left) Improvements in performance (blue) and energy (green) using TEMPO as a fraction of baseline execution; and (right) fraction of memory footprint devoted to 2MB superpages.

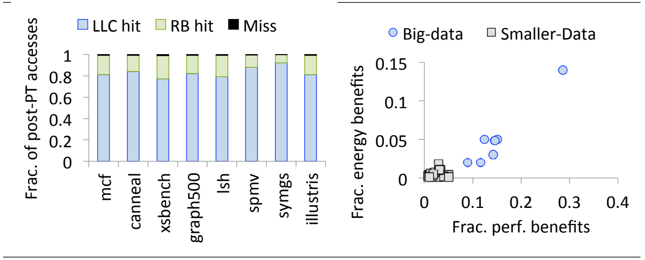

Figure 11. (Left) Fraction of replays serviced from the LLC and row buffer; and (right) energy/performance graphs comparing big-data workloads with Spec/Parsec workloads with small memory footprints.

We then feed these traces to an in-house simulation framework modeling Intel's Skylake-style processors with the parameters shown in Figure 9. Our detailed out-of-order core models are allied with a DRAM timing simulator inspired by prior work [57, 58]. Note that we assume a system with a large 4TB physical memory. Further, our memory controller supports different scheduling policies including closed-row, open-row, and adaptive policies. Like prior work [17], our adaptive open-row policy uses a cache structure to predict the length of time that a row should be left open. We use 2048-set, 4-way prediction caches [17]. Further, we study the impact of 8 sub-row buffers – since the large row buffer in Figure 9 is 8KB, each sub-row buffer is 1KB – with the default FOA and POA designs from prior work [18]. Finally, we implement IMP prefetchers, using the default configuration from prior work [44]. We assume a 16-entry prefetch table, 4 entry indirect pattern detector, with a 2 maximum indirect ways and levels, and 16 as the maximum prefetch distance [44].

We implement TEMPO in Verilog and synthesize, place, and route using Synopsis 32-nm generic libraries. We assess area, energy, and timing implications.

6. Evaluation

We begin by assuming a design with a single 8KB row buffer, and FR-FCFS scheduling with an adaptive-row policy from prior work [18]. Subsequent sections detail TEMPO's performance with different schedulers and row buffer organizations.

6.1 Performance and Energy Improvements

The graph on the left of Figure 10 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. A 0 fraction benefit implies no change in execution time over the baseline. In tandem, the graph on the right shows the fraction of memory footprint devoted to 2MB superpages, measured on the real system from which we collect traces.

Performance benefits: Figure 10 shows that TEMPO consistently boosts all our big-data workloads. Workloads with frequent DRAM page table accesses (e.g., see xsbench) enjoy close to 30% performance boosts. In general, the poorer the access locality of the workload, the more useful TEMPO is. Note that TEMPO is beneficial because DRAM page table accesses remain frequent despite the fact that most workloads back more than 50% of their memory footprint with superpages (the graph on the right of Figure 10).

Energy benefits: Figure 10 shows that TEMPO saves energy, despite its area overheads, by speeding up execution and hence reducing static energy. We see 1-14% energy savings, trending similarly with performance improvements.

Breakdown of benefits: The graph on the left of Figure 11 distinguishes the benefits of row buffer and LLC prefetching. For each workload, we separate the fraction of replays that hit 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 page table 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% 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, by design, TEMPO initates prefetches only when workloads use memory sufficiently aggressively to initiate many DRAM page table accesses. We also, however, need to ensure that TEMPO's overheads do no harm workloads with smaller memory footprints (i.e., that the additional hardware doesn't significantly compromise system energy, etc.). The graph on the right of Figure 11 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. Naturally, the big-data workloads benefit most from TEMPO. However, not a sin-

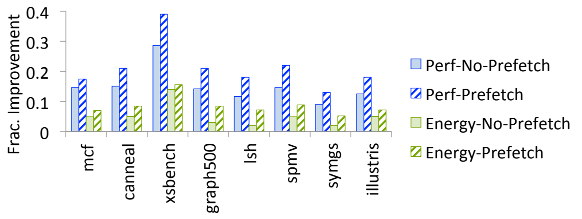

Figure 12. Comparison of TEMPO's performance and energy benefits with and without IMP prefetchers.

