Software-Based Prefetching: How Much Can It Buy?

Roland Kühn
roland.kuehn@cs.tu-dortmund.de
TU Dortmund University

Jan Mühlig
jan.muehlig@tu-dortmund.de
TU Dortmund University

Jens Teubner
jens.teubner@cs.tu-dortmund.de
TU Dortmund University

ABSTRACT

Software-based prefetching is an effective method for tolerating one of the most formidable barriers encountered by data processing systems: *memory latency*. Although the idea appears simple—just inform the CPU about upcoming data accesses—the intricacies of its implementation remain insufficiently understood. Existing works demonstrate how algorithms need to be prepared for prefetching, yet they often overlook the limitations and hardware implications of bringing data into the cache hierarchy. In this paper, we thoroughly examine software-based prefetching by delving into its implementation and identifying pitfalls across various platforms. Furthermore, we provide actionable insights and recommendations for developers seeking to boost their applications through this technique.

1 INTRODUCTION

Memory bandwidth and latencies present one of the highest barriers on the way to system performance optimization. While many steps have been made to increase the DRAM bandwidth, latency improvements have hit a plateau, leaving a chasm between potential and realized system speed. And new memory technologies, such as high-bandwidth and non-volatile memory, bring along heightened access penalties, up to hundreds of nanoseconds. Trying at least to *tolerate* these latencies, hardware manufacturers deploy huge caches and implement sophisticated *hardware prefetching* mechanisms. However, the reliance on hardware alone quickly reaches its limit when non-trivial access patterns cannot be detected by hardware. Navigation through tree and hash data structures are prime examples of this limitation.

Software-based prefetching promises to overcome this situation: Applications can proactively hint to the CPU about upcoming data accesses, which are hard to predict by only monitoring the access stream. Typically, these hints are communicated via specific instructions embedded within modern ISAs such as `x86` and `ARMv8`. Particularly in the context of trees and hash tables, many studies have underscored the significant potential that software prefetching holds (e.g. [5, 9, 12, 18, 21, 22]). However, the discourse often overlooks the incurred costs, specific limitations, and hardware implications across different platforms. We argue that in the future it will be even more important to understand and assess software prefetching in order to optimize data-intensive systems. The trend in Figure 1 backs up our reasoning, demonstrating that the potential of software-based prefetching over random access has increased dramatically over the past few years—and with an eye on the often cited *memory wall*, we expect the trend to continue.

Hence, this survey conducts an in-depth examination of software-based prefetching. We study how the technique is implemented in various systems and turn our insights into guidance for developers seeking to optimize their applications using software prefetching. To that end, Section 2 recaps software prefetching from what we found in the literature. In Section 3 we utilize a micro-benchmark to reveal the true costs and limitations of executing prefetch instructions and show how prefetchers from hard and software can work collaboratively. Section 4 demonstrates that our insights are truly valuable for widely-used data structures. Finally, we summarize in Section 5.

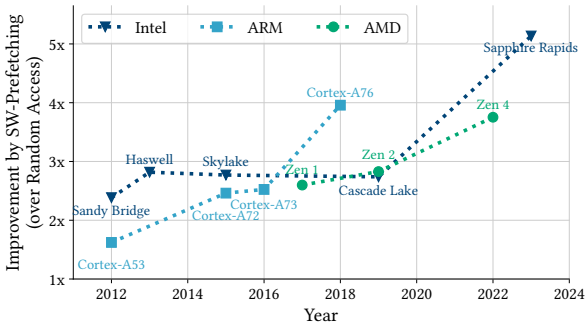


Figure 1: Benchmark accessing individual cache lines—software-prefetched vs. randomly. We see an emerging *trend* developing across different CPU manufacturers and generations: an accelerated performance is achieved by software prefetching.

