Athena: An Early-Fetch Architecture To Reduce On-Chip Page Walk Latencies

Seyed Armin Vakil Ghahani*
arminvak@umich.edu
University of Michigan
USA

Soheil Khadirsharbiyani
szk921@psu.edu
Pennsylvania State University
USA

Jagadish B. Kotra
jagadish.kotra@amd.com
AMD Research
USA

Mahmut T. Kandemir
kandemir@psu.edu
Pennsylvania State University
USA

ABSTRACT

Large-scale applications from various domains are becoming increasingly irregular, posing significant strains on virtual memory performance. On the other hand, increasing hardware SRAM structures like TLB is becoming challenging due to technology scaling constraints imposed by the limitations of Moore's law. This emerging trend in applications, coupled with the lack of technology scaling in hardware, requires innovations at the hardware level to avoid expensive memory accesses for traversing page tables to keep page walk latencies in check.

In this work, we introduce and evaluate Athena, an early-fetch architecture that reduces the on-chip latency of page walk requests. More specifically, Athena reduces page walk latency by issuing early fetch without waiting on the Memory Management Unit to initiate the fetch. Athena improves performance by 6.5% in native non-virtualized environments, and by 15.6% in virtualized environments. Moreover, combining Athena with a recent complementary prior work, leads to further improvements of 16.5% and 23.4% in the native and virtualized environments, respectively.

CCS CONCEPTS

• **Computer systems organization** → **Architectures**; • **Hardware** → *Memory and dense storage*.

KEYWORDS

Translation Lookaside Buffer, Virtual Address, Address Translation, Cache Management

ACM Reference Format:
Seyed Armin Vakil Ghahani, Soheil Khadirsharbiyani, Jagadish B. Kotra, and Mahmut T. Kandemir. 2022. Athena: An Early-Fetch Architecture To Reduce On-Chip Page Walk Latencies. In *PACT '22: International Conference on Parallel Architectures and Compilation Techniques (PACT), October 10–12, 2022, Chicago, IL.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.XXXX/XXXXXX.XXXXXX

*This work was done while the author was at Pennsylvania State University.

PACT '22, October 10–12, 2022, Chicago,IL

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.XXXX/XXXXXX.XXXXXX

1 INTRODUCTION

Big Data workloads are increasingly becoming common covering applications ranging from from irregular graph analytics [21, 22] to genome sequencing [31]. Since modern processors fail to accommodate the working sets of such big workloads in on-chip structures, we witness significant increases in memory traffic. Unfortunately, conventional data storage and access strategies are not able to cope with this increased memory traffic and new approaches are needed.

Virtual memory is a crucial abstraction that ensures per-process isolation by virtualizing main memory. At the heart of virtual memory are per-process page tables that are stored in main memory. Per-process page tables are typically implemented as radix tree structures that contain virtual-to-physical address mappings. Modern x86 processors employ 5-level page tables in native non-virtualized environments [42], whereas the virtualized environments employ separate host and guest page tables. Consequently, a page table walk[1] can result in a maximum of 5 memory accesses in non-virtualized environments where as the nested page walk in virtualized environments can result in up to 35 memory accesses [35, 42].

Despite sustained efforts to improve the virtual memory performance via huge pages [24, 33, 40, 53], translation coalescing/prefetching and similar techniques [5, 9, 12, 23, 28, 34, 36, 39, 41, 44–46, 49–51, 53], applications with irregular/sparse data accesses like graph processing continue to pose significant virtual memory challenges and bottlenecks as they exhibit low temporal and spatial locality across accesses. The on-chip structures such as Translation Lookaside Buffer (TLB) that cache final address translations are ineffective in capturing the huge working sets of such applications [23, 45]. The non-deterministic access patterns in these applications make it hard for adapting optimizations like TLB prefetching. This non-deterministic access patterns result in expensive page walks to memory that fall on the critical path of data accesses. Virtual memory traffic accounts from 20% to 40% of the total memory traffic in large-scale irregular applications [7–9, 11–14, 18], translating to 20% to 50% of the overall application execution time. Note that

[1]We use the terms 'page walk' and 'page table walk' interchangeably in the rest of the paper.

emerging memory technologies like Intel Optane [25] will only exacerbate this problem due to their longer access latencies.

In this work, based on the observation that page walk latency can impose significant performance overhead for workloads with high TLB misses (e.g., irregular applications), we propose and evaluate an optimization in cache subsystem to reduce the page walk overhead and improve the overall system performance. More specifically, we introduce a Page Walk Aware Early-Fetching (*early-fetch*) [2] scheme, called **Athena**, that reduces the overall page walk latency by initiating the intermediate page walk accesses from the cache controllers, in contrast to relying on Memory Management Unit (MMU) to issue page walk accesses.

The main proposal in this work is based on the key insight that, when a page walk entry 'N' hits at a particular cache level 'X', it is highly unlikely that the levels of page walk entries beyond 'N' will hit in the cache levels below 'X' that are closer to processor. From our experiments, we observed that 99.8% of the page walk requests that hit in a cache level will have the subsequent page walk accesses hit at or beyond the same cache level. This is because the number of nodes in a page table radix tree increases from root to leaf levels, and consequently, as we traverse the page table from root towards leaf, the coverage reduces by $512x$. As an example, if a Page Upper Directory (PUD) covering a 1GB segment is hitting in last-level cache (LLC), since the Page Middle Directory (PMD) covers a 2MB page and Page Table Entry (PTE) covers a finer 4KB page, it is highly unlikely that PMD and PTEs are found in Page Walk Cache (PWC) or L2.

Our **main contributions** in this work include:

- We profile real-world applications to quantify the overheads imposed by virtual memory, and specifically page walks, in terms of both memory traffic and performance.
- We note that page walk latency plays a crucial role in the overall address translation latency experienced by the processor. These page walk latencies are long because the MMU requires looking up all the on-chip caches before the page walks are sent to memory. We further observe that the current MMU-initiated page walks increase the on-chip latency for page table walks.
- Based on the above observation, to reduce the on-chip page walk request latency, we propose Athena, a novel architecture that issues *early-fetches* to page walks in the cache hierarchy, *without* waiting on the MMU to initiate the fetch.
- We evaluate Athena in *virtualized environment* as well, and show that it can significantly reduce the overall nested page walk latency in these systems. This is due to the fact that each nested page walk is composed of multiple host page walks that are indeed optimized similar to a native page walk.
- Our evaluations on a variety of benchmark suites show that the proposed Athena architecture improves workload performance, on average, by 6.5% in the native non-virtualized environment. Additionally, we implement and evaluate a previously-proposed complementary optimization scheme, TEMPO [12], that only initiates the early-fetch of data from memory based on page walk hints, which can achieve 8.7% performance improvement, on average. We observe that early-fetching the page walk requests *complements* the early data fetch proposed in TEMPO nicely, and consequently, the Athena + TEMPO combination improves the performance of the native and virtualized environments, on average, by 16.5%, and 23.4%, respectively.

[2]Note that we propose an 'early-fetching' scheme that targets initiating early-fetch of page walk entries. This is *not* 'pre-fetching', as page walk prefetching requires predicting an access to a page for which it requires additional hardware structures to capture access patterns, etc. In contrast, our early-fetch does *not* require any additional hardware structures to track access patterns, as there is *no* prediction in early-fetch – unlike prefetching.

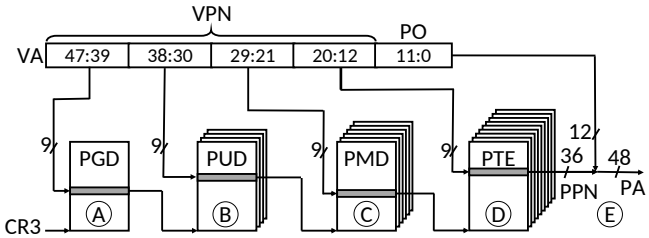


Figure 1: x86 radix page table organization.

2 BACKGROUND

2.1 x86 Address Translation

Modern systems offer virtual memory by providing a per-process page table which maintains translations from virtual addresses (VA) to physical addresses (PA). As the page table is a critical component of the virtual-to-physical address translation process, page table design and access latency have a significant impact on applications' execution time, particularly in the case of irregular workloads [7–9, 11–14, 18] with non-predictable access patterns.

Commercial systems implement page tables as *radix tree* structures with 4-levels. Figure 1 depicts the structure of a *four-level* page table, which is used in the x86-64 architecture. The page table levels are known as *PGD* (Page Global Directory), *PUD*, *PMD*, and *PTE*. These levels are accessed *sequentially* Memory-Management-Unit (MMU) as part of a *page walk* in which the output of each level is used to determines the base address of the next level.

