Pushing the Envelope on Free TLB Prefetching

Georgios Vavouliotis*†, Lluc Alvarez*†, Marc Casas*
*Barcelona Supercomputing Center, Barcelona, Spain †Universitat Politècnica de Catalunya, Barcelona, Spain
E-mail: {georgios.vavouliotis, lluc.alvarez, marc.casas}@bsc.es

Keywords—TLB, prefetching, microarchitecture, caches.

I. Extended Abstract

Frequent Translation Lookaside Buffer (TLB) misses pose significant performance and energy overheads due to page walks required for fetching the translations. The address translation performance bottleneck is further exacerbated by the advent of big data and graph processing workloads due to their massive data footprints. Prefetching page table entries (PTEs) ahead of demand TLB accesses is an intuitively effective approach for alleviating the TLB performance bottleneck. However, each TLB prefetch request implies traversing the page table to fetch the corresponding PTE, triggering additional accesses to the memory hierarchy. Therefore, TLB prefetching is a promising, although costly, technique that may undermine performance when the prefetches are not accurate.

This work exploits the locality in the last level of the page table to reduce the cost and enhance the performance benefits of TLB prefetching by prefetching adjacent PTEs "for free". We design *Dynamic Free TLB Prefetching (DFTP)*, a scheme that predicts via sampling the usefulness of these "free" PTEs and prefetches only the ones most likely to save TLB misses. DFTP can be combined with any TLB prefetcher to provide further performance enhancements by exploiting page table locality for both demand and prefetch page walks.

A. Dynamic Free TLB Prefetching (DFTP)

1) Motivation: Figure 1 depicts the operation of a x86-64 page walk and illustrates the locality of the PTEs in the last level of the page table. PTEs are stored contiguously in memory, and each PTE is 8B, so a single cache line can store 8 PTEs. When the requested PTE is read from memory at the end of a page walk, it is grouped with 7 neighboring PTEs and they are stored into a single 64B cache line. Hence, a cache line holds the requested PTE plus 7 more PTEs that do not require additional memory operations to be prefetched.

The naive approach is to prefetch all available free PTEs into a TLB Prefetch Buffer (PB)[1]. However, TLB prefetching is limited by the PB size, the PB area overhead, and the cost of PB lookups. Thus, naively storing all free prefetches per page walk into the PB may limit the performance benefits by evicting useful prefetches and polluting the PB with inaccurate prefetches. Hence, to exploit page table locality with a realistic PB size, a scheme that dynamically identifies and prefetches only the useful free prefetches per page walk is required.

2) Design and Operation: To address the findings of Section I-A1, we design *Dynamic Free TLB Prefetching (DFTP)*, a scheme that predicts via sampling the usefulness of the different free PTEs per page walk, and fetches in the PB only the most useful ones. We define *free distance* as the distance,

[1]TLB prefetchers typically use a prefetch buffer to store the prefetches since prefetching directly into the TLB can negatively affect performance [1], [2].

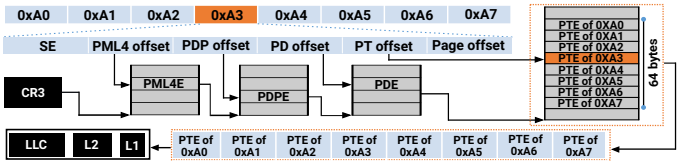

Fig. 1. Page table locality on x86-64 page table walks.

within the cache line, between the PTE that holds the demand translation and another free PTE. Depending on the cache line position of the requested PTE, there are 14 possible free distances: from -7 to +7, excluding 0.

The DFTP scheme associates each free PTE with a free distance and exploits this information to predict the usefulness of the corresponding PTEs. Figure 2 presents the components and the functionality of DFTP: the *Sampling Queue (SQ)*, the *Free Distance Table (FDT)* and the *Prefetch Buffer (PB)*. The SQ is a small buffer that detects phases when free distances, which were previously useless, can provide useful prefetches. Each SQ entry stores the virtual page and its corresponding free distance for every free PTE that is decided not to be placed in the PB. The decision whether to place a free prefetch into the PB or the SQ is made by the FDT, a table with 14 counters; each counter monitors the hit ratio of one free distance. The PB is a buffer that stores the virtual page, the physical page and the corresponding free distance of the prefetches.

