Twentieth Int. Conf. on Embedded Computer Systems: Architecture, Modeling and Simulation (SAMOS XX), Samos, Greece, July 2020.

CoPTA: Contiguous Pattern Speculating TLB Architecture

Yichen Yang, Haojie Ye, Yuhan Chen, Xueyang Liu, Nishil Talati, Xin He, Trevor Mudge, and Ronald Dreslinski

University of Michigan, Ann Arbor MI 48109, USA
{yangych, yehaojie, chenyh, marliu, talatin, xinhe, tnm, rdreslin}@umich.edu

Abstract. With the growing size of real-world datasets running on CPUs, address translation has become a significant performance bottleneck. To translate virtual addresses into physical addresses, modern operating systems perform several levels of page table walks (PTWs) in memory. Translation look-aside buffers (TLBs) are used as caches to keep recently used translation information. However, as datasets increase in size, both the TLB miss rate and the overhead of PTWs worsen, causing severe performance bottlenecks. Using a diverse set of workloads, we show the PTW overhead consumes an average of 20% application execution time.

In this paper, we propose CoPTA, a technique to speculate the memory address translation upon a TLB miss to hide the PTW latency. Specifically, we show that the operating system has a tendency to map contiguous virtual memory pages to contiguous physical pages. Using a real machine, we show that the Linux kernel can automatically defragment physical memory and create larger chunks for contiguous mapping, particularly when transparent huge page support is enabled. Based on this observation, we devise a speculation mechanism that finds nearby entries present in the TLB upon a miss and predicts the address translation of the missed address assuming contiguous address allocation. This allows CoPTA to speculatively execute instructions without waiting for the PTW to complete. We run the PTW in parallel, compare the speculated and the translated physical addresses, and flush the pipeline upon a wrong speculation with similar techniques used for handling branch mispredictions.

We comprehensively evaluate our proposal using benchmarks from three suites: SPEC CPU 2006 for server-grade applications, GraphBIG for graph applications, and the NAS benchmark suite for scientific applications. Using a trace-based simulation, we show an average address prediction accuracy of 82% across these workloads resulting in a 16% performance improvement.

Keywords: virtual memory · page table walk · TLB · speculative execution.

1 Introduction

Virtual memory is widely used in modern computing systems because it offers an abstraction for different applications to own a large, exclusive memory space along with some security guarantees. Supporting virtual memory requires a processor to translate a virtual address to a physical memory address before requesting the data from the memory hierarchy. The Translation Look-aside Buffer (TLB) serves as a dedicated cache for storing address translation information. A miss in the TLB triggers a Page Table Walk (PTW), which incurs several serialized memory accesses. The core has to wait to receive the translated address before servicing the memory request, therefore, PTW can cause a serious performance bottleneck upon frequent TLB misses.

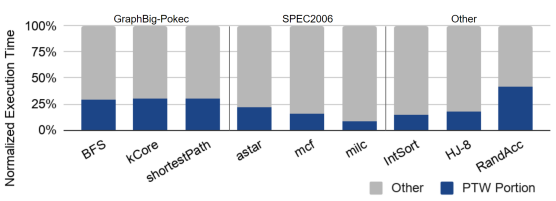


Fig. 1: Processor time breakdown for different benchmarks.

Modern big-data workloads use large volumes of data that can easily stress the TLB both due to limited TLB size and complex PTW procedure. TLBs employ a fully-associative structure with limited capacity (e.g., 64-entries for L1 TLBs in modern CPUs) in order to keep a reasonable lookup latency. To illustrate this bottleneck, we profiled several graph processing, server, and scientific workloads from GraphBIG [26] and SPEC CPU2006 [18] and elsewhere on an x86-based host machine. Fig. 1 shows the fraction of execution time spent on the PTW, which shows that a significant portion (more than 25% in some benchmarks) is spent on translating addresses upon TLB misses. Prior endeavors to mitigate this issue adopt superpages to increase TLB reach [34]. However, this approach suffers from internal fragmentation, wastes space, and stresses memory bandwidth.

The goal of this work is to reduce the overhead of PTW by speculating the physical address upon a TLB miss. To this end, we propose CoPTA, a Contiguous Pattern Speculating TLB Architecture, which speculates the address translation of a TLB misses using a nearby entry present the TLB. CoPTA exploits the opportunities that contiguous virtual addresses are likely to be mapped to contiguous physical addresses. This is supported by characterization experiments performed on workloads running in a real machine running Linux. By predicting a memory address translation, CoPTA allows the execution to speculatively proceed while performing the PTW in parallel. The benefit of such a scheme is that the core can execute dependent instructions without waiting for the translation. Given most contiguous virtual pages are mapped to contiguous physical pages, this speculation yields correct translation most of the time. In the event of miss-speculation, we use a technique similar to what is used for a branch misprediction to flush the speculatively executed instructions from the re-order buffer (ROB).

To evaluate the performance benefits of CoPTA, we use a trace-based simulation methodology. A variety of real-world big-data workloads from graph analytics, server applications, and scientific computing domains are used to evaluate CoPTA. We show that using a negligible 0.4KB of storage, CoPTA can achieve an average address translation prediction accuracy of 82%, which can potentially result in an average performance by 16%.

2 Background and Motivation

2.1 Virtual Address Translation

To achieve process isolation, each process issues instructions with virtual addresses [25]. This is an abstraction provided by the operating system (OS) that can (a) hide physical memory fragmentation, (b) leave the burden of managing the memory hierarchy to the kernel, and (c) create the illusion of an infinite address space for each process.