To translate VA into PA, the virtual address is divided into virtual page number (VPN) and page offset (12 bits). The virtual address is 48-bit long, and its first 36 bits (VPN) are divided into four 9-bit sections. Each of these 9-bit sections is used as an index to access a level of the page table during the page walk. MMU uses the *CR3 register* as the base address of the PGD table and the first 9-bit section (bits 47-39) as an index for accessing this table Ⓐ. The value in this entry contains the base address of the PUD table, and the second 9-bit section (bits 38-30) is used as an index for accessing this level of page table Ⓑ. For the traditional 4KB pages, this process continues until MMU obtains the last-level entry (PTE) that contains the physical page number (PPN) corresponding to the virtual page that is under translation Ⓒ-Ⓓ . Lastly, this value is combined with the page offset, and PA is calculated Ⓔ. For systems with huge page support, page walk process may require fewer steps. For instance, in the case of 2MB pages, PA is calculated right after Ⓒ by combining page offset with PMD. For 1GB pages, even step Ⓒ can be skipped and PA is resolved after accessing Ⓑ.

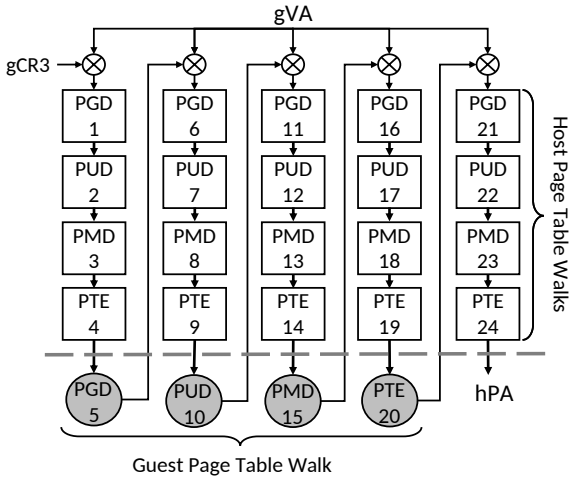


Figure 2: Nested page walk.

2.2 Averting Page Walk

As page walks can incur up to four additional memory accesses for each processor memory (load/store) instruction, processors cache the recently accessed final address translations ($VPN \rightarrow PPN$) in the per-core private Translation Lookaside Buffers (TLBs), which are organized as three separate structures: (1) Instruction TLB (iTLB), which stores the translations for instructions; (2) Data TLB (dTLB), which stores the translations of data accesses; and (3) shared TLB (sTLB), which contains both instruction and data address translations. Upon a memory access, first iTLB/dTLB is looked up with VPN for instruction/data accesses, and in case of a miss, sTLB will be looked up. In case of a TLB hit in either of these structures, the requested PA is calculated by combining the PPN that is stored in TLB and the corresponding page offset, and the address translation is completed. On the other hand, upon a TLB miss, a *page walk is performed in hardware* (as described in 2.1) and the instruction execution is delayed until the page walk is completed.

2.3 Accelerating Page Walk

To improve the performance of page-walks, and eventually address translation, x86-64 processors *cache* page table entries in designated caches in MMU, known as Page-Walk Cache (PWC) [4, 26]. PWC are implemented as partitioned caches where PGD, PUD, and PMD entries are stored in separate partitions. As PGD, PUD and PMD have different memory reach, PWC's hit rate vary across different levels. For example, each PGD, PUD, and PMD entry is associated with a 512GB, 1GB, and 2MB contiguous memory regions, respectively. Consequently, PGDs incur higher hit rates compared to PUDs, which in turn incur higher hits compared to PMDs. PWC does *not* cache PTEs as they generally have low locality. Upon a hit in PWC, the requested memory address is sent to the MMU, which subsequently issues the next memory request based on the state of the page walk. Upon a miss in PWC, the request is forwarded to L2, last-level cache (LLC), and then memory to complete the page walk.

2.4 Address Translation in Virtualized Systems

Address translation in virtual machines (VM) involves multiple page walks. As depicted in Figure 2, upon an address translation in the virtualized environment, a *guest virtual address* (gVA) should be translated to a *host physical address* (hPA). Translating a gVA to hPA consists of five different host page walks (each corresponding to a column in Figure 2, accesses 1-4, 6-9, 11-14, 16-19, 20-24) followed by a data memory access (5, 10, 15, 20). The host page walks traverse the host page table that effectively maps the guest's view of physical memory to the system's view of physical memory. This page table is maintained by the host OS, and the processor initiates each host page walk by starting from *hCR3*, which points to the first level of host page table.

The data memory access after the first four host page walks is part of the guest page table walk, whereas the data memory access after the last host page walk is the actual data memory request that will be issued by the core after the address translation and is not part of the nested page walk. The four memory accesses in the last row of Figure 2 are essentially the page walk memory requests that are performed from the perspective of the guest operating system.

Similar to the native page walks, the nested page walk memory accesses will also be cached in their designated PWCs. More specifically, the memory accesses in the first three rows of Figure 2 will be cached in the PGD, PUD, and PMD page walk caches, respectively. Accesses 5, 10, and 15 are part of the guest page table walk and will also be cached in the PGD, PUD, and PMD page walk caches, respectively. In addition to PWC, processors maintain another structure named *nested TLB* (NTLB) that caches the translation of one column in the nested page walk. This structure is beneficial in the first two host page walks of the nested page walk as the PGD and PUD entries of the guest page table cover 512GB and 1GB of the guest OS's memory. As a result, they have high temporal locality, and a small NTLB can cache their translation with a high hit efficiency.

Before discussing how our work reduces this page walk latency, we first delve into the details of how miss requests are handled in the cache hierarchy and the role of the Miss Status Handling Registers (MSHRs). Athena interacts with the cache MSHRs and hence it helps to understand the role of MSHRs in the baseline system.

2.5 Miss-Status Handling Register (MSHRs)

MSHRs enable *non-blocking caches* in memory hierarchy, increasing the memory-level parallelism. For every outstanding miss in caches, there is an MSHR entry that stores the address of the outstanding memory request. Upon receiving a response from upper level cache or memory, all MSHR entries are looked up, and then, (1) outstanding misses regarding the requested cache block are serviced and (2) a cache block is allocated for this response in cache. As the number of MSHR entries in a cache is limited, it is possible that all MSHR entries can be occupied at some point. In such a scenario, the cache is blocked and does not accept any memory requests from the lower levels of the cache hierarchy or the core, until at least one of the outstanding misses is resolved and the corresponding MSHR entry could be freed. Consequently, MSHRs need to be appropriately sized to leverage full-parallelism from the cache and memory hierarchy in processors.

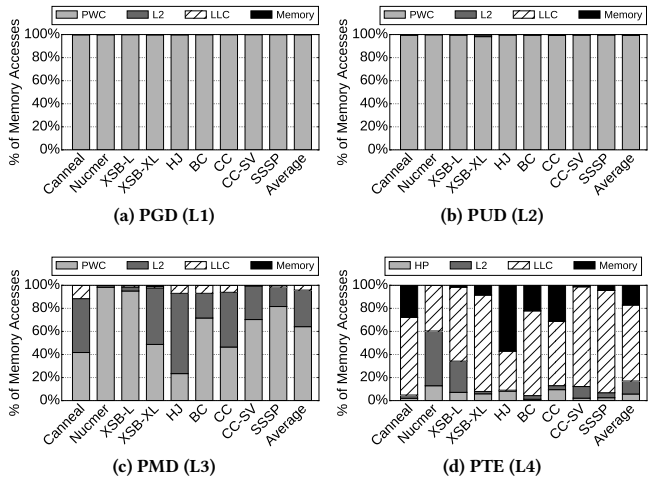


Figure 3: Distribution of the page walk hits across different levels of the memory hierarchy in native environments.

To reduce the number of the MSHR entries consumed, each MSHR entry can accommodate multiple outstanding memory requests (targets) to the same cache block from different sources (e.g., lower-level caches or prefetchers) in the cache hierarchy. As a result, it is *not* necessary to have different MSHR entries for the same cache block if there are multiple outstanding requests associated with that cache block and each MSHR can have multiple targets. An MSHR target refers to the requestor of the corresponding cache block that experiences a cache miss. The maximum number of targets can be the maximum number of requestors. The number of targets in an MSHR entry is also limited based on the design. Caches allocate multiple MSHR entries for the same cache block if the number of requests for that cache block exceeds the maximum number of targets in a single MSHR entry. Figure 7(a) shows an MSHR architecture with four MSHR entries and a maximum of two MSHR targets per entry. For example, physical address x in Figure 7a has three outstanding misses, $L2_1, L2_2, L2_3$ (cache block x is requested from private L2 of core 1, 2, and 3), and occupies two MSHR entries. In comparison, the physical cache block y has one outstanding prefetch (PF) request.

3 MOTIVATION

3.1 The problem

Although employing TLBs and PWCs improve the performance of the virtual-to-physical address translation process, page walk is still an expensive operation as on-chip TLBs fail to capture large working sets. While the majority of page walk memory requests are resolved in the cache hierarchy, some are forwarded to memory, and consequently, stall the processor for hundreds of cycles. To study the distribution of page walk memory requests that end up accessing memory, we conducted a simulation-based study using a variety of benchmarks in *both* native and virtualized systems, as described in detail in Section 5.