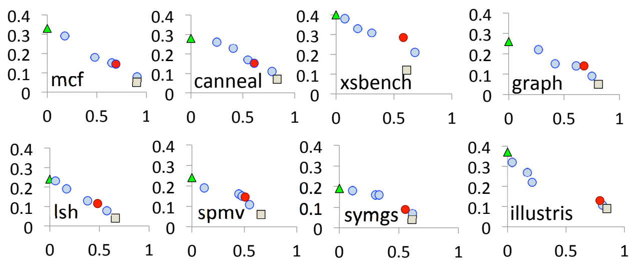

Figure 13. The effect of superpages on TEMPO performance. The x-axis shows the fraction of the memory footprint covered by 4KB pages only (triangles), 4KB and 2MB pages (circles), or 4KB and 1GB pages (boxes). The y-axis shows the fraction of improved performance versus the baseline without TEMPO, where 0 represents no change in runtime, and higher numbers mean runtime improvements. The red circle represents the configuration we have been using.

gle smaller-footprint workload becomes becomes slower or consumes more energy. Instead, performance improves by 1-2% and energy by roughly 1%.

Interactions with cache prefetching: Figure 12 quantifies TEMPO's benefits in the presence of recently-proposed IMP prefetchers, which prefetch into the L1 cache. The blue bars represent performance, and the green represent energy savings. Striped bars represent TEMPO in the presence of prefetching. In every case, TEMPO is even more useful in the presence of prefetching. Workloads with especially poor memory access locality (e.g., xsbench and smpv) are particularly aided, seeing almost a 10% performance improvement from the no-prefetching case. Energy savings track these performance benefits too; since application runtime is further reduced, static energy is mitigated.

6.2 Interactions with Superpages

Thus far, our results have focused on a system where our target applications run on Linux with transparent hugepage support. Therefore, as the right graph of Figure 10 shows, the OS generates 2MB superpages when possible. In our experiments, the OS can usually back more than 50% of the memory footprint with superpages. We also, however, assess the benefits of TEMPO as a function of page size distribution. Naturally, the more the DRAM page table accesses, the more the benefits from TEMPO. Superpages, which cut down TLB misses and the page table size [36, 59], reduce the incidence of DRAM page table accesses.

We assess TEMPO's relationship with superpages. We first turn off transparent hugepage support, allowing only 4KB pages. We expect TEMPO's benefits to be highest for this configuration. Then, we turn on transparent hugepages to create 2MB superpages [42], but modulate its effectiveness by introducing memory fragmentation via the use of a memory-thrashing application, memhog. Like prior work [2, 34, 36], we configure memhog to randomly allocate 0%, 25%, 50%, and 75% of system memory. We expect that as memhog's footpring increases, physical memory becomes increasingly fragmented, making it harder to create 2MB superpages. In addition, we consider alternatives to transparent hugepage support; specifically, Linux also supports 2MB pages using libhugetlbfs. Unlike transparent hugepages, libhugetlbfs requires the developer to link his/her application to the library; however, it is also likelier to back application footprint with superpages, as they are demanded explicitly [38]. We therefore use libhugetlbfs to generate 2MB pages. Finally, we also consider x86-64 1GB superpages (note that Linux currently supports 1GB pages only with libhugetlbfs). For each study, we collect memory traces as per Sec. 5.

Figure 13 shows the results of our experiments. For each workload, we plot TEMPO's performance improvements on the y-axis as a function of the fraction of total memory footprint backed by superpages (the x-axis). The green triangle corresponds to the case where only 4KB pages are allowed; hence, 0% of the memory footprint (on the x-axis) is covered with superpages. The circles correspond to cases where either transparent hugepage support or libhugetlbfs generates 2MB superpages, with differing memhog configurations. The red circle, in particular, corresponds to the results assumed for the rest of the paper (i.e., transparent hugepage support with memhog at 0%). Finally, libhugetlbfs for 1GB pages is shown with boxes with the x-axis indicating the fraction of memory covered by 1GB superpages.