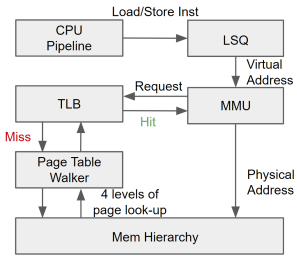


Fig. 2: Overview of address translation.

The translation of virtual addresses to physical addresses is enforced by the Memory Management Unit (MMU) (Fig. 2). Memory is typically split into 4KB pages, and a page table keeps track of all the mappings between a virtual page number (VPN) and a physical page number (PPN). To avoid storing all the translations in a monolithic large mapping table, MMU usually uses a hierarchical page table. For example, the x86 architecture uses 4 levels of page tables [3,5]. Even with this design, the overall size of the page table is too large to fit in any on-chip cache structure entirely. A PTW for address translation in an x86 system will incur 4 levels of page table lookup, which leads to a significant performance overhead. To alleviate the PTW latency, a dedicated cache called the translation look-aside buffer (TLB) is used to cache the recently used translations.

Each time an address translation is requested, the MMU will look up a corresponding entry in the TLB for faster access. Thus the TLB plays a very important role in the performance of address translation [15, 24]. Upon a TLB hit, the matched PPN combined with the offset will be used for the data access. If the TLB misses a request, it invokes a PTW that looks up the corresponding PPN in the memory hierarchy, and caches it into the TLB for future references.

Each level of the page table needs to be fetched from the main memory, so in the worst case, a single PTW can incur 4 times the memory latency before the MMU can access the physical address. Note that the PTW can be done in hardware (i.e., there is a dedicated controller to lookup the page tables) or in software (i.e., an interrupt subroutine is called to service the address translation lookup). When virtualization is involved, the nested page table lookup can take up to 24 memory access, increasing the latency to get the final page translation, which becomes a huge overhead [6, 17].

2.2 Physical Page Allocation

When a process requests a new page, the OS allocates a physical page to be mapped to the virtual page, either by allocating an unused page or by evicting a used page. Linux uses a buddy allocator that operates as follows: when an application makes a `malloc` call and asks for multiple pages, the OS will try to allocate continuous physical page frames whenever possible [2, 29]. In detail, when the process requests N pages at a time, the OS will look for N contiguous physical pages in the memory or break larger contiguous chunks to create N contiguous physical pages. Our experiment shows that the OS tends to map contiguous physical pages to virtual pages, which opens up an opportunity for our TLB speculation mechanism.

2.3 Memory Compaction and Transparent Hugepage Support

With applications constantly allocating and freeing pages, memory will be fragmented, making it hard to find a large contiguous region. Many OS provide a memory compaction daemon. Linux, for example, invokes memory compaction when it is hard to find groups of physically-contiguous pages. The OS will relocate the movable pages to the free pages, filling up the holes in memory and reducing fragmentation, also known as defragmentation.

Linux has a Transparent Hugepage Support (THS) mechanism, that allows the memory allocator attempting to find a free 2MB block of memory [2]. In the 2MB memory chunk, if the VPNs and PPNs are aligned, the OS will construct a 2MB superpage with 512 consecutive 4KB pages. Memory compaction and THS provide a better memory mapping that favors CoPTA.

3 Related Work

Previous works mainly focused on reducing the TLB miss rate and reducing page walk overhead.

Reduce TLB Miss Rate. CoLT [29] exploits and takes advantage of contiguous page allocation by coalescing contiguous page translations into one TLB entry, and therefore increasing the TLB reach. However, the maximum number of pages coalesced in CoLT is limited by the size of a cache line. Hybrid TLB coalescing [28] relaxes this limitation with the help of software. Our work has no limit to the range that the contiguous pattern can be searched, thus allowing a more flexible contiguous mapping range, not limited by the size of a cache line.

Superpages [9,34] increase the TLB coverage and therefore reduces miss rate. Works have shown efforts to exploit support for superpages by using either split TLB hardware [29] or unified TLB hardware for all page sizes [16]. Other works also explored ways to accelerate multiprocessor TLB access [8, 11]. CoPTA instead uses speculation and can be applied orthogonally to superpages, CoLT, and multiprocessors.

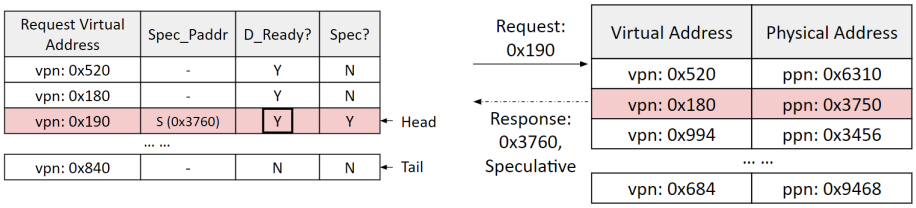
Hiding the Page Walk Latency. SpecTLB [6] utilizes a reservation-based physical memory allocator. When allocating memory, the handler will reserve

large chunks if a superpage is appropriate, and any future allocation in the reserved superpage will be aligned [27, 33]. SpecTLB requires the OS to support superpage reservation [27, 33], and can only provide speculation if the miss request is part of the superpage. Our work requires minor changes to the TLB, and can potentially provide speculations for any virtual page number because the searching range can be customized.