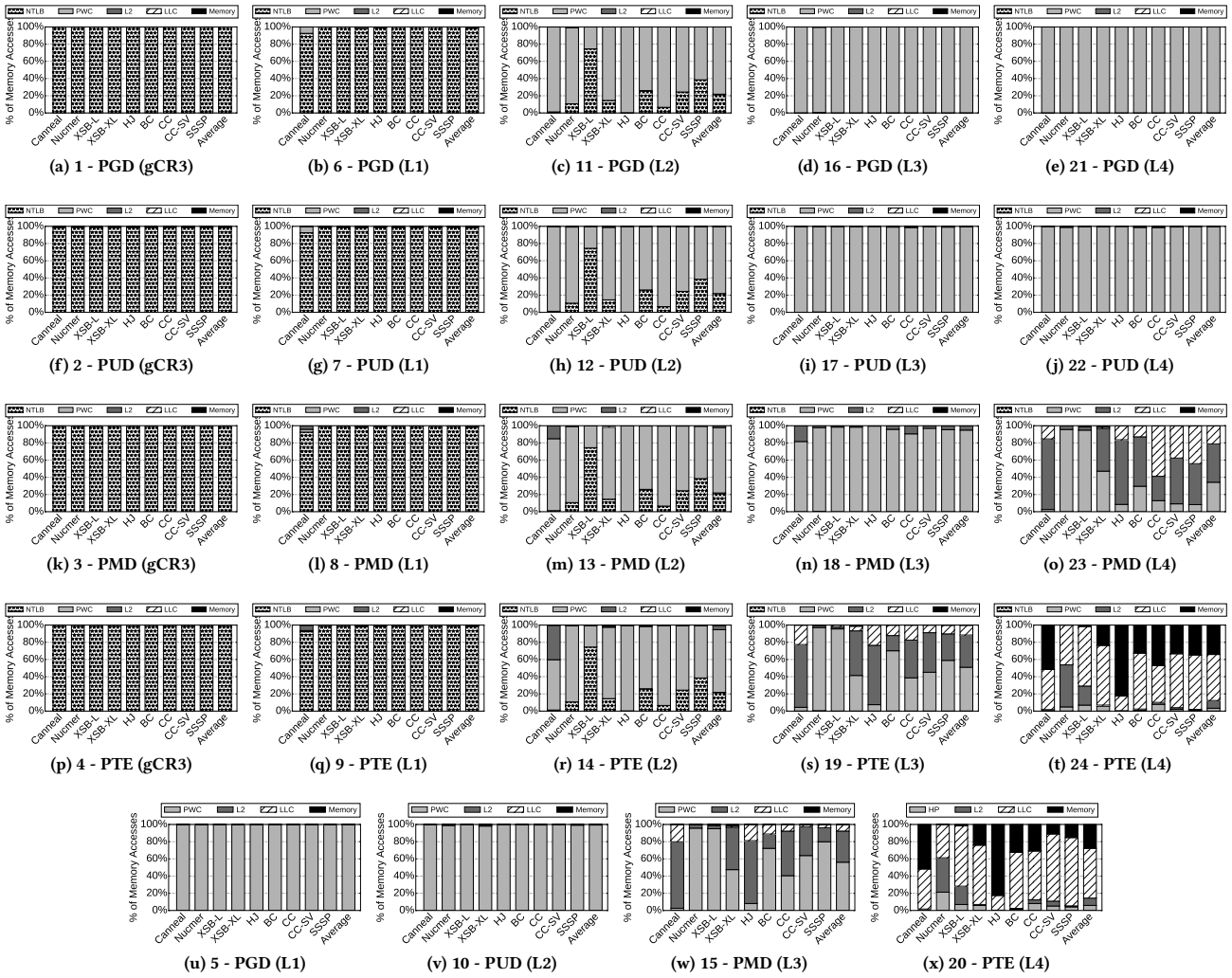
Figure 3 shows the distribution of the page walk hits across the different levels of the memory hierarchy – PWC, L2, LLC and memory, in native execution environment. For example, in the CC benchmark, 46.5%, 47.5%, 6.0% and 0.0% of the PMD requests hit in the PWC, L2, LLC and memory, respectively. As can be observed from Figures 3a, 3b, and 3c, the PWC hit rate decreases when moving from PGD to PMD, since each entry in one level corresponds to a 512x bigger region of memory compared to the next level. Nevertheless, as most of the PGD, PUD and PMD requests do not need to be sent to the memory, the latency associated with them do not contribute significantly to the overall page walk latency.

On the other hand, a large fraction of the PTE requests end up being serviced from memory, which can take hundreds of clock cycles. In fact, Figure 3d indicates that 18.1% of the PTE requests have to access memory, on average. This number can be as high as 58.3% in the HashJoin benchmark. Therefore, reducing the overhead of accessing the PTE level requests can potentially lead to significant improvements in performance.

The emerging big data workloads can exacerbate this problem even further. For example, scientific workloads and graph analytics frameworks can have up to tens of gigabytes of memory footprints [7–9, 11–14, 18]. This increase in the application memory footprints, coupled with the limited number of TLB entries, results in high TLB miss rates, and eventually a large number of page walks [14]. The increase in memory footprint also causes the address translation latency to grow since the PWC and other caches in the system cannot capture all address translations. Therefore, the number of requests end up going to memory increases, which in turn affects the overall performance of the system [11, 45].

The virtualized environments that can incur a maximum of 24 page table walk requests [42] for a single load/store memory instruction further emphasize the importance of virtual memory traffic in datacenter environments. To evaluate how each of these 24 memory requests is serviced in the cache hierarchy, we conduct a similar study to Figure 3. Figure 4 shows the distribution of the level of the cache hierarchy that each memory access hits. Note that accesses #1-4, #6-9, #11-14, #16-19, and #21-24 are for the translation of a gVA to gPA, and these memory accesses are for five invocations of a host page table walk. On the other hand, accesses #5, #10, #15, and #20 are within the guest page table and, from the perspective of the guest operating system, only these four memory accesses are made, which is essentially similar to a page walk in the native execution environment. Here are the key takeaways from this figure:

- The first two host page walks (#1-4 and #6-9) mostly hit in the *nested TLB* (NTLB), and each of them requires only one memory access to the guest page table (accesses #5 and #10). This shows the effectiveness of NTLB to cache the translations of gVA to gPA for the first two host page walks. On the other hand, NTLB is not able to cache most of the next host page table walks, and MMU needs to perform the host page table walk for each of the three gVA-to-gPA address translations.
- The PGD and PUD accesses of the last three host page table walks (accesses #11-12, #16-17, #21-22), which experience a miss in NTLB, are mostly hit in PWC and have low access latency, as well as the guest page table PGD and PUD access (accesses #5, #10).



(a) 1 - PGD (gCR3) (b) 6 - PGD (L1) (c) 11 - PGD (L2) (d) 16 - PGD (L3) (e) 21 - PGD (L4)

(f) 2 - PUD (gCR3) (g) 7 - PUD (L1) (h) 12 - PUD (L2) (i) 17 - PUD (L3) (j) 22 - PUD (L4)

(k) 3 - PMD (gCR3) (l) 8 - PMD (L1) (m) 13 - PMD (L2) (n) 18 - PMD (L3) (o) 23 - PMD (L4)

(p) 4 - PTE (gCR3) (q) 9 - PTE (L1) (r) 14 - PTE (L2) (s) 19 - PTE (L3) (t) 24 - PTE (L4)

(u) 5 - PGD (L1) (v) 10 - PUD (L2) (w) 15 - PMD (L3) (x) 20 - PTE (L4)

Figure 4: Distribution of the page walk hits across different levels of the memory hierarchy in virtualized systems.

– Among the remaining memory accesses, the only accesses that reach LLC and have higher latency are accesses #15, #19-20, and #23-24. Moreover, only memory accesses #20 and #24 access the main memory and experience long latency in some of the benchmarks.

3.2 Key Insight

Our goal in this work is to reduce the latency of page walks in native and virtualized systems with lightweight hardware changes. The first key insight upon which we build our proposal is that the page walk requests are sequentially issued by MMU, as shown in Figure 5a, and the coverage of each level of page table gets lower, as we go from PGD to PTE, by magnitude of 512 after each access. Consequently, if a page walk request hits in one level of the cache hierarchy, it is very likely that the next page walk request will hit in a cache in the same level or the level above. Consequently, we can start issuing **early-fetch** of the next page walk request as soon as we get the response for this level of page walk from this particular cache level.

Further, in the context of virtualized systems, the 24 memory accesses for a single address translation can be divided into five host page table walks (#1-4, #6-9, #11-14, #16-19, #21-24) and a single guest page table walk (#5, #10, #15, #20) which is interleaved between these host page table walks. If we look more closely, each column in Figure 4 is actually a page walk and a subsequent data access that accesses the guest page table. As a result, our first key insight still holds true for these individual page walks and their subsequent memory access. In other words, we can employ the *early-fetch* mechanism discussed above to these five page walks and their subsequent memory access, thereby improving the performance of each host page walk and nested page walk, in overall.