As expected, the more frequent the superpages (higher x-axis values), the less the performance impact of TEMPO. Nevertheless, even when 2MB superpages are prevalent (e.g., libhugetlbfs with 2MB pages), we still consistenly enjoy performance benefits of 8-25%. In fact, reasonable fragmentation levels that might exist in cloud settings [36] (i.e., transparent hugepage support with memhog of 25-50%) see performance benefits ranging from 10-30%. Furthermore, even when using 1GB pages – which one might imagine would mostly eliminate DRAM page table accesses – TEMPO provides 5%+ performance benefits in many cases. This is because our big-data workloads with 3-4TB memory footprints still require several 1GB pages to cover the entire memory footprint. Furthermore, TEMPO is effective in pernicious cases when address translation is a huge problem

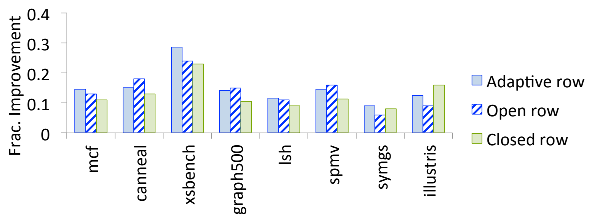

Figure 14. Performance improvements for TEMPO assuming adaptive, open, and closed-row-policies.

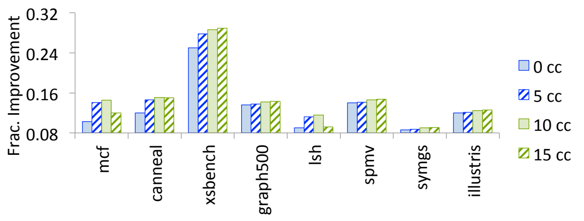

Figure 15. Performance improvements for TEMPO as the number of cycles we wait anticipating future page table accesses varies.

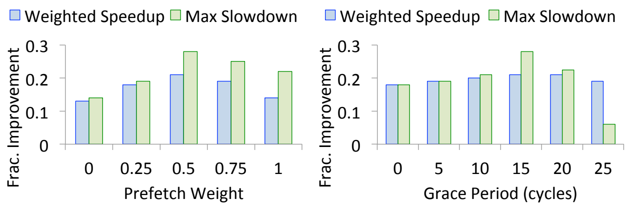

Figure 16. (Left) Fractional improvements versus the baseline, using TEMPO, in weighted speedup and maximum slowdown of any application as a function of the post-page table prefetch weight versus other memory reference; and (right) as a function of the grace period after the replay's data is prefetched into the row buffer.

(i.e., when superpages are scarce), with benefits consistently exceeding 25%.

6.3 Memory Scheduler Interactions

Row buffer policies: Figure 14 quantifies TEMPO's performance benefits for different row management policies. Results are normalized to a baseline with the particular policy; i.e., the adaptive row results show TEMPO's performance benefits versus a baseline with adaptive row management.

Figure 14 show that TEMPO consistently improves all row management strategies. The exact benefits vary depending on the relationship between the policy and workload. For example, consider canneal, where TEMPO is actually most useful (18% benefits) using open-row policies. This is because canneal's multiple threads occassionally share spatially-adjacent data; an open-row policy improves row buffer hit rates (more than adaptive policies) for multiple threads simultaneously. As another example, xsbench is best aided by adaptive row management, followed by open, and then closed-row management; even the least-performant case of the closed-row policy is boosted by 25%.

Consider also applications like illustris, which have such poor locality of memory accesses that adaptive and open-row policies are outperformed by closed-row policies. TEMPO generates more row buffer hits, boosting performance.