Prefetched Address Translation [23] modifies the OS to force contiguous page allocation. Therefore it is faster to locate the page table and faster to prefetch the translation, reducing the page walk latency.

4 CoPTA: TLB Speculation Architecture

In this section, we detail the proposed CoPTA architecture. We modify the TLB and load store queue (LSQ) to support the TLB speculation scheme, as shown in Fig. 3. The modified TLB architecture relaxes the exact matching requirement of address translation. Instead of searching for a single, exact match of the query in the TLB by a CAM circuit, CoPTA will return a hit entry upon matching a predefined number of most significant bits in the tag array. In this way, the *approximate* search will find a "close-neighbor" of the virtual address query, even if the exact match is missing in the TLB. The proposed speculation-supported LSQ adds a physical address column and 1 additional bit to each entry, totalling in 0.4KB additional storage for a 48 entry LSQ. The physical address column is used to indicate the value of the speculative physical address translation that has not been verified yet. The additional bit is used to indicate whether data in each LSQ entry is generated by a speculative translation.



(a) (b)

Fig. 3: (a) Speculation supported Load Store Queue (Data for VPN 0x190 may arrive earlier than the translation response from walker, the data remain speculative in LSQ before translation is resolved, `D_ready` indicates Data is ready to commit from LSQ). (b) Speculation supported Translation Lookaside Buffer (Incoming request VPN 0x190 will incur a miss but find the close neighbor (0x180) in the TLB, a speculative translation response 0x3760=0x3750+(0x190-0x180) is returned within Hit latency.)

With these modifications, CoPTA can parallelize the data request sent to the memory hierarchy and the hierarchical PTW process. The processor can then send data requests without stalling for the translation response to complete. The

pipeline is illustrated in Fig. 4. When encountering a TLB miss, conventional architectures will block the data request until the address translation is resolved. These events happen in the order of A1 → B1 → C1 → C2 → B2 → D1 → D2 → A2. By contrast, with CoPTA, the augmented L1 TLB is able to predict the physical address and issue a data request with the predicted physical address, along with sending the translation request to the page table walker, thus overlapping with the time interval during which the page table walker serves the translation miss. The speculated physical address translation will have a copy saved in the corresponding LSQ entry, with the marked speculation bit. The CPU pipeline can speculatively execute based on the predicted physical address. In Fig. 4, event A1 triggers event B1 with speculative response b2. Then C1→C2 happens in parallel with D1→D2 to hide the PTW latency, and event a2 send the speculated result to the LSQ. When the translation is resolved from the PTW, the verification signal is sent to the LSQ (C2→a2). The returned (accurate) physical address is compared against the prediction (saved in the LSQ) to verify its correctness. If the result matches the speculation, the speculation bit is cleared and the PTW latency can be hidden from the execution. If the speculation is incorrect, the CPU pipeline will be flushed similar to a branch miss-prediction, and the pipeline will roll back to the state before the request with the accurate physical address being issued. Given a high prediction accuracy, speculative execution can hide the PTW latency and improve TLB performance.

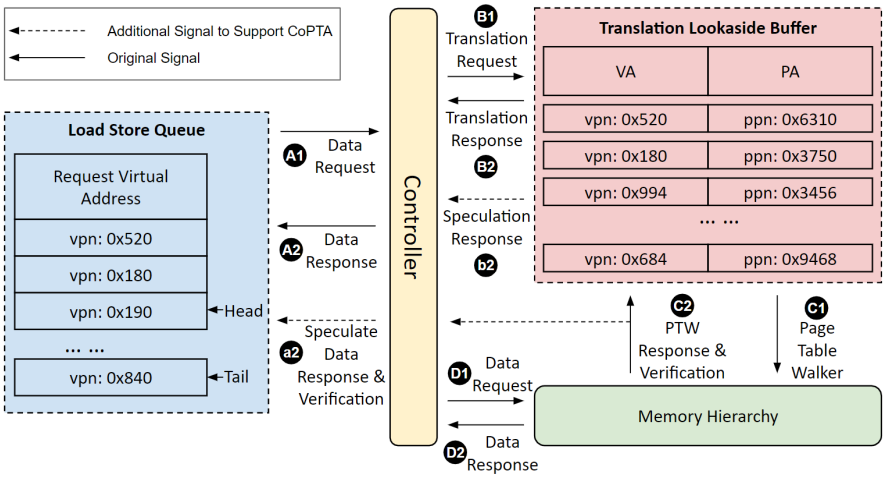


Fig. 4: Overview of TLB speculation architecture.

Regardless of whether an exact match/miss is obtained, CoPTA executes the same procedures as the original TLB design. The only difference is when the translation misses an exact match but hits a neighborhood match. The conventional design will block the data request for this LSQ entry until the translation is resolved, while the CoPTA will match a close-neighbor in the TLB, as shown in Fig. 3b. In the close-neighbor match scenario, the TLB returns

the speculative physical address based on the distance of virtual addresses between the request and its close-neighbor (in Figure 3b, the close-neighbor of requesting VPN=0x190 is VPN=0x180, the speculative PPN=0x3750+(0x190-0x180)=0x3760). The speculative address is returned to LSQ and attached with a speculation bit to the LSQ.