In this work, we propose ***Athena***, a scheme to early fetch the PTE memory request as soon as the content of the PMD request is available in the cache hierarchy. Consequently, instead of waiting

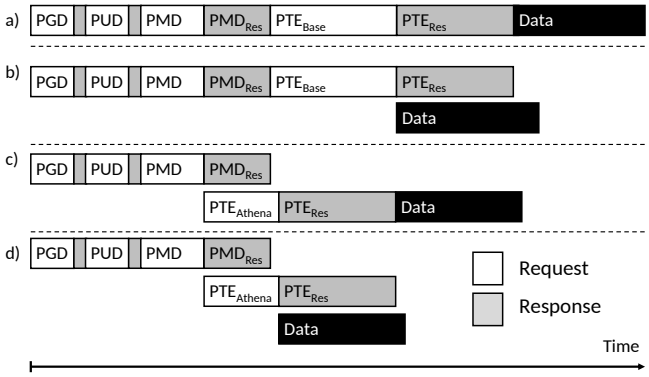


Figure 5: Timeline of a page walk and data access in the native setup for (a) Baseline, (b) TEMPO, (c) Athena, and (d) Athena + TEMPO.

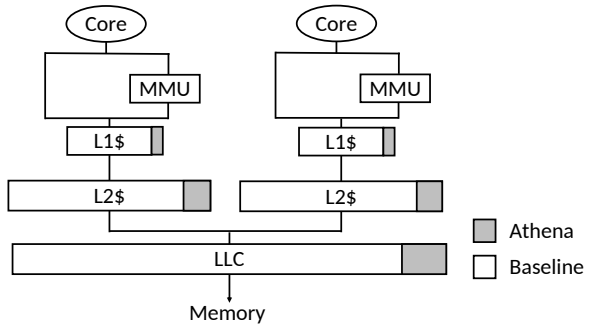


Figure 6: Modifications required for Athena.

for the MMU to issue the PTE request corresponding to the next level of page walk, Athena detects such accesses and issues an ***early-fetch request***. As illustrated in Figure 5(c), Athena issues the next page walk request starting from the level of the cache hierarchy where the previous access has hit, and consequently, saves (1) the response time for the previous request to be received in MMU (PMD_{Res}) and (2) the lookup time of smaller caches for the next request ($PTE_{Athena} < PTE_{Base}$). Hence, Athena *overlaps* the response of the previous level request in the page walk to core with the issue of the next level request, thereby reducing the overall on-chip latency of the page walk request.

Additionally, Athena complements prior work TEMPO architecture [12]. TEMPO issues the early-fetch requests only for data based on the hints from page walk. Unlike TEMPO that operates at the memory-controller level, Athena + TEMPO provides benefits for *all* page walk accesses that hit in L2 or LLC or memory. This is particularly significant as we observe in Figure 3d that, even though a high percentage of PTE memory accesses require accessing the main memory, 15.7% and 55.5% of the PTE requests are serviced by L2 and LLC, respectively, on average. As a result, TEMPO cannot prefetch the data for these memory accesses, and it can only capture the performance opportunity of the black portion of Figure 3. Figure 5(d) shows how Athena reduces the overall memory latency of a processor load instruction *in tandem* with TEMPO. This combination, as will be demonstrated later in the paper, leads to further improvements in performance.

Furthermore, Athena can improve the performance of virtualized systems by early-fetching the five separate page walks and their subsequent memory access without any further changes. Athena optimizes each of these page walks by early-fetching the last two accesses of each page walk (the last two rows in Figure 4). Note that the early-fetching of page walk accesses within a nested page walk is enabled by the unique design of Athena at the *cache-controller level*, and that the prior work (TEMPO) *cannot* improve the performance of intermediate page walk accesses as it only reduces the latency of the data memory access *after* these 24 page walk memory accesses. One can argue that TEMPO could early-fetch the intermediate guest memory access at the end of each individual host page walk (accesses #4, #9, #14, #19). It is to be noted however that, since TEMPO operates at the memory-controller level, it can only early-fetch the memory requests that reach main memory and miss in the cache hierarchy (which happens only for access 24, based on Figure 4). As a result, it would not provide benefits for the intermediate memory accesses within a nested page walk.

4 OUR PROPOSAL

In this section, we explain our design objectives as well as the implementation details of our proposal, and discuss how our approach reduces the page walk latency.

4.1 Design Objectives

We propose Athena based on two main design objectives:

Exposing Page Walks to Cache Hierarchy: We aim to extend the memory hierarchy in such a way that it is *aware* of page walks. We address the shortcomings of the current systems in which all memory requests associated with a page walk are issued by MMU *sequentially*, as MMU can only issue the next memory access when it receives the prior response. We overcome this drawback by issuing page walk memory requests *as soon as* the response from the previous level of the page walk is available in the memory hierarchy.

Lightweight Hardware Modifications: As caches and TLBs play significant roles in the performance of address translation in modern processors, it is crucial that the proposed changes to be both lightweight and easy-to-adapt. Unlike the prior works [2, 18, 45] that require extensive changes to page table organization, and subsequently TLBs, which make them difficult to adopt in practice, we propose a design that avoids these complications.

In the rest of this section, we describe the implementation details of our proposed optimizations.

4.2 Athena: Page Walk Early-Fetching

Figure 6 shows the overall modifications required for Athena. To calculate the next address that should be accessed after each level of the page walk, it is required to maintain a portion of virtual address that is under translation in the memory hierarchy. As our observation for the distribution of page walk hits in Figure 3 suggests, page walk requests that reach L2 or LLC are mostly PMD or PTE accesses. As a result, it is neither necessary nor beneficial to send the rest of bits of the virtual address to cache controllers

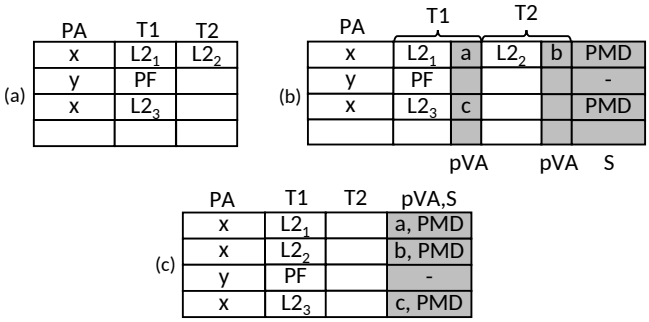


Figure 7: The MSHR architecture used in (a) baseline, (b) Athena, and (c) sensitivity analysis.

and maintain them in Athena. The offsets that are required for early-fetching after PMD and PTE access are bits 20-12 and 11-6, respectively. Consequently, we only send bits 20-6 from virtual address to cache controllers upon PMD and PTE accesses, and they will be stored in the pVA (partial virtual address) field of the MSHR targets, as shown in Figure 7b. Additionally, Athena stores the *level* of page walk that the memory request in MSHR corresponds to in the *S* field of MSHR entries, as illustrated in Figure 7b. Note that this value is only required per MSHR entry since each cache line can only be in one level of the page table.

Upon receiving the response of a page table request in MSHR, Athena first detects the level of the page walk request response using *S*. Then, it generates the physical address of the next level page table access in case of a PMD response, for each target. This address is generated by combining the first 9 bits of the pVA field with the value of PMD. Athena looks up the generated block addresses in this cache and simultaneously sends the early-fetch requests to the next level of the cache hierarchy, and discards this request in case of a cache hit. Note that this is done in parallel with sending the response of PMD to each requestor.

Similarly, upon receiving a PTE response in the cache, Athena can calculate the block address of the data cache line that experienced the corresponding TLB miss and this subsequent page walk, in order to *early-fetch* it to LLC. This is similar to TEMPO [12]; however, it does *not* require any operating system support to ensure that the page table and the requested data are on the same memory channel.

Athena in Action. Figure 8 illustrates how each request is serviced in MSHR using an example in LLC. Figure 8a shows the initial state of the MSHR, where there is an outstanding miss sent to the next level of the cache hierarchy for physical address X. This MSHR entry is waiting for the response, to forward it to the first core's L2 cache.

In the next step, a page walk request for address Y is received from the second core and a new MSHR entry is allocated for this outstanding miss, as shown in Figure 8b. Upon the receipt of the request, the cache controller creates an entry with the corresponding pVA (B), and the level of the page table request (PTE in this case) that Athena requires in addition to PA and the target location of the requested cache line.

Figure 8c illustrates the next step in which the response of the PMD request arrives and the cache controller initiates two processes *simultaneously* – (1) sending responses to the targets and (2) early-fetching the next level PA address (Z) by using the first 9 bits of the pVA in this MSHR target (A) and the value of PMD. The only difference for the new entry is that the MSHR keeps it as an "early-fetch request", which will *not* be forwarded to any core upon getting the response.

Finally, in the last step (Figure 8d), a request from MMU is received for the same address (Z), indicating that MMU sends the next level request after receiving the previous level's response. In this case, by comparing with the occupied entries in the MSHR, we find that there is an early-fetch request currently in the system for the same request. Therefore, the MSHR updates the target list while waiting for the result (MSHR hit). Upon receiving the response, the MSHR forwards the data to the target list (except for early-fetch), which overlaps the response latency of the current level with the request latency of the next level, thereby improving the performance of the program. Additionally, as this example shows, our approach does *not* introduce any additional requests into the LLC and memory; as a result, it does *not* have any negative impact on system performance.