2 UNDERSTANDING SOFTWARE PREFETCHING

At first glance, the concept of software-based prefetching is seemingly straightforward: data accesses, which are challenging or unfeasible for the hardware to predict, can be communicated to the CPU via specific instructions. Accesses within data structures that suffer from indirect accesses are good examples, such as linked lists, queues, hash tables, and trees. Based on these hints, the underlying hardware substrate can move data *asynchronously* into a cache close to the CPU to minimize the access latency—hiding the *actual* latency behind computational work.

2.1 Utilize Software Prefetching

However, formulating such predictions is often cumbersome. Consider the traversal through a tree-like data structure: The immediate succession of identifying and accessing a node leaves negligible time for the hardware to load the data; even if the software provides a hint about the soon-to-be-accessed tree node. In order to create a (sufficient) temporal gap between pinpointing the next node and accessing it, the literature explores various methods. One approach uses asynchronous control flow abstractions, such as coroutines [9, 12, 22, 27] and fine-grained tasks [21], which facilitate prefetching and computation in parallel. An earlier concept proposes processing operations in groups of multiple items to overlay

the prefetch of one item with the compute of others [5]. In addition, the discussion over the strategic implementation of optimization passes to automatically inject prefetch instructions into compiler-generated code has been ongoing for decades (e.g., [3, 11, 16, 20]).

Not only redesigning algorithms and data structures plays a decisive role in the efficient use of software prefetching, but the instruction must also be issued at the appropriate moment [11]. When software executes the prefetching instruction too early before the actual access, the data may have been already evicted from the cache when needed. Conversely, if prefetched too late, the data might not be transferred to the cache completely, potentially causing the CPU to wait. Accurate prefetch timing necessitates careful understanding of both the execution time of the instructions preceding the actual load and the system's memory latencies [3, 20]. However, the presence of intricate memory systems like NUMA and high-bandwidth memory, as well as the increasing sophistication of CPUs, which now feature out-of-order execution and superscalar architectures, exacerbate the complexity of achieving precise timing. A promising method to address these challenges is the adoption of profile-guided strategies [11].

2.2 Lifecycle of a Prefetch

Almost all modern server and desktop platforms offer several prefetch instructions to target different levels of the cache hierarchy. Notably, each instruction will initiate the transfer for one cache line. The implementation on various hardware systems has only slight variations, even across different vendors. When the software executes a prefetch, the logical address to be prefetched is firstly translated into a physical one. This procedure happens synchronously, i.e., if the translation is not cached within the Translation Lookaside Buffer (TLB), the CPU will contact a Page Miss Handler (PMH) before continuing the execution of the prefetch instruction. We visualize the lifecycle of prefetch requests through the memory subsystem in Figure 2.

Once the physical address is known and the request misses the L1 data cache (L1d), the CPU will initiate the transfer from memory to caches asynchronously. To that end, the L1d cache requests the cache line through the Line Fill Buffer (LFB) (or Miss Address Buffer (MAB) on AMD platforms[1]). As the name implies, the LFB is a buffer-like structure that interfaces between the L1d and L2 cache to communicate requests on a cache-line granularity [4, 24, 25]. For every cache line that misses the L1d, the CPU allocates a slot in the LFB. The L2 cache, on the opposite end, retrieves and delivers the requested lines. This nature makes it straightforward to implement *asynchronous* prefetches: The desired cache line will not cause the instruction to wait until the data is transferred into the cache but only until the miss is communicated to the LFB. Furthermore, the LFB enables the CPU to handle several pending requests simultaneously without blocking a single one, hence supporting out-of-order execution and performing micro-optimizations like merging multiple requests to the same cache line [25]. When the request also misses the L2 cache, it seeks the data from even lower cache levels or main memory. On Intel platforms, the superqueue [13]—a buffer positioned between the L2 and off-core last-level caches (LLCs)—tracks pending requests.

[1]For the sake of simplicity, we only ever refer to the LFB, but also intend the MAB.

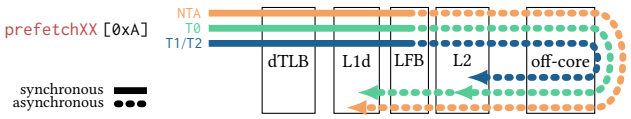


Figure 2: Lifecycle of various prefetch instructions through the memory subsystem.