Anticipating page table accesses: Sec. 4.3 detailed the benefits of adopting a 10-cycle wait time before closing rows with page table contents. We now present experimental data to show this design point, varying the wait time from 5 to 15 cycles. Figure 15 shows our results. Note that in order to showcase the performance differences, we zoom in on the y-axis, changing its scale from prior graphs.

Figure 15 shows that waiting even 5 additional cycles before closing an open row with page table contents boosts performance by 1-3%. This is because the row buffer is 8KB and can hence hold information about spatially-adjacent translations (the exact number depends on how the OS maps and interleaves data across DRAM banks and channels) in the page table. Whenever the LLC suffers a miss for some of these adjacent translations, the row buffer improves performance. Waiting 10 cycles further boosts performance, but beyond this (15 cycles), performance may be compromised as prefetches are delayed.

Memory schedulers for fairness: TEMPO also aids fairness-based memory schedulers like BLISS [23, 24]. Like previous work, we create 80 multiprogrammed workloads with 32 applications each. Our workloads are constructed using the Spec and Parsec applications with a range of memory intensities. Furthermore, like past work [23, 24], we measure weighted speedup for performance, and the maximum slowdown experienced by any application, for fairness.

Figure 16 shows the results of our experiments. We quantify the fractional improvement in weighted speedup and maximum slowdown. The higher the improvement, the better, with 0 indicating no change versus the baseline without TEMPO. Note also that an improvement in the maximum slowdown plots the fraction with which the slowest application has been improved. The graph on the left shows these metrics (averaged over all multiprogrammed worloads to save space) as a function of how prefetches are weighed versus other DRAM accesses by the BLISS counters. Not only is the weighted speedup consistely higher for *every* single case, the slowest application is also made faster by 10%+ in all configurations. Further, we find that treating prefetches as half the weight of other memory references provides the best gains.

The graph on the right of Figure 16 shows the same metrics but as a function of the grace period discussed in Sec.

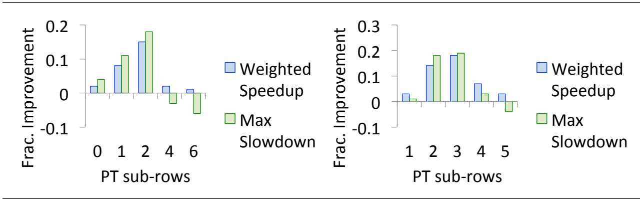

Figure 17. (Left) Fractional improvements versus the baseline for FOA, in terms of weighted speedup and maximum slowdown of any application as a function of the number sub-row buffers dedicated for post-page table prefetches; and (right) the same for POA

4.3. Recall that we need a grace period to keep a row's prefetched contents open for a few cycles before switching to another application's memory stream, to ensure that prefetches are not wasted. Waiting for too short a time wastes prefetch effort, while waiting for too long slows down the application next serviced by the memory controller. We find while the weighted speedup remains largely unchanged, the grace period affects the slowest application's runtime. This makes sense – applications that see relatively less service rates at the memory controller are particularly sensitive prefetching. Wasting prefetching resources because the grace period is too small curtails the potential benefits. So does waiting too long for the slow application's references to be serviced. Figure 16 shows that while *every* configuration does benefit from TEMPO, a grace period of 15 cycles is the best choice.

6.4 Sub-Row Buffers

We show that TEMPO aids sub-row buffers too. We replace the per-bank 8KB row buffers with 8 separate 1KB sub-row buffers, and use FOA and POA from prior work [18]

Figure 17 quantifies the performance benefits, as a fraction of the baseline FOA (left) and POA (right) schemes, that TEMPO provides. We show the results as a function of the number of sub-rows that we dedicate to post-page table prefetches. TEMPO's ability to improve performance is highly dependent upon the number of sub-row buffers allocated to prefetches. In general, dedicating 2 out of the 8 rows ensures that prefetch data (but also other spatially-adjacent accesses) enjoys row buffer hits; however, dedicating too many sub-rows degrades performance by de-prioritizing other memory accesses. Overall, dedicating 2 sub-rows provides roughly 15% and 20% boosts in weighted speedups and the performance of the slowest application in our multiprogrammed workloads.