To explain the operation of DFTP, we consider the example presented in Figure 2 that assumes a page walk triggered by virtual page 0xF3. First, we identify the position of the requested PTE inside the cache line by extracting the 3 least significant bits of the page. Then we calculate the free distances of all PTEs residing in the same cache line and we associate each PTE with a free distance.

To determine whether a free prefetch has to be placed in the PB or the SQ, we compare the FDT saturating counter corresponding to its free distance with a threshold. If the counter exceeds the threshold, the free prefetch is fetched in the PB; otherwise, is placed in the SQ. The same procedure is followed for each free PTE in the cache line.

On PB or SQ hits, the FDT counter that corresponds to the free distance of the hit entry is increased. To prevent permanent saturation, we shift right one bit all the FDT counters when one of the counters saturates.

To summarize, DFTP adjusts the values of FDT counters depending on which free distances are frequently producing PB or SQ hits, thus DFTP is able to adapt to phase-behavior and predict the most useful free PTEs per page walk.

3) Combining DFTP with TLB prefetching schemes: Apart from fetching the most useful free prefetches per demand page walk, i.e., a page walk due to a demand TLB miss, DFTP is also able to operate on prefetch page walks, i.e., page walks triggered by TLB prefetch requests. Specifically, at the end of a prefetch page walk the prefetched PTE is grouped with 7 PTEs that can be prefetched for free due to page table locality.

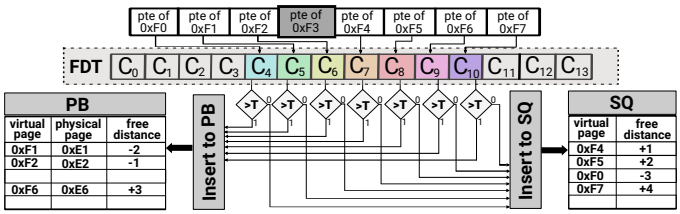

Fig. 2. Dynamic Free TLB Prefetching (DFTP) module.

At this point, DFTP is activated to decide which of the free prefetches should be placed in the PB or the SQ, essentially applying lookahead prefetching with depth 2.

4) Methodology: We consider a big set of industrial workloads provided by Qualcomm (QMM) for CVP1 [3], the SPEC CPU 2006 [4] and SPEC CPU 2017 [5] suites, and big data workloads included in the GAP [6] suite and the XSBench [7]. We refer to GAP and XSBench workloads as Big Data (BD) workloads. Our evaluation takes into account the workloads with a TLB MPKI of at least 1. All traces have been obtained using the SimPoint [8] methodology. For the QMM workloads we use 50M warmup instructions and 100M instructions for measuring the results. The rest of the workloads run 250M warmup instructions and 1B instructions are executed to measure the experimental results.

For evaluation we use ChampSim [9], a detailed simulator that models a 4-wide out-of-order processor. We extend ChampSim with a realistic x86 page table walker, modeling (i) the variant latency cost of page walks, (ii) the page walk references to memory hierarchy, and (iii) the cache locality in page walks.The page table walker supports up to 4 concurrent TLB misses [10], while one page walk can be initiated per cycle. Table I summarizes our experimental setup.

TLB Prefetchers. We consider the state-of-the-art TLB prefetchers: (i) Sequential Prefetcher (SP); SP [2] prefetches the PTE located next to the one that triggered the TLB miss, (ii) Arbitrary Stride Prefetcher (ASP); ASP [2] is a table-based prefetcher that captures miss streams with varying strides, and (iii) Distance Prefetcher (DP); DP [2] is a table-based prefetcher that correlates miss patterns with distances between pages that produce consecutive TLB misses. Our evaluation considers the most common scenario where a Prefetch Buffer (PB) is used to store the prefetched PTEs (Section I-A1).

5) Evaluation: To highlight the benefits of DFTP we compare it against the following scenarios: (i) free prefetches are not exploited (NoFP), *i.e.*, they are not stored in the PB; (ii) all free prefetches are naively placed in the PB (NaiveFP).