The data request is sent to the memory hierarchy in parallel with launching a PTW to fetch the accurate physical address translation. In this situation, the data may be returned to the corresponding LSQ entry earlier than the accurate translation and will be marked as a speculative data value, because the verified translation has not arrived to clear the speculation bit of the entry (Fig. 3a). The current register states will be check-pointed (e.g. using register renaming) and the commits beyond this speculation point will be saved in a write buffer to protect the memory state. When the accurate translation returns to the TLB, a verification signal will be set and the LSQ will be notified (Fig. 4). If the speculative translation matches with the accurate translation from the page walker, the speculation bit is cleared, and the registers and memory states can be safely committed. If the speculative translation does not match with the walker, the pipeline is flushed as if there is a misprediction. The speculative register state and the write buffer are cleared, and the CPU pipeline will restore to the point before issuing the mispredicted load/store request in the LSQ.

Note that pipeline flushes are relatively rare events. Flushes are only necessary when both the data arrives earlier than the page table walker in a TLB miss and a misprediction happens because the incorrect data has been retrieved and used. We discuss the overhead of misprediction in Section 8. When the accurate translation returns earlier than the outstanding data request, the address verification happens before the data is used. If the speculative translation matches the translation result from the page table walker, no further action needs to be performed. If the speculative translation does not match the accurate result, a second LSQ request is sent to the memory hierarchy and the previously speculative data request is discarded. Therefore, the data response of the previous LSQ request with an incorrect physical address is ignored.

5 Methodology

5.1 Real Machine Memory Allocation Characterization

The proposed CoPTA architecture relies on the contiguity of the system memory mapping. With a longer contiguous virtual memory address to physical memory address mapping region, this architecture will achieve higher prediction accuracy and better performance improvement. To characterize the contiguity of memory mapping on a real machine, we modified DynamoRio [1, 14] to dump the trace of the virtual to physical mappings from a real machine and then analyze the contiguity. The configuration of our modeled real machine is shown in Table 1.

Similar to the prior works [29] that study the effect of memory compaction, we manually invoke memory compaction in the Linux kernel and characterize

the memory contiguity. With memory compaction, the system will defragment the memory and create a larger chunk in the memory to map the following address allocation requests. By setting the Linux `defrag` flag, the system triggers the memory compaction for different situations. We also enable and disable the Linux transparent hugepage support (THS) to investigate how the system built-in memory defragmentation functions and how supperpage support helps our proposed architecture. The experimental machine has been running for a month to mimic typical memory fragmentation as compared to a clean boot. We study the memory contiguity under different configurations. Due to space constraints, we present the result of the following configurations: (a) before and after manually invoking memory compaction, (b) THS disabled, normal memory compaction (current default setting for Linux), (c) THS enabled, normal memory compaction, and (d) THS enabled, low memory compaction.

5.2 Simulation Based Speculation Evaluation

Similar to the previous works [5, 6, 8, 10, 29] that evaluate the performance of CoPTA, we use a trace-based approach. Performing an online prediction evaluation with a full-system out-of-order processor simulator is infeasible due to inordinate simulation time. Also, it is not possible to evaluate the performance on a real machine because CoPTA needs hardware modifications that cannot be emulated on the host machine. To collect the translation trace, we apply a simulation-based approach instead of using real machine traces to avoid interference from DynamoRio [1,14]. We run the benchmarks on the gem5 simulator [12] in full system mode and modify the simulator to collect the virtual to physical address translation trace. The trace is then fed into a customized TLB simulator. Prior works on TLBs [5, 6, 8, 32] mainly focus on measuring the TLB miss rate. In addition, we estimate the overall performance improvement by assuming the number of L1 TLB misses is directly propotional to the time spent on the PTW, following the methodology used by Basu *et al.* [7]. We run Linux `perf` on a real machine, using hardware counters to collect the average and total number of cycles spent on PTWs. The performance improvement metric is defined in Equation 1, which is calculated by multiplying the portion of time spent on the PTWs (T_{PTW}) with the TLB predictor hit rate ($A_{CoPTA\_prediction}$). This is an upper-bound estimation of CoPTA because we make several assumptions, including an optimal misprediction penalty and collecting the translation trace under the best memory conditions, detailed in Section 8.

$$Performance\_Improvement = T_{PTW} \times A_{CoPTA\_prediction} \tag{1}$$

Table 1 shows the specification of the real machine and gem5 simulator setup. Both run a similar Linux kernel on x86 ISA processors to ensure a similar page assignment strategy. The customized TLB simulator has a 64-entry, fully-associative L1 TLB and no L2 TLB, adopting the least recently used (LRU) replacement policy. When the TLB simulator matches multiple close-neighbors

(based on the higher order bits of the virtual addresses), it selects the first hit and calculates the result based on its physical address.

5.3 Benchmarks

We study the memory mapping contiguity on the graph workloads from GraphBig [26] with the SlashDot dataset [20], and later measure the performance on GraphBig [26], SPEC CPU2006 benchmark suite [18], NAS [4], Hash Join [13] and the HPC Challenge Benchmark [22]. For graph workloads, two different datasets are used. SlashDot and Pokec are real-world social network graphs from SNAP [20]. They have 0.08M nodes, 0.95M edges and 1.6M nodes, 30M edges, respectively. For the SPEC CPU2006 benchmark suite, the largest inputs are used. Detailed information about the benchmarks is listed in Table 2.

Table 1: Experiment setup specifications.

	Host Machine	gem5 Simulator
Processor	Intel i7-6700K	X86 single-core atomic
Linux Kernel	4.15.0	4.8.13
L1 Cache	32kB	32kB
L2 Cache	256kB	256kB
L3 Cache	2MB	No
RAM	16GB	16GB
L1 TLB	64 entries	64 entries
L2 TLB	1536 entries	No

Table 2: Summary of the benchmarks evaluated.