7. Conclusion

This work introduces TEMPO, a low-overhead, hardware-only augmentation of the page table walker and memory controller to remove DRAM accesses from replayed instructions off the critical path of execution. TEMPO prefetches the 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. Overall, we believe that TEMPO is readily-implementable in upcoming systems.

8. Acknowledgments

We thank Guilherme Cox, Ján Veselý, Zi Yan, and Ronil Mehta for preliminary discussions on the technical content of this work. We also thank Mikko Lipasti, Carole-Jean Wu, and Tushar Krishna for their feedback on drafts of this work.

References

[1] O. Mutlu and L. Subramaniam, "Research Problems and Opportunities in Memory Systems," *SUPERFRI*, 2015.

[2] G. Cox and A. Bhattacharjee, "Efficient Address Translation with Multiple Page Sizes," *ASPLOS*, 2017.

[3] R. Cooksey, S. Jourdan, and D. Grunwald, "A Stateless, Content-Directed Data Prefetching Mechanism," *ASPLOS*, 2002.

[4] O. Mutlu, "Memory Scaling: A Systems Architecture Perspective," *MEMCON*, 2015.

[5] B. Jacob, "The Memory System: You Can't Avoid It; You Can't Ignore It; You Can't Fake It," *Morgan Claypool Synthesis Lectures Series*, 2009.

[6] K. Chang, P. Nair, S. Ghose, D. Lee, M. Qureshi, and O. Mutlu, "Low-Cost Inter-Linked Subarrays (LISA): Enabling Fast Inter-Subarray Data Movement in DRAM," *HPCA*, 2016.

[7] V. Seshadri, T. Mullins, A. Boroumand, O. Mutlu, P. Gibbons, M. Kozuch, and T. Mowry, "Gather-Scatter DRAM: In-DRAM Address Translation to Improve the Spatial Locality of Non-unit Strided Accesses," *MICRO*, 2015.

[8] K. K.-W. Chang, D. Lee, Z. Chishti, A. Alameldeen, C. Wilkerson, Y. Kim, and O. Mutlu, "Improving DRAM Performance by Parallelizing Refreshes with Accesses," *HPCA*, 2014.

[9] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, M. Kozuch, P. Gibbons, and T. Mowry, "RowClone: Fast and Energy-Efficient In-DRAM Bulk Data Copy and Initialization," *MICRO*, 2013.

[10] D. Lee, Y. Kim, V. Seshadri, J. Liu, L. Subramaniam, and O. Mutlu, "Tiered-Latency DRAM: A Low Latency and Low Cost DRAM Architecture," *HPCA*, 2013.

[11] S.-L. Lu, Ying-Chen, and C.-L. Yang, "Improving DRAM Latency with Dynamic Asymmetric Subarray," *MICRO*, 2015.

[12] Y. H. Son, O. Seongil, Y. Ro, J. Lee, and J. H. Ahn, "Reducing Memory Access Latency with Asymmetric DRAM Bank Organizations," *ISCA*, 2013.

[13] A. Udipi, N. Muralimanohar, N. Chatterjee, R. Balasubramonian, A. Davis, and N. Jouppi, "Rethinking DRAM Design and Organization for Energy-Constrained Multi-Cores," *ISCA*, 2010.

[14] Y. Kim, V. Seshadri, D. Lee, J. lee, and O. Mutlu, “A Case for Exploiting Subarray-Level Parallelism (SALP) in DRAM,” *ISCA*, 2012.

[15] H. Hassan, G. Pekhimenko, N. Vijaykumar, V. Seshadri, D. Lee, O. Ergin, and O. Mutlu, “ChargeCache: Reducing DRAM Latency by Exploiting Row Access Locality,” *HPCA*, 2016.

[16] X. Shen, F. Shong, H. Meng, S. An, and Z. Zhang, “Rbpp: A Row Based DRAM Page Policy for the Manycore Era,” *ICPADS*, 2014.