The performance impact of the above explained scenarios for the state-of-the-art TLB prefetchers is presented in Figure 3. We observe that all prefetchers achieve high performance gains for all scenarios considering free prefetching (NaiveFP, DFTP) than when free prefetching is not exploited (NoFP). We observe this behavior because (i) the free prefetches provide PB hits that reduce demand page walks, and (ii) most of the prefetch requests have already been prefetched for free, avoiding prefetch page walks. For instance, SP+DFTP outperforms SP+NoFP by 5.6% for the SPEC workloads.

Furthermore, we observe that DFTP significantly improves performance over NaiveFP for the QMM and SPEC workloads, across all prefetchers. For the BD workloads we observe that DFTP and NaiveFP provide similar performance benefits because these workloads exhibit highly irregular patterns, thus

Component	Description
L1 DTLB	64-entry, 4-way, 1-cycle, 4-entry MSHR
L2 TLB	1536-entry, 12-way, 8-cycle, 4-entry MSHR, 1 page walk / cycle
Prefetch Buffer (PB)	64-entry, fully assoc, 2-cycle
Sampling Queue (SQ)	64-entry, fully assoc, 2-cycle
L1 DCache	32KB, 8-way, 4-cycle, 8-entry MSHR, next line prefetcher
L2 Cache	256KB, 8-way, 8-cycle, 16-entry MSHR, ip stride prefetcher
LLC	2MB, 16-way, 20-cycle, 32-entry MSHR
DRAM	4GB, DDR4, 4GHz, 1600 MT/s

TABLE I. SYSTEM SIMULATION PARAMETERS.

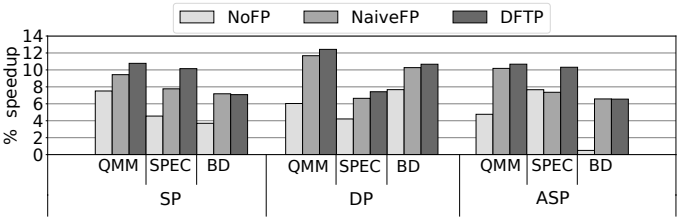

Fig. 3. Performance impact of free TLB prefetching scenarios.

it is difficult to detect the most useful free PTEs per page walk. Finally, we expect that designing a smarter TLB prefetcher would highlight more the benefits of DFTP over NaiveFP; we leave this exploration as future work.

B. Conclusions

This work reveals the importance of exploiting page table locality for TLB prefetching purposes. We propose DFTP, a dynamic scheme that identifies the most useful free PTEs per page walk, and we show that DFTP can be combined with any TLB prefetcher to provide great performance enhancements.

REFERENCES

[1] A. Bhattacharjee and M. Martonosi, "Inter-core Cooperative TLB for Chip Multiprocessors," in *Proceedings of the 15th Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XV. NY, USA: ACM, 2010, pp. 359–370.

[2] G. B. Kandiraju and A. Sivasubramaniam, "Going the Distance for TLB Prefetching: An Application-driven Study," in *29th Annual International Symposium on Computer Architecture*, ser. ISCA '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 195–206.

[3] "CVP-1," https://www.microarch.org/cvp1/.

[4] "SPEC CPU 2006," https://www.spec.org/cpu2006/, [Online].

[5] "SPEC CPU 2017," https://www.spec.org/cpu2017/, [Online].

[6] S. Beamer et al., "The GAP benchmark suite," *CoRR*, vol. abs/1508.03619, 2015.

[7] "XSBench," https://github.com/ANL-CESAR/XSBench.

[8] E. Perelman et al., "Using simpoint for accurate and efficient simulation," *SIGMETRICS Perform. Eval. Rev.*, vol. 31, no. 1, pp. 318–319, Jun. 2003.

[9] "ChampSim," https://crc2.ece.tamu.edu/, [Online].

[10] Abishek Bhattacharjee, "Advanced concepts on address translation," http://www.cs.yale.edu/homes/abhishek/abhishek-appendix-l.pdf.

Georgios Vavouliotis received his Diploma on Electrical and Computer Engineering from National Technical University of Athens (NTUA), Athens in 2018. Since fall 2018, he has been a Ph.D. candidate at the Computer Architecture department of Universitat Politècnica de Catalunya (UPC), Spain, and he has been working on the Runtime Aware Architecture research group of Barcelona Supercomputing Center (BSC).