Multiple Page Walks in a Single MSHR Entry. Each MSHR entry stores multiple requests in a cache line, and consequently, if two page table walkers are translating virtual addresses in a same page, the corresponding MSHR entry in the cache needs to store and process multiple page walks. This requires to store the pVA and the state of the page walk (S) for each target of MSHR, as shown in Figure 7b.

The hardware overhead of maintaining the state of the page walk for each target could be high if the number of outstanding page walk requests in a cache at each point of time is low. In other words, we might unnecessarily pay the price of supporting multiple page walks in one MSHR entry while having only a few page walk memory requests at each point in time, which is actually the case in LLC. To reduce this overhead, Athena allocates *multiple MSHR entries* if multiple page walks are accessing the same cache line in LLC, as shown in Figure 7c. With this design choice, we decrease the overhead and complexity of Athena significantly, without affecting the performance, as shown in Section 6.3. For more details on the hardware overheads brought by Athena, refer to Section 6.5.

Prefetch vs Early-Fetch. Data prefetchers are widely used in commercial processors and they prefetch, *speculatively*, cache lines that might be accessed by the processor in the future. Even though the state-of-the-art prefetchers can improve performance significantly, they pollute the on-chip network and memory bandwidth as they aggressively prefetch cache lines into LLC. In contrast, Athena early-fetches cache lines *non-speculatively*, i.e., it early-fetches memory addresses that are going to be accessed in the near future (i.e., it is 100% accurate).

5 METHODOLOGY

We model a 8-core processor similar to an Intel Skylake Server [52] with 128GB main memory, as described in detail in Table 1. We use gem5 (v20) [29] in Full System (FS) mode running Linux Kernel v4.15 [48] with **Transparent Hugepage Support** [17] enabled.

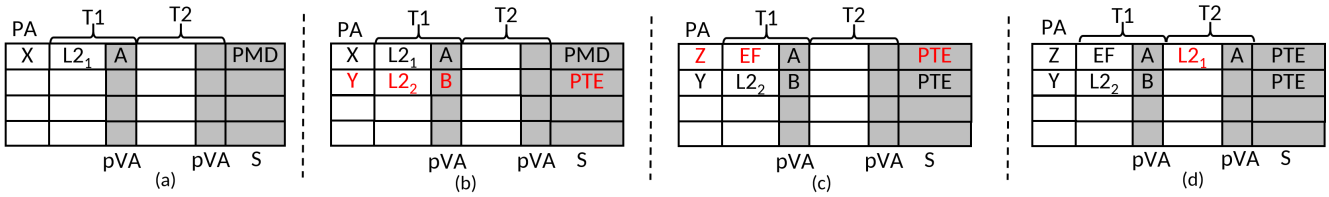


Figure 8: An example illustrating how Athena operates in LLC: (a) An outstanding PMD request in MSHR, (b) An outstanding PTE request arrives at LLC, (c) The response for physical address X arrives and the PTE request Z is early-fetched, (d) The PTE request Z is received from MMU after it got the response, and the next request is issued and experienced an MSHR hit. As a result, the PTE request will not be sent to memory again, and the early-fetch request target can be removed from the MSHR.

Table 1: Important architectural parameters.

Processor Parameters	
Processor	8 cores, OoO 192-entry ROB, 2GHz
Per-core TLB/Caches Parameters	
I-TLB/D-TLB	64/32 4KB/2MB entries, 4-way, 1 cycle
Unified S-TLB	1536 4KB/2MB entries, 12-way, 6 cycle
PT Walkers	2 walkers, 16-entry NTLB, 2 cycle RT
PWC	2/4/32 entries for PGD/PUD/PMD fully-associative, 2 cycle RT
L1-I/L1-D	32KB, 64B, 8-way, 4 cycle RT
Unified L2	512KB, 64B, 8-way, 16 cycle RT
LLC (shared)	16MB, 64B, 16-way, 60 cycle RT [52]
Memory Parameters	
Memory Controllers	One per channel, 64-entry read queue, 128-entry write queue
Main-Memory	128GB, DDR4-2400 [32], 4 channels, 2 ranks/channel, 16 banks/rank 128K rows/bank, 8KB row

Table 2: Description of our workloads.

Benchmark	PTW-PKI	Memory Footprint
Scientific Applications		
Canneal[15]	19.4	32GB
Nucmer[3]	23.8	2GB
XSBench-L[47]	15.1	5.6GB
XSBench-XL[47]	16.4	114 GB
Micro-benchmark		
HashJoin[1]	113.1	13GB
Graph Applications[10]		
BC	78.2	74GB
CC	1.8	74GB
CC-SV	70.3	74GB
SSSP	74.5	87GB

Gem5 does not support virtualized systems and it is not possible to run a virtual machine inside its OS with KVM enabled. In order to evaluate virtualized systems with enabled KVM, we assume gem5 acts as a host OS that is essentially running a guest operating system, and upon a page walk after a TLB miss, instead of walking a single-level page table (guest page table) that is handled by the guest OS, we perform a nested page walk. We maintain a host page table in gem5 that maps guest physical page #X to host physical page #X. The host page table is located at the end of physical main memory, and it is not accessible by the Linux inside gem5 as it is the case in real-world virtualized systems. Nevertheless, memory accesses to this page table will be interleaved with the memory accesses from the guest OS and they both share cache hierarchy and memory controllers. The guest page table is managed by an *unmodified* Linux OS. This is contrary to prior work [35] that emulates both guest and host page table in gem5.

5.1 Experimental Setup

We implemented Athena in gem5-v20 by extending its MMU unit to support two-level TLBs, separate PWCs for each of the first three levels of the page table, and a nested-TLB for virtualized systems, to have a realistic evaluation. The important simulation parameters are given in Table 1. Our simulations are fast-forwarded using the X86KvmCPU model in gem5 to reach the region-of-interest in each application using Sandberg et al's methodology [43], and then actual simulations are modeled using gem5's OoO CPU model. The CPU micro-architecture and caches are warmed up for 100 million instructions, and following that, we simulate at least 1 billion instructions for each core to report the results. We use two page-table walkers to not block the core upon a TLB miss, and also employ a 16-entry NTLB in the virtualized mode. We calculate the latency of the TLBs and caches using CACTI [6].

We evaluated each benchmark in our experimental suite using different combinations of the following configurations:

(i) **Baseline:** MMU page walk without any modification.
(ii) **TEMPO [12]:** MMU page walk by initiating a data early-fetch after the page walk in memory controller.
(iii) **Athena:** MMU page walk with extended support for PTE early-fetching in caches during the page walk.
(iv) **Athena + TEMPO:** Athena architecture with the support of early-fetching data requests at the cache-controller level.

All our evaluations are performed on the CloudLab [20] machines.

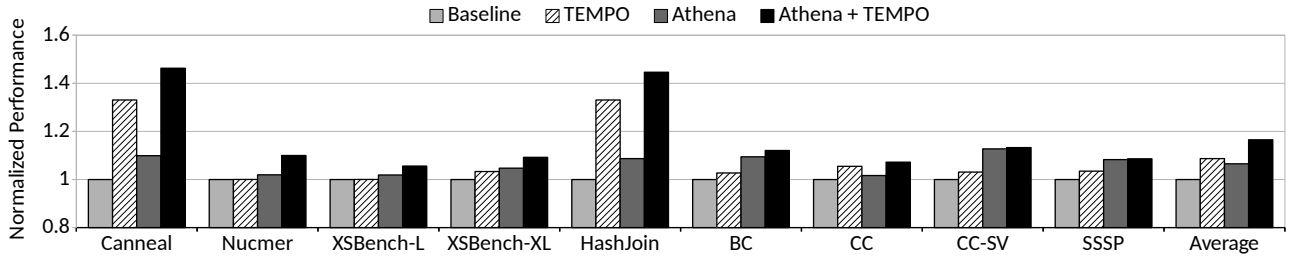


Figure 9: Normalized performance results.

5.2 Benchmarks

We use a variety of applications with different ranges of PTW-PKIs to evaluate our design under different scenarios. All our evaluated benchmarks are multi-threaded OpenMP [19] applications (see Table 2). As can be observed, these benchmarks are diverse and include scientific applications such as Canneal [15], Nucmer [3], and XSBench [47]. We use PARSEC netlist generator [38] to generate a netlist with 100M elements as the input for the Canneal benchmark. We evaluate the Nucmer benchmark using hs_chr17 [27] input dataset. We use both the large and XL input sizes for evaluating XSBench. We also evaluate the HashJoin [1] micro-benchmark, which look-up elements in a hash-table with 1B elements distributed in 100M rows of a hash-table. We also assess our design on four graph processing applications from GAPBS [10], including Betweenness Centrality (BC), Connected Components (CC), Shiloach and Vishkin algorithm for CC (CC-SV), and Single-Source Shortest Path (SSSP). We run Twitter [16] dataset in all our graph processing applications.

6 EVALUATION

We analyze the performance of our proposals by conducting different studies. Figure 9 shows the overall performance speedup brought by Athena, TEMPO, and combination of them, with respect to the *baseline* described in Section 5.1. We also evaluate the average page walk latency improvement of Athena in comparison to the baseline, and the results are plotted in Figure 10. The key takeaways from these studies can be summarized as follows.