[17] M. Awasthi, D. Nellans, R. Balasubramonian, and A. Davis, “Prediction based DRAM Row-Buffer Management in the Many-Core Era,” *PACT*, 2011.

[18] N. Dwarkanath, Gulur, M. Mehendale, R. Manikantan, and R. Govindarajan, “Multiple Sub-Row Buffers in DRAM: Unlocking Performance and Energy Improvement Opportunities,” *ICS*, 2012.

[19] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter, “Atlas: A scalable and high-performance scheduling algorithm for multiple memory constrollers,” *HPCA*, 2010.

[20] O. Mutlu and T. Moscibroda, “Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems,” *ISCA*, 2008.

[21] K. Nesbit, N. Aggarwal, J. Laudon, and J. Smith, “Fair Queueing Memory Systems,” *MICRO*, 2006.

[22] D. Abts, N. Enright-Jerger, J. Kim, D. Gibson, and M. Lipasti, “Achieving Predictable Performance Through Better Memory Controller Placement in Many-Core CMPs,” *ISCA*, 2009.

[23] L. Subramanian, D. Lee, V. Seshadri, H. Rastogi, and O. Mutlu, “BLISS: Balancing Performance, Fairness, and Complexity in Memory Access Scheduling,” *TPDS*, 2016.

[24] L. Subramanian, D. Lee, V. Seshadri, H. Rastogi, and O. Mutlu, “The Blacklisting Memory Scheduler: Achieving High Performance and Fairness at Low Cost,” *ICCD*, 2014.

[25] K. Sudan, N. Chatterjee, D. Nellans, M. Awasthi, R. Balasubramonian, and A. Davis, “Micro-Pages: Increasing DRAM Efficiency with Locality-Aware Data Placement,” *ASPLOS*, 2010.

[26] H. Huang, P. Pillai, and K. Shin, “Design and Implementation of Power-Aware Virtual Memory,” *USENIX ATC*, 2003.

[27] L. Peeled, S. Mannor, U. Weiser, and Y. Etsion, “Semantic Locality and Context-based Prefetching Using Reinforcement Learning,” *ISCA*, 2015.

[28] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. Pugsley, and Z. Chisti, “Efficiently Prefetching Complex Address Patterns,” *MICRO*, 2015.

[29] A. Fuchs, S. Mannor, U. Weiser, and Y. Etsion, “Loop-Aware Memory Prefetching Using Code Block Working Sets,” *MICRO*, 2014.

[30] T. Barr, A. Cox, and S. Rixner, “SpecTLB: A Mechanism for Speculative Address Translation,” *ISCA*, 2011.

[31] A. Bhattacharjee, D. Lustig, and M. Martonosi, “Shared Last-Level TLBs for Chip Multiprocessors,” *HPCA*, 2011.

[32] D. Lustig, A. Bhattacharjee, and M. Martonosi, “TLB Improvements for Chip Multiprocessors: Inter-Core Cooperative Prefetchers and Shared Last-Level TLBs,” *TACO*, 2012.

[33] A. Bhattacharjee and M. Martonosi, “Inter-Core Cooperative TLB Prefetchers for Chip Multiprocessors,” *ASPLOS*, 2010.

[34] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee, “CoLT: Coalesced Large-Reach TLBs,” *MICRO*, 2012.

[35] B. Pham, A. Bhattacharjee, Y. Eckert, and G. Loh, “Increasing TLB Reach by Exploiting Clustering in Page Translations,” *HPCA*, 2014.

[36] B. Pham, J. Vesely, G. Loh, and A. Bhattacharjee, “Large Pages and Lightweight Memory Management in Virtualized Systems: Can You Have it Both Ways?,” *MICRO*, 2015.

[37] V. Karakostas, J. Gandhi, A. Cristal, M. Hill, K. McKinley, M. Nemirovsky, M. Swift, and O. Unsal, “Energy-Efficient Address Translation,” *HPCA*, 2016.