Benchmark	Source	Input
BFS	GraphBig [26]	SlashDot & Pokec [20]
ConnectedComp	GraphBig [26]	SlashDot & Pokec [20]
kCore	GraphBig [26]	SlashDot & Pokec [20]
PageRank	GraphBig [26]	SlashDot & Pokec [20]
ShortestPath	GraphBig [26]	SlashDot & Pokec [20]
TriangleCount	GraphBig [26]	SlashDot & Pokec [20]
astar	SPEC [18]	ref input
lbm	SPEC [18]	ref input
mcf	SPEC [18]	ref input
milc	SPEC [18]	ref input
IntSort	NAS [4]	/
ConjGrad	NAS [4]	/
HJ-8	Hash Join [13]	-r 12800000 -s 12800000
RandAcc	HPCC [22]	100000000

6 Memory Allocation Contiguity Characterization

We qualitatively analyze the effect of the Linux buddy allocator, memory compaction and THS on the memory contiguity on a real machine. Here we define the address mapping contiguity as follows: if contiguous virtual pages VP, VP+1, VP+2 are mapped to contiguous physical pages PP, PP+1, PP+2, the contiguity is 3. Therefore, a contiguity of 1 means this address mapping is not contiguous with any other. We use a cumulative density function (CDF) to show the contiguity of the memory allocation for some graph workloads (Fig. 5a to 6c). Note that the x-axis is in a log scale. A steep line to the right indicates that the memory mapping is more contiguous, thus benefiting CoPTA more. To illustrate these we present graph workloads and use the moderately sized SlashDot dataset as an input.

6.1 Memory Compaction Effect

We first characterize the effect of memory compaction. Fig. 5 shows the memory contiguity before and after manually invoking memory compaction. Before

memory compaction (Fig. 5a), some benchmarks reflect a certain level of contiguity. But when manually invoking the memory compaction before launching each benchmark (Fig. 5b), all the benchmarks show better memory contiguity, as a result of the Linux buddy allocator. As long as there are unused contiguous memory spaces in physical memory, the buddy allocator will assign these contiguous physical addresses to contiguous virtual addresses. Thus, memory compaction along with the buddy allocator provides the prerequisite for CoPTA.

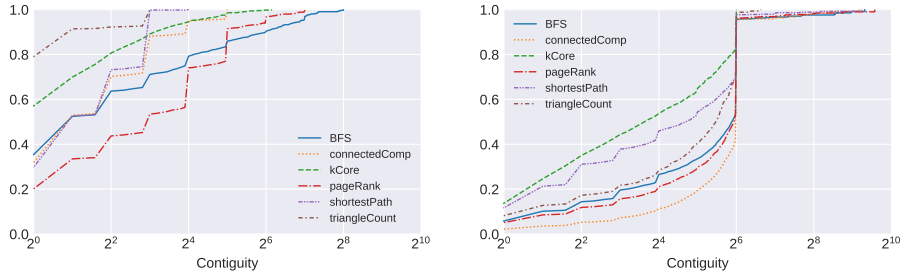
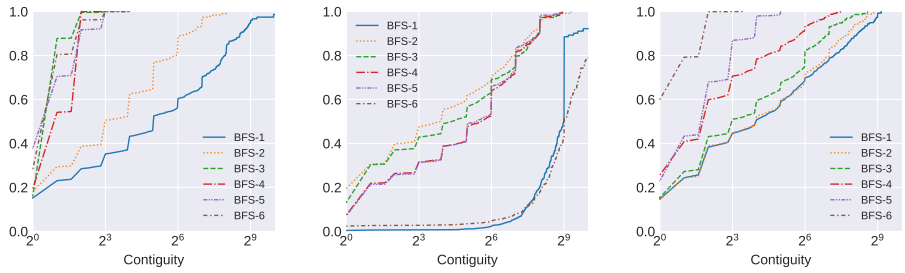


Fig. 5: (a) CDF for memory contiguity before memory compaction. (b) CDF for memory contiguity after memory compaction.

6.2 Effect of Transparent Hugepage Support

Fig. 6 shows the memory contiguity under different Linux kernel configurations. Here, we first manually invoke memory compaction and launch BFS six times sequentially. BFS-1 stands for the first run, BFS-2 stands for the second run, etc. In Fig 6a, the memory contiguity decreases over time, as the memory is not defragmented by the Linux kernel. When THS is disabled, the system doesn't need to reserve larger memory fragments for hugepages, so the Linux kernel compaction is not triggered after 6 runs of BFS. A similar situation is observed in Fig 6c. As we disabled the `defrag` flag in the kernel, the system does not trigger memory compaction automatically, thus resulting in decreased memory contiguity. When THS is enabled with the normal `defrag` flag (Fig 6b), the memory contiguity decreases for the first five runs and returns to a good condition during the sixth run. The memory compaction is automatically triggered during that time, resulting in memory defragmentation.

With auto-triggered memory compaction, the memory mapping contiguity is bounded within a certain range. When the memory is highly fragmented, the kernel will trigger memory compaction to defragment the memory mapping. This bounds the performance benefit from CoPTA. High memory fragmentation will lead to performance degradation on the CoPTA architecture. To correct this, the system will thus trigger memory compaction automatically and CoPTA can continue offering high performance improvements.