Athena Performance Results: It can be observed that Athena achieves a geo-mean speedup of 6.5% across all benchmarks, in comparison to the baseline. Athena reduces the execution time as it *overlaps* the response access of PMD with the request access of PTE in the next level, which improves the page walk latency. Note also that we achieve this speed up, despite we only exploit for early for page walk requests and not for data (unlike TEMPO).

Athena provides significant improvements for Canneal [15] and HashJoin [1]. This is because the page walk latency presents a significant performance overhead in these benchmarks and PWC cannot hide the latency of the page walk. Figures 3, 10 and the PTW-PKI field in Table 2 confirm the same. As a result, the processor is stalled for hundreds of cycles waiting for page walks to be serviced in these benchmarks. Athena can reduce this overhead and improve IPC by 9.9% and 4.7% for Canneal and HashJoin, respectively. In contrast, for benchmarks such as Nucmer [3] and CC [10], the page walk overheads do not seem to be significant, meaning that the PWC is effective in reducing the page walk requests to the cache

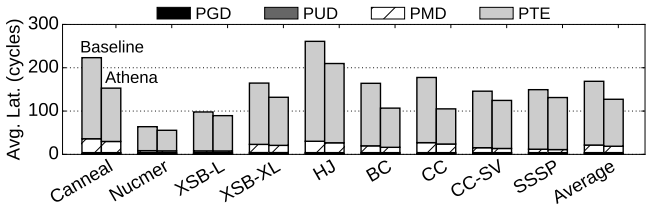


Figure 10: Page walk request latency for each level of page table in Athena.

hierarchy. Consequently, our proposal enhances the performance by only 1.9% and 1.6% in Nucmer and CC, respectively.

Further, we want to emphasize that Athena does *not* bring any performance overheads. This is because, during a workload's execution, Athena does *not* create any additional memory accesses and improves performance only by early-fetching (not pre-fetching) addresses from memory before the MMU issues the same request. Additionally, when MMU issues the early-fetched request, there will be an MSHR hit and, consequently, this request will *not* be sent to the next level of the cache hierarchy. Therefore, Athena does not hurt performance and only optimizes the latency by overlapping the requests. This observation is applicable to TEMPO as well, since it early-fetches requests for *data* when the corresponding *physical address* is available at the end of the page walk.

Athena with TEMPO: Athena and TEMPO individually achieve, 6.5% and 8.7% speedups, respectively, over the baseline. Since they are orthogonal, by combining these two approaches, one can further improve the performance. This is possible because *Athena focuses on early-fetching the page table cache lines, whereas TEMPO improves the performance by early-fetching of the data as soon as the physical address is available.* Therefore, by combining these approaches (Athena + TEMPO), we achieve an average speedup of 16.5% over the baseline. Athena improves the performance of graph applications by 8%, on average, and its performance improvements can be up to 12.7% in the CC-SV graph application.

6.1 Virtualized System Analysis

Figure 11 plots the performance improvements brought by Athena in virtualized systems. As shown in the figure, Athena's effectiveness in the virtualized system is *higher* compared to its effectiveness in the native execution environment. This is to be expected, as an address translation in the virtualized environment can result in up to 24 memory accesses in the worst case. Consequently, Athena has

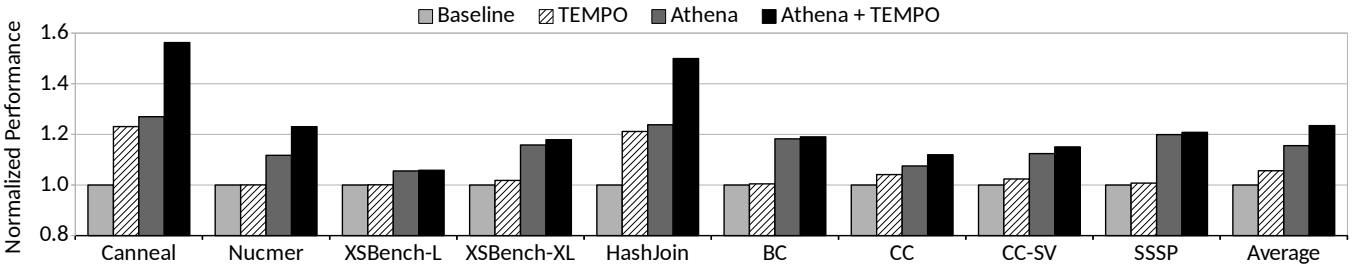


Figure 11: Performance improvements brought by Athena in the virtualized environment.

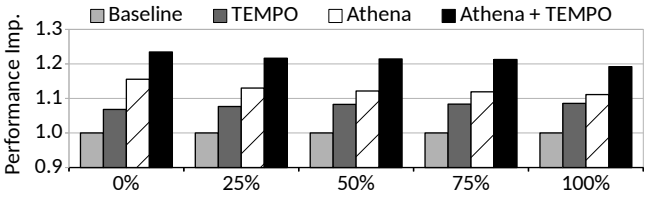


Figure 12: Sensitivity analysis on the percentage of huge pages in the host page table in a virtualized environment.

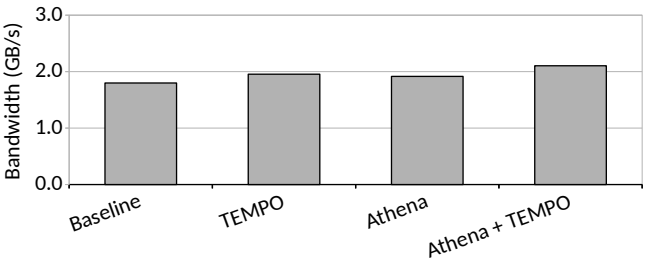


Figure 13: Average memory bandwidth consumption.

more opportunities to overlap the response delays with the next requests. For example, the performance improvement brought by the proposed optimizations in Canneal reaches 56.2% in the virtualized environment, while the corresponding improvement in the native execution for this benchmark was 46.2%. On average, the combination of Athena and TEMPO improves the performance of the evaluated benchmarks in the virtualized environment by 23.4%.

We also studied the impact of *huge pages* in the virtualized environment. In this experiment, we promote baseline pages to huge pages with uniform distribution. We ran this experiment with five different percentages of huge pages: 0%, 25%, 50%, 75%, and 100%, while THP is enabled for the guest operating system. Figure 12 plots the average performance improvements brought by Athena and TEMPO in the virtualized environment with different number of huge pages in the host page table. These results reveal that, while the relative performance improvements decrease when increasing the number of huge pages, Athena still leads to significant performance improvements in the virtualized environment, primarily because huge pages can only *partially* reduce the latency of the nested page walks.

6.2 Memory Bandwidth Results

Figure 13 compares the average memory bandwidth consumptions of the baseline, Athena, TEMPO, as well as their combination. We note that the memory bandwidth consumption increases by up to 20.1% when employing the Athena + TEMPO configuration. This increase in the bandwidth consumption is mainly because our approach sends a number of memory requests over a shorter period of time. Athena may introduce extra memory requests when a translation is squashed in MMU because of the speculative execution but the number of such cases is negligible based on our experiments.

6.3 Sensitivity Analysis on MSHRs with Multiple Page Walks

The MSHR entries can have multiple targets and it is possible that multiple page walk requests, which are accessing the same cache line, are under translation in the LLC. In such a scenario, Athena allocates *multiple* MSHR entries for this cache line, in an attempt to reduce the required hardware metadata, to enable the early-fetch of the page walk requests, as discussed earlier in Section 4.2. The results plotted in Figure 14 indicate that this design choice does not affect the performance of Athena in any benchmark (except in HashJoin, by 0.5%), due to the low likelihood of occurrence.

6.4 Sensitivity to Round-Trip Latency

Athena targets reducing the latency of a page walk in *both* the request and response paths of the page walk memory request. To study the impact of LLC latency on the performance improvements achieved by Athena, we also performed a sensitivity analysis by varying the round-trip latency. Figure 15 shows the performance improvements brought by Athena with different round-trip latencies from LLC. It can be observed that the performance improvements are higher for longer round-trip latency values. This is expected as longer round-trip latency imposes higher performance overheads, and as a result, Athena can achieve higher performance benefits by reducing both the request and response latencies of the page walk memory requests. Note that Athena consistently achieves performance improvements for address translation in all cases.

6.5 Hardware Overheads

Athena stores the PTE offset (bits 20-12) of the VA under translation and the state of the page walk request in caches to enable early-fetching of the page table requests. For each request, Athena also

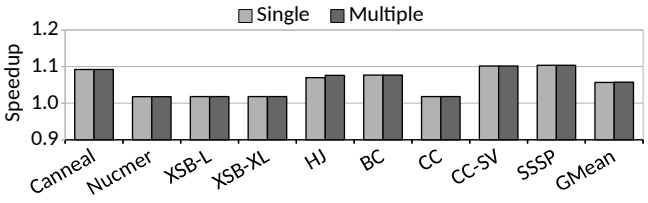


Figure 14: Sensitivity analysis on supporting single/multiple page walk requests in each MSHR entry.