[38] V. Karakostas, J. Gandhi, F. Ayar, A. Cristal, M. Hill, K. McKinley, M. Nemirovsky, M. Swift, and O. Unsal, “Redundant Memory Mappings for Fast Access to Large Memories,” *ISCA*, 2015.

[39] A. Basu, J. Gandhi, J. Chang, M. Hill, and M. Swift, “Efficient Virtual Memory for Big Memory Servers,” *ISCA*, 2013.

[40] J. Gandhi, A. Basu, M. Hill, and M. Swift, “Efficient Memory Virtualization,” *MICRO*, 2014.

[41] M. Papadopoulou, X. Tong, A. Seznec, and A. Moshovos, “Prediction-Based Superpage-Friendly TLB Designs,” *HPCA*, 2014.

[42] A. Arcangeli, “Transparent Hugepage Support,” *KVM Forum*, 2010.

[43] S. Rixner, W. Dally, U. Kapasi, P. Mattson, and J. Owens, “Memory Access Scheduling,” *ISCA*, 2000.

[44] X. Yu, C. Hughes, N. Satish, and S. Devadas, “IMP: Indirect Memory Prefetcher,” *MICRO*, 2015.

[45] T. Barr, A. Cox, and S. Rixner, “Translation Caching: Skip, Don’t Walk (the Page Table),” *ISCA*, 2010.

[46] A. Bhattacharjee, “Large-Reach Memory Management Unit Caches,” *MICRO*, 2013.

[47] A. Clements, F. Kaashoek, and N. Zeldovich, “RadixVM: Scalable Address Spaces for Multithreaded Applications,” *Eurosys*, 2013.

[48] A. Clements, F. Kaashoek, and N. Zeldovich, “Scalable Address Spaces Using RCU Balanced Trees,” *ASPLOS*, 2012.

[49] D. Lustig, G. Sethi, M. Martonosi, and A. Bhattacharjee, “COATCheck: Verifying Memory Ordering at the Hardware-OS Interface,” *ASPLOS*, 2016.

[50] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne, “Accelerating Two-Dimensional Page Walks for Virtualized Systems,” *ASPLOS*, 2008.

[51] A. Basu, M. Hill, and M. Swift, “Reducing Memory Reference Energy with Opportunistic Virtual Caching,” *ISCA*, 2012.

[52] Intel, “Haswell microarchitecture,” *www.7-cpu.com/cpu/Haswell.html*.

[53] Intel, “Skylake microarchitecture,” *www.7-cpu.com/cpu/Skylake.html*.

[54] H. Yoon, J. Meza, R. Ausavarungnirun, R. Harding, and O. Mutlu, “Row Buffer Locality Aware Caching Policies for Hybrid Memories,” *ICCD*, 2012.

[55] J. Vesely, A. Basu, M. Oskin, G. Loh, and A. Bhattacharjee, “Observations and Opportunities in Architecting Shared Virtual Memory for Heterogeneous Systems,” *ISPASS*, 2016.

[56] D. Nelson, A. Pillepich, S. Genel, M. Vogelsberger, V. Springel, P. Torrey, V. Rodriguez-Gomez, D. Sijacki, G. Snyder, B. Griffen, F. Marinacci, L. Blecha, L. Sales, D. Xu, and L. Hernquist, “The Illustris Simulation: Public Data Release,” *Arxiv*, 2015.

[57] Q. Deng, D. Meisner, L. Ramos, T. Wenisch, and R. Bianchini, “MemScale: Active Low-Power Modes for Main Memory,” *ASPLOS*, 2011.

[58] Q. Deng, D. Meisner, A. Bhattacharjee, T. Wenisch, and R. Bianchini, “CoScale: Coordinatd CPU and Memory System DVFS in Server Systems,” *MICRO*, 2012.

[59] J. Navarro, S. Iyer, P. Druschel, and A. Cox, “Practical, Transparent Operating System Support for Superpages,” *OSDI*, 2002.