(a) (b) (c)

Fig. 6: (a) CDF for memory contiguity with normal `defrag`, THS disabled. (b) CDF for memory contiguity with normal `defrag`, THS enabled. (c) CDF for memory contiguity disable `defrag`, THS enabled.

7 CoPTA Performance Evaluation

In this section, we quantitatively evaluate the performance improvements of our proposed CoPTA architecture and justify it by presenting a reduced TLB miss rate compared to the baseline.

7.1 CoPTA Prediction Accuracy

Fig. 7 shows the address translation prediction accuracy of CoPTA. The gem5 full-system simulator is set to disable THS and normal `defrag` to mimic the default setting in Linux. The benchmarks begin execution in a state of minimal memory fragmentation after the simulator has booted up. This state is similar to the memory condition after the memory compaction is triggered in a real machine. Note that these conditions estimate the upper bound of the performance benefits from CoPTA. For the graph workloads, the CoPTA achieves a higher prediction accuracy with a smaller dataset. The accuracy of a smaller dataset SlashDot is 80% compared to 55% for a larger dataset Pokec. For other workloads, the prediction accuracy is higher than irregular graph workloads. The overall average prediction accuracy is 82%.

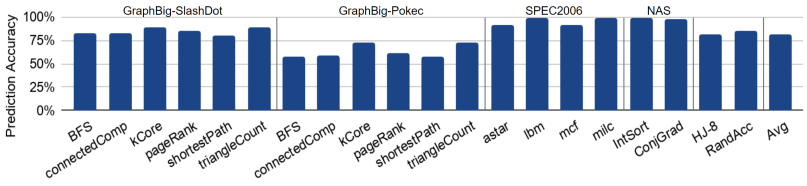


Fig. 7: CoPTA prediction accuracy.

7.2 Improvement Over the Baseline

Fig. 8 compares the TLB miss rates of CoPTA with the baseline. With CoPTA, the average L1 TLB miss rate reduced to 2%, where it is 8% at the baseline. For

the graph workloads with a larger dataset, the TLB size is insufficient to cover the whole data range in the baseline, which results in a higher TLB miss rate of 11%. Other benchmarks like RandAcc incur a higher TLB miss rate of 21% that is also caused by the irregular data access pattern over a large working set size. CoPTA predicts address translations well in these cases, reducing the TLB miss rate by 82% on average.

Fig. 9 compares the percentage of time spent on the PTWs by CoPTA and the baseline. For the baseline, this is collected on a real machine with hardware counters. For CoPTA, we only scale the fraction of PTW time spent by the CPU using CoPTA prediction estimations. Graph workloads with a larger dataset show a higher portion of the execution time spent on the PTWs. The performance loss due to time spent on PTWs becomes a serious bottleneck for some workloads (e.g., RandAcc) and can be as high as 40%. Although some of this 40% can be overlapped with out-of-order execution, the pipeline will quickly fill the instruction buffer with dependent instructions and stall. By predicting the address translations, CoPTA is able to reduce an average fraction of the application execution time spent on the PTW to 4% compared to 20% in the baseline.

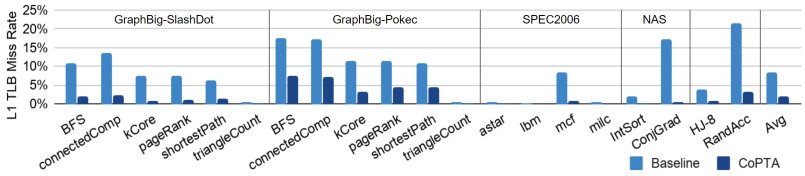


Fig. 8: L1 TLB miss rate for baseline and CoPTA.

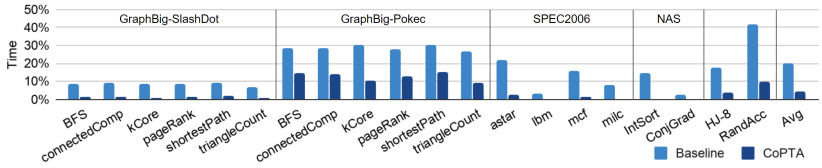


Fig. 9: Percentage of time spent on PTWs for baseline and CoPTA.

7.3 Overall Performance Improvement

Fig. 10 shows that CoPTA improves the performance of the baseline by 16% on average (up to 35%). To evaluate this, we used the performance improvement rule defined in Equation 1. For graph workloads, even though the address translation prediction accuracy is relatively low on larger datasets, the percentage of time spent on PTWs is dominant enough to gain a significant performance boost. CoPTA improves the performance of irregular workloads with larger data sets

better than others. With `defrag` flag enabled in Linux, the system will automatically defragment the memory when the system is too fragmented (Fig. 6b) and create as large of a contiguous region as possible. This is similar to the optimal memory condition in our full-system simulation-based experiments.

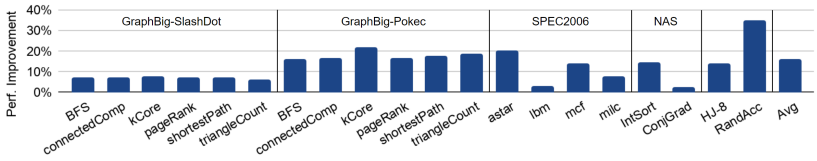


Fig. 10: Performance improvement estimation with CoPTA.