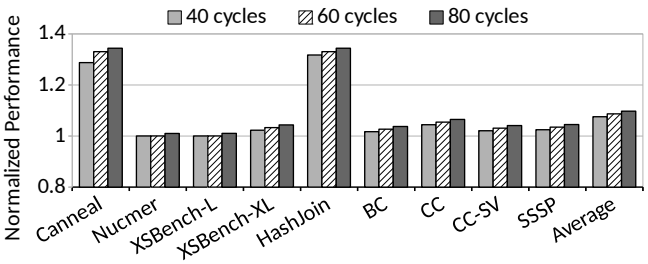


Figure 15: Sensitivity analysis on the latency of LLC.

stores 6 most-significant bits of the page-offset of the VA as it early-fetches the data request when working along with TEMPO. Note that, the remaining 6 least-significant bits of the page offset are not required as we early-fetch a cache line. Athena also maintains a 2-bit metadata to capture the state of the page walk request. As a result, the total overhead brought by Athena is $9+6+2 = 17$ bits for each MSHR entry in the LLC. This overhead will be 2 bit per MSHR entry and 15 bits for each MSHR target in the L2 cache. The number of MSHR entries varies at different levels of the cache hierarchy and across architectures. We evaluated our design with 16 and 128 MSHR entries in L2 and LLC, respectively, and 12 targets per MSHR entry. This leads to 364 bytes per L2 and 272 bytes in LLC, as L2 maintains per MSHR target metadata and LLC per MSHR entry. Moreover, since the impact of Athena in virtualized environments comes primarily from reducing the latency of each host page walk, which is essentially a native page walk in the perspective of caches, Athena's design does not incur any additional hardware overhead in virtualized environments.

7 RELATED WORK

7.1 Page Walk Optimizations

Bhattacharjee [12] proposes TEMPO, an approach that early-fetches data requests that experienced a page walk in memory controller, as shown in Figure 5b. TEMPO requires *OS support* to ensure the data that is going to be accessed after page walk and the corresponding last-level PTE entry are mapped to the *same* memory controller. This requires exposing the memory address-interleaving information to the OS so that OS can ensure a PTE and the corresponding data pages belong to the same memory controller. As discussed in prior sections, this work is orthogonal to our work and, as presented in Section 6, our work complements this proposal, and a combination of Athena and TEMPO generates further performance improvements over the individual schemes.

Margaritov et al. [30] propose reducing the latency of page walks by allocating the levels of the radix page table in contiguous physical memory and storing the entries in a sorted order to establish a direct mapping between a virtual page number and the physical address of its corresponding PTE.

This approach requires *intrusive changes* to system software (OS) to allocate Page Tables in contiguous physical memory. This is a *significant challenge* in long-running datacenter systems [33] that suffer from fragmentation. Other modules in the system software (OS) like Kernel Same Page Merging (KSM), compaction, etc., require marking the already-allocated Page Table pages as immovable to satisfy the contiguity constraints imposed in [30]. Additionally, reserving a contiguous physical memory region ahead of time can lead to under-utilization of memory. Contrary to this approach, Athena is *hardware-only* and does *not* require any intrusive changes in system software (OS) to reduce expensive page walk latency.

Park et al. [37] propose flattening the page table in order to reduce the number of page walk accesses given a contiguous memory, and relaxes the necessity of contiguous memory as it falls back to conventional page walk in case of not finding a large region of memory. Athena can be combined with this approach and early-fetch the next page table request similar to conventional page walks.

7.2 Page Table Structure Optimizations

Elastic Cuckoo Hashing [45] proposed a new hash table architecture for replacing radix page tables in order to convert sequential page walk accesses into parallel look-ups while resolving hash collisions. These proposals improve the page walk latency either by changing the structure of the page tables completely or through significant modifications to page tables' current structure (radix page table). In contrast, our proposal improves the performance through minor modifications to the MSHR organization, which can be easily implemented as part of current architectures.

8 CONCLUDING REMARKS

In this paper, we present and evaluate Athena, a novel optimization for minimizing the page walk latency, to improve the overall system performance. Athena architecture improves the page walk latency by issuing early-fetch of page walk requests. Our evaluation results using a set of scientific and graph analytics benchmark programs and a cycle-accurate simulation environment show that Athena improves the performance, on average, by 6.5% and 15.6% for the native and virtualized environments, respectively. Further, when combined with a recently-proposed complementary work [12], these savings jump to 16.5% and 23.4%, respectively.

REFERENCES

[1] Reto Achermann and Ashish Panwar. 2020. HashJoin Workload. https://github.com/mitosis-project/mitosis-workload-hashjoin

[2] Reto Achermann, Ashish Panwar, Abhishek Bhattacharjee, Timothy Roscoe, and Jayneel Gandhi. 2020. Mitosis: Transparently Self-Replicating Page-Tables for Large-Memory Machines. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 283–300. https://doi.org/10.1145/3373376.3378468

[3] K. Albayraktaroglu, A. Jaleel, Xue Wu, M. Franklin, B. Jacob, Chau-Wen Tseng, and D. Yeung. 2005. BioBench: A Benchmark Suite of Bioinformatics Applications. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, 2005 (ISPASS '05)*. IEEE Computer Society, USA, 2–9. https://doi.org/10.1109/ISPASS.2005.1430554

[4] AMD Inc. 2018. AMD64 Architecture Programmer's Manual,Volume 2. http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/24593_APM_v21.pdf.

[5] Mohammad Bakhshalipour, Aydin Faraji, Seyed Armin Vakil Ghahani, Farid Samandi, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. 2019. Reducing Write-backs Through In-Cache Displacement. *ACM Trans. Des. Autom. Electron. Syst.* 24, 2, Article 16 (jan 2019), 21 pages.

[6] Rajeev Balasubramonian, Andrew B. Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories. *ACM Trans. Archit. Code Optim.* 14, 2, Article 14 (June 2017), 25 pages.

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

[8] Thomas W. Barr, Alan L. Cox, and Scott Rixner. 2011. SpecTLB: A Mechanism for Speculative Address Translation. In *Proceedings of the 38th Annual International Symposium on Computer Architecture* (San Jose, California, USA) *(ISCA '11)*. Association for Computing Machinery, New York, NY, USA, 307–318.

[9] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. 2013. Efficient Virtual Memory for Big Memory Servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (Tel-Aviv, Israel) *(ISCA '13)*. Association for Computing Machinery, New York, NY, USA, 237–248. https://doi.org/10.1145/2485922.2485943

[10] Scott Beamer, Krste Asanović, and David Patterson. 2017. The GAP Benchmark Suite. arXiv:1508.03619 [cs.DC]

[11] Abhishek Bhattacharjee. 2013. Large-Reach Memory Management Unit Caches. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture* (Davis, California) *(MICRO-46)*. Association for Computing Machinery, New York, NY, USA, 383–394. https://doi.org/10.1145/2540708.2540741

[12] Abhishek Bhattacharjee. 2017. Translation-Triggered Prefetching. *SIGPLAN Not.* 52, 4 (April 2017), 63–76. https://doi.org/10.1145/3093336.3037705

[13] A. Bhattacharjee, D. Lustig, and M. Martonosi. 2011. Shared last-level TLBs for chip multiprocessors. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*. IEEE, San Antonio, TX, USA, 62–63. https://doi.org/10.1109/HPCA.2011.5749717

[14] Abhishek Bhattacharjee and Margaret Martonosi. 2010. Inter-Core Cooperative TLB for Chip Multiprocessors. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Pittsburgh, Pennsylvania, USA) *(ASPLOS XV)*. Association for Computing Machinery, New York, NY, USA, 359–370. https://doi.org/10.1145/1736020.1736060

[15] Christian Bienia. 2011. *Benchmarking Modern Multiprocessors*. Ph. D. Dissertation. Princeton University.

[16] Meeyoung Cha, Hamed Haddadi, Fabricio Benevenuto, and Krishna Gummadi. 2010. Measuring User Influence in Twitter: The Million Follower Fallacy. *Proceedings of the International AAAI Conference on Web and Social Media* 4, 1 (May 2010), 10–17.

[17] Jonathan Corbet. 2011. Transparent Hugepage Support. https://lwn.net/Articles/423584/

[18] Guilherme Cox and Abhishek Bhattacharjee. 2017. Efficient Address Translation for Architectures with Multiple Page Sizes. *SIGARCH Comput. Archit. News* 45, 1 (April 2017), 435–448. https://doi.org/10.1145/3093337.3037704

[19] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Comput. Sci. Eng.* 5, 1 (Jan. 1998), 46–55. https://doi.org/10.1109/99.660313

[20] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 1–14.

[21] Assaf Eisenman, Ludmila Cherkasova, Guilherme Magalhaes, Qiong Cai, Paolo Faraboschi, and Sachin Katti. 2016. Parallel Graph Processing: Prejudice and State of the Art. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering* (Delft, The Netherlands) *(ICPE '16)*. Association for Computing Machinery, New York, NY, USA, 85–90. https://doi.org/10.1145/2851553.2851572

[22] A. Eisenman, L. Cherkasova, G. Magalhaes, Q. Cai, and S. Katti. 2016. Parallel Graph Processing on Modern Multi-core Servers: New Findings and Remaining Challenges. In *2016 IEEE 24th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE Computer Society, Los Alamitos, CA, USA, 49–58.