8 Discussion

Performance Improvement Metric. The performance improvement metric we defined in Equation 1 is based on several assumptions and limitations. First, the portion of time spent on the PTWs is directly propotional to the L1 TLB miss rate. Ideally, executing the PTW in parallel with the following instructions will remove the PTW portion from the critical path, but this may incur structural stalls in the pipeline. This effect is difficult to quantitatively analyze. Second, we ignore the overhead of miss-speculation recovery. Even if miss-speculation happens, the data should respond earlier than the page table walker, and the CPU pipeline should be flushed and start over. This may also pollute the cache with data from miss-speculated addresses. Based on our experiments, pipeline flushes will rarely occur because of the high prediction accuracy. Furthermore the cost of flushing the pipeline is as small as branch miss-speculation. For these reasons, we consider the miss-speculation cost is negligible. Finally, we used the translation trace collected in the gem5 full-system simulator and use the `perf` result from a real machine. Because the benchmark starts execution right after the system is booted up and there are no other processes running simultaneously, the memory is not fragmented and the the Linux buddy allocator can assign the contiguous physical pages to the memory allocation requests, which is the optimal condition for CoPTA to achieve the best performance.

Software vs. Hardware Page Walker. Different architectures use different mechanisms for page walkers. With a software page walker, CoPTA will have limited benefits since the instructions for a PTW will be executed by the core, preventing the translation results from being actually used. However, the hardware page walker will execute the PTW in the background and let the pipeline continue executing the proceeding instructions.

Virtualization. PTW takes as many as 4 memory accesses on a host machine. But with virtualization involved, a CPU will have a nested page table where

the host and guest OS manage their own set of page tables, shown in previous works [6,17,23,30,31]. In this case, 4 memory accesses will be needed in the host OS to find the next level hierarchy page table of the guest OS, thus resulting in as many as 24 memory accesses [17]. If the prediction accuracy can maintain at the same level, with larger overhead of PTWs in the virtualization environment, we expect better performance improvement of our proposed CoPTA architecture. We leave this as future works.

Security Implication. CoPTA adopts the speculation idea for address translation, similar to branch prediction and speculatively reading data from the store queue. Attacks such as Spectre [19] and Meltdown [21] can exploit this feature to steal information from the system by accessing cache when doing speculative execution. CoPTA does not introduce any new types of security vulnerabilities in addition to what already exists for speculation.

9 Conclusion

TLB misses introduce a non-trivial overhead as page table walks require several slow main memory accesses. This can especially be problematic for workloads with large data footprints and irregular memory accesses with insignificant data locality (e.g. graph processing workloads). Our experiments show that the overhead caused by the page table walks can take up to 40% of total execution time. We also demonstrate that the built-in Linux buddy allocator and defragmentation mechanism tend to map contiguous virtual pages to contiguous physical pages. Motivated by this observation, we propose the CoPTA architecture that leverages this mechanism to predict the address translations upon TLB misses and speculatively execute proceeding instructions while concurrently performing the page table walk. With a negligible storage requirement of 0.4KB, CoPTA achieves an average address prediction accurate of 82% while improving end-to-end performance by 16%.

References

1. Dynamic instrumentation tool platform. https://dynamorio.org/
2. Linux kernel documentation. https://www.kernel.org/doc/
3. Advanced micro devices. amd x86-64 architecture programmer's manual (2002)
4. Bailey, D.H., Barszcz, E., Barton, J.T., Browning, D.S., Carter, R.L., Dagum, L., Fatoohi, R.A., Frederickson, P.O., Lasinski, T.A., Schreiber, R.S., et al.: The nas parallel benchmarks summary and preliminary results. In: Supercomputing'91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing. pp. 158–165. IEEE (1991)
5. Barr, T.W., Cox, A.L., Rixner, S.: Translation caching: skip, don't walk (the page table). In: ACM SIGARCH Computer Architecture News. vol. 38, pp. 48–59. ACM (2010)