[23] J. Gandhi, V. Karakostas, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. S. Ünsal. 2016. Range Translations for Fast Virtual Memory. *IEEE Micro* 36, 3 (2016), 118–126. https://doi.org/10.1109/MM.2016.10

[24] Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin Funston, Alexandra Fedorova, and Vivien Quéma. 2014. Large Pages May Be Harmful on NUMA Systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (Philadelphia, PA) *(USENIX ATC'14)*. USENIX Association, USA, 231–242.

[25] Intel. 2022. Intel® Optane™ DC Persistent Memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html

[26] Intel Corporation. 2015. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3B. http://tinyurl.com/intel-x86-3b.

[27] John Johnston. 2015. Biobench input data sets. https://github.com/reiverjohn/biobench2/blob/master/MUMmer/input/inputData.tgz

[28] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman Ünsal. 2015. Redundant Memory Mappings for Fast Access to Large Memories. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (Portland, Oregon) *(ISCA '15)*. Association for Computing Machinery, New York, NY, USA, 66–78.

[29] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jeronimo Castrillon, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Carlos Escuin, Marjan Fariborz, Amin Farmahini-Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Anthony Gutierrez, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, Austin Harris, Timothy Hayes, Adrian Herrera, Matthew Horsnell, Syed Ali Raza Jafri, Radhika Jagtap, Hanhwi Jang, Reiley Jeyapaul, Timothy M. Jones, Matthias Jung, Subash Kannoth, Hamidreza Khaleghzadeh, Yuetsu Kodama, Tushar Krishna, Tommaso Marinelli, Christian Menard, Andrea Mondelli, Miquel Moreto, Tiago Mück, Omar Naji, Krishnendra Nathella, Hoa Nguyen, Nikos Nikoleris, Lena E. Olson, Marc Orr, Binh Pham, Pablo Prieto, Trivikram Reddy, Alec Roelke, Mahyar Samani, Andreas Sandberg, Javier Setoain, Boris Shingarov, Matthew D. Sinclair, Tuan Ta, Rahul Thakur, Giacomo Travaglini, Michael Upton, Nilay Vaish, Ilias Vougioukas, William Wang, Zhengrong Wang, Norbert Wehn, Christian Weis, David A. Wood, Hongil Yoon, and Éder F. Zulian. 2020. The gem5 Simulator: Version 20.0+. arXiv:2007.03152 [cs.AR]

[30] Artemiy Margaritov, Dmitrii Ustiugov, Edouard Bugnion, and Boris Grot. 2019. Prefetched Address Translation. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 1023–1036. https://doi.org/10.1145/3352460.3358294

[31] Aaron McKenna, Matthew Hanna, Eric Banks, Andrey Sivachenko, Kristian Cibulskis, Andrew Kernytsky, Kiran Garimella, David Altshuler, Stacey Gabriel, Mark Daly, and Mark DePristo. 2010. The Genome Analysis Toolkit: A MapReduce framework for analyzing next-generation DNA sequencing data. *Genome research* 20 (09 2010), 1297–303. https://doi.org/10.1101/gr.107524.110

[32] Micron. 2022. Micron DDR4 SDRAM Datasheet - MT40A2G4. https://www.micron.com/-/media/client/global/documents/products/data-sheet/dram/ddr4/8gb_ddr4_sdram.pdf

[33] Ashish Panwar, Aravinda Prasad, and K. Gopinath. 2018. Making Huge Pages Actually Useful. *SIGPLAN Not.* 53, 2 (March 2018), 679–692. https://doi.org/10.1145/3296957.3173203

[34] M. Papadopoulou, X. Tong, A. Seznec, and A. Moshovos. 2015. Prediction-based superpage-friendly TLB designs. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE Computer Society, Los Alamitos, CA, USA, 210–222.

[35] Chang Hyun Park, Sanghoon Cha, Bokyeong Kim, Youngjin Kwon, David Black-Schaffer, and Jaehyuk Huh. 2020. Perforated Page: Supporting Fragmented Memory Allocation for Large Pages. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA '20)*. IEEE Press, Valencia, Spain, 913–925.

[36] Chang Hyun Park, Taekyung Heo, Jungi Jeong, and Jaehyuk Huh. 2017. Hybrid TLB Coalescing: Improving TLB Translation Coverage under Diverse Fragmented Memory Allocations. *SIGARCH Comput. Archit. News* 45, 2 (jun 2017), 444–456.

[37] Chang Hyun Park, Ilias Vougioukas, Andreas Sandberg, and David Black-Schaffer. 2022. *Every Walk's a Hit: Making Page Walks Single-Access Cache Hits.* Association for Computing Machinery, New York, NY, USA, 128–141. https://doi.org/10.1145/3503222.3507718

[38] PARSEC. 2012. Canneal Netlist Generator. https://parsec.cs.princeton.edu/download/other/canneal_netlist.pl

[39] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. 2012. CoLT: Coalesced Large-Reach TLBs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture* (Vancouver, B.C., CANADA) *(MICRO-45)*. IEEE Computer Society, USA, 258–269.

[40] Binh Pham, Ján Veselý, Gabriel H. Loh, and Abhishek Bhattacharjee. 2015. Large Pages and Lightweight Memory Management in Virtualized Environments: Can You Have It Both Ways?. In *Proceedings of the 48th International Symposium on Microarchitecture* (Waikiki, Hawaii) *(MICRO-48)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/2830772.2830773

[41] Akshay Krishna Ramanathan, Sara Mahdizadeh Shahri, Yi Xiao, and Vijaykrishnan Narayanan. 2022. Achieving Crash Consistency by Employing Persistent L1 Cache. In *Proceedings of the 2022 Conference & Exhibition on Design, Automation & Test in Europe* (Antwerp, Belgium) *(DATE '22)*. European Design and Automation Association, Leuven, BEL, 1407–1412.

[42] Jee Ho Ryoo, Nagendra Gulur, Shuang Song, and Lizy K. John. 2017. Rethinking TLB Designs in Virtualized Environments: A Very Large Part-of-Memory TLB. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA '17)*. Association for Computing Machinery, New York, NY, USA, 469–480.

[43] Andreas Sandberg, Nikos Nikoleris, Trevor E Carlson, Erik Hagersten, Stefanos Kaxiras, and David Black-Schaffer. 2015. Full speed ahead: Detailed architectural simulation at near-native speed. In *2015 IEEE International Symposium on Workload Characterization*. IEEE, Atlanta, GA, USA, 183–192.

[44] S. Shahri, S. Armin Vakil Ghahani, and A. Kolli. 2020. (Almost) Fence-less Persist Ordering. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, Los Alamitos, CA, USA, 539–554.

[45] Dimitrios Skarlatos, Apostolos Kokolis, Tianyin Xu, and Josep Torrellas. 2020. Elastic Cuckoo Page Tables: Rethinking Virtual Memory Translation for Parallelism. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 1093–1108. https://doi.org/10.1145/3373376.3378493

[46] Shixin Song, Tanvir Ahmed Khan, Sara Mahdizadeh Shahri, Akshitha Sriraman, Niranjan K Soundararajan, Sreenivas Subramoney, Daniel A. Jiménez, Heiner Litz, and Baris Kasikci. 2022. Thermometer: Profile-Guided Btb Replacement for Data Center Applications. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 742–756.

[47] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. 2014. XS-Bench - The Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis. In *PHYSOR 2014 - The Role of Reactor Physics toward a Sustainable Future*. Kyoto.

[48] Ubuntu. 2020. Ubuntu Server. https://www.ubuntu.com/server

[49] Armin Vakil-Ghahani, Sara Mahdizadeh-Shahri, Mohammad-Reza Lotfi-Namin, Mohammad Bakhshalipour, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. 2018. Cache Replacement Policy Based on Expected Hit Count. *IEEE Comput. Archit. Lett.* 17, 1 (jan 2018), 64–67.

[50] Seyed Armin Vakil Ghahani, Mahmut Taylan Kandemir, and Jagadish B. Kotra. 2020. DSM: A Case for Hardware-Assisted Merging of DRAM Rows with Same Content. *Proc. ACM Meas. Anal. Comput. Syst.* 4, 2, Article 33 (jun 2020), 26 pages.

[51] Seyed Armin Vakil-Ghahani, Sara Mahdizadeh-Shahri, Mohammad Bakhshalipour, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. 2018. Making Belady-Inspired Replacement Policies More Effective Using Expected Hit Count. *CoRR* abs/1808.05024 (2018). arXiv:1808.05024 http://arxiv.org/abs/1808.05024

[52] Wikichip. 2017. Intel Skylake micro-architecture details. https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(server)#Memory_Hierarchy

[53] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. 2019. Translation Ranger: Operating System Support for Contiguity-Aware TLBs. In *Proceedings of the 46th International Symposium on Computer Architecture* (Phoenix, Arizona) *(ISCA '19)*. Association for Computing Machinery, New York, NY, USA, 698–710. https://doi.org/10.1145/3307650.3322223