6. Barr, T.W., Cox, A.L., Rixner, S.: Spectlb: a mechanism for speculative address translation. In: ACM SIGARCH Computer Architecture News. vol. 39, pp. 307–318. ACM (2011)
7. Basu, A., Gandhi, J., Chang, J., Hill, M.D., Swift, M.M.: Efficient virtual memory for big memory servers. In: Proceedings of the 40th Annual International Symposium on Computer Architecture. pp. 237–248. ISCA '13, ACM, New York, NY, USA (2013). https://doi.org/10.1145/2485922.2485943, http://doi.acm.org/10.1145/2485922.2485943
8. Bhattacharjee, A., Lustig, D., Martonosi, M.: Shared last-level tlbs for chip multiprocessors. In: 2011 IEEE 17th International Symposium on High Performance Computer Architecture. pp. 62–63 (Feb 2011). https://doi.org/10.1109/HPCA.2011.5749717
9. Bhattacharjee, A., Lustig, D.: Architectural and operating system support for virtual memory. Synthesis Lectures on Computer Architecture **12**(5), 1–175 (2017)
10. Bhattacharjee, A., Martonosi, M.: Characterizing the tlb behavior of emerging parallel workloads on chip multiprocessors. In: 2009 18th International Conference on Parallel Architectures and Compilation Techniques. pp. 29–40. IEEE (2009)
11. Bhattacharjee, A., Martonosi, M.: Inter-core cooperative tlb for chip multiprocessors. SIGARCH Comput. Archit. News **38**(1), 359–370 (Mar 2010). https://doi.org/10.1145/1735970.1736060, https://doi.org/10.1145/1735970.1736060
12. Binkert, N., Beckmann, B., Black, G., Reinhardt, S.K., Saidi, A., Basu, A., Hestness, J., Hower, D.R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M.D., Wood, D.A.: The gem5 simulator. SIGARCH Comput. Archit. News **39**(2), 1–7 (Aug 2011). https://doi.org/10.1145/2024716.2024718, http://doi.acm.org/10.1145/2024716.2024718
13. Blanas, S., Li, Y., Patel, J.M.: Design and evaluation of main memory hash join algorithms for multi-core cpus. In: Proceedings of the 2011 ACM SIGMOD International Conference on Management of data. pp. 37–48 (2011)
14. Bruening, D.L.: Efficient, Transparent, and Comprehensive Runtime Code Manipulation. Ph.D. thesis, Cambridge, MA, USA (2004), aAI0807735
15. Chen, J.B., Borg, A., Jouppi, N.P.: A simulation based study of tlb performance. In: Proceedings of the 19th Annual International Symposium on Computer Architecture. p. 114–123. ISCA '92, Association for Computing Machinery, New York, NY, USA (1992). https://doi.org/10.1145/139669.139708, https://doi.org/10.1145/139669.139708
16. Cox, G., Bhattacharjee, A.: Efficient address translation for architectures with multiple page sizes. ACM SIGOPS Operating Systems Review **51**(2), 435–448 (2017)
17. Gandhi, J., Basu, A., Hill, M.D., Swift, M.M.: Efficient memory virtualization: Reducing dimensionality of nested page walks. In: 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture. pp. 178–189 (Dec 2014). https://doi.org/10.1109/MICRO.2014.37
18. Henning, J.L.: Spec cpu2006 benchmark descriptions. ACM SIGARCH Computer Architecture News **34**(4), 1–17 (2006)
19. Kocher, P., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., Yarom, Y.: Spectre attacks: Exploiting speculative execution. 2019 IEEE Symposium on Security and Privacy (SP) pp. 1–19 (2018)
20. Leskovec, J., Krevl, A.: SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data (Jun 2014)

21. Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., Hamburg, M.: Meltdown: Reading kernel memory from user space. In: 27th USENIX Security Symposium (USENIX Security 18). pp. 973–990. USENIX Association, Baltimore, MD (Aug 2018), https://www.usenix.org/conference/usenixsecurity18/presentation/lipp
22. Luszczek, P.R., Bailey, D.H., Dongarra, J.J., Kepner, J., Lucas, R.F., Rabenseifner, R., Takahashi, D.: The hpc challenge (hpcc) benchmark suite. In: Proceedings of the 2006 ACM/IEEE conference on Supercomputing. vol. 213, pp. 1188455–1188677. Citeseer (2006)
23. Margaritov, A., Ustiugov, D., Bugnion, E., Grot, B.: Prefetched address translation. In: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture. pp. 1023–1036. ACM (2019)
24. McCurdy, C., Cox, A., Vetter, J.: Investigating the tlb behavior of high-end scientific applications on commodity microprocessors. pp. 95–104 (05 2008). https://doi.org/10.1109/ISPASS.2008.4510742
25. Mittal, S.: A survey of techniques for architecting tlbs. Concurrency and Computation: Practice and Experience **29**(10), e4061 (2017)
26. Nai, L., Xia, Y., Tanase, I.G., Kim, H., Lin, C.: Graphbig: understanding graph computing in the context of industrial solutions. In: SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–12 (Nov 2015). https://doi.org/10.1145/2807591.2807626
27. Navarro, J., Iyer, S., Druschel, P., Cox, A.: Practical, transparent operating system support for superpages. SIGOPS Oper. Syst. Rev. **36**(SI), 89–104 (Dec 2003). https://doi.org/10.1145/844128.844138, https://doi.org/10.1145/844128.844138
28. Park, C.H., Heo, T., Jeong, J., Huh, J.: Hybrid tlb coalescing: Improving tlb translation coverage under diverse fragmented memory allocations. In: Proceedings of the 44th Annual International Symposium on Computer Architecture. pp. 444–456 (2017)
29. Pham, B., Vaidyanathan, V., Jaleel, A., Bhattacharjee, A.: Colt: Coalesced large-reach tlbs. In: Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture. pp. 258–269. IEEE Computer Society (2012)
30. Pham, B., Veselỳ, J., Loh, G.H., Bhattacharjee, A.: Large pages and lightweight memory management in virtualized environments: Can you have it both ways? In: Proceedings of the 48th International Symposium on Microarchitecture. pp. 1–12 (2015)
31. Ryoo, J.H., Gulur, N., Song, S., John, L.K.: Rethinking tlb designs in virtualized environments: A very large part-of-memory tlb. ACM SIGARCH Computer Architecture News **45**(2), 469–480 (2017)
32. Saulsbury, A., Dahlgren, F., Stenström, P.: Recency-based tlb preloading. In: Proceedings of the 27th annual international symposium on Computer architecture. pp. 117–127 (2000)
33. Talluri, M., Hill, M.D.: Surpassing the tlb performance of superpages with less operating system support. SIGOPS Oper. Syst. Rev. **28**(5), 171–182 (Nov 1994). https://doi.org/10.1145/381792.195531, https://doi.org/10.1145/381792.195531
34. Zhen Fang, Lixin Zhang, Carter, J.B., Hsieh, W.C., McKee, S.A.: Reevaluating online superpage promotion with hardware support. In: Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture. pp. 63–72 (Jan 2001). https://doi.org/10.1109/HPCA.2001.903252