Upon the completion of a memory request, the specific software prefetch instruction used dictates how close the data is brought to the CPU. In the `x86` and `ARM` ISAs, most prefetch instructions clearly specify the cache level target for the fetched data: Cache lines fetched by `prefetcht1` (and its counterpart `pldl2keep` on ARM) are stored in the L2 cache; `prefetcht0` (`pldl1keep` on ARM) moves data further into the L1d, thereby passing through the LFB. The `prefetcht2` instruction is generally aimed to fetch data into the LLC. However, recent Intel platforms (since Skylake) implement the LLC as a non-inclusive victim cache—here, the data is placed into the L2 instead. Conversely, for the *non-temporal* `prefetchnta` instruction, the `x86` ISA does not specify a particular cache target but only the objective: reducing cache pollution for data that will be accessed nonrecurring.

AMD's documentation for earlier processor generations (e.g., Bulldozer [7]) specifies that the `prefetchnta` pulls data into the L1d and ensures it is not evicted to the L2 cache unless it originated from there. For later generations (i.e., for Zen), the documentation no longer mentions the exact cache destination but merely states that the L2 is bypassed when evicting non-temporal data. Intel, in contrast, specifies also for modern plattforms that non-temporal prefetches move data into the L1d and into the LLC, if the LLC is inclusive [10]. Importantly, both Intel and AMD note that these non-temporally prefetched cache lines are prioritized for quicker eviction [8, 10].

2.3 Limitations

Modern platforms exhibit two notable limitations in the implementation of software prefetches. First, the execution of a single prefetch instruction stalls until the virtual address is translated into a physical address. Although the TLB might accelerate this process, it must be assumed that most prefetched addresses are not present in the TLB as software prefetching is primarily used when the application accesses scattered data objects. Consequently, the latency of the prefetch instruction is dominated by the latency of the PMH performing a page-table walk to translate the address. However, this restriction only applies to the initial prefetch operation within a memory page, whenever several prefetches are performed consecutively to load a block larger than a single cache line. While hardware indeed implements also prefetchers for address translations [26], the software is only capable of preloading data.

Drawing from that obersavtion, it seems advantageous to apply prefetching to larger data regions by executing multiple instructions. However, this might trigger a secondary limitation: Due to the limited number of LFB entries (typically between 3 and 24), the hardware can accommodate only a small number of outstanding memory requests. If an excessive volume of prefetch requests is

sent out rapidly, the LFB becomes filled and cannot accept further requests. Under this circumstance, the CPU will either stall until an LFB slot becomes ready or—according to Intel's documentation [10]—drop new prefetch requests.

3 SOFTWARE PREFETCHES UNDER THE MAGNIFYING GLASS

To our surprise, these limits have only been studied in depth to a limited extent, although the technique is put to use by several applications in both research (e.g., [5, 9, 12, 18, 19, 22]) and "the real world" (e.g., *TCMalloc*). More than a decade ago, Lee et al. investigated both hardware and software prefetching in the context of widely used applications [14]. In this chapter, we aim to expand upon their foundational research, delving deeper into the nuances of software prefetching. We carefully analyze the costs and implications involved on modern platforms and further examine the synergies between software prefetching and modern hardware prefetchers.

3.1 Micro-Benchmark

To that end, we utilize a micro-benchmark that aligns with conventional latency and throughput benchmarks: We employ two arrays, one array dictating the order in which the second data array is accessed. The data within this second array is organized as blocks; for different experiments we vary the size of the blocks from one to several cache lines. Each block's data is accessed sequentially. To introduce work behind which we can hide memory latencies with prefetching, we split each cache line into an array of 32-bit integers and compute hash values (a total of 568 instructions per cache line). While we will evaluate a number of different hardware platforms to ensure a broad understanding, we will primarily focus on an Intel Xeon Gold 6226 CPU (Cascade Lake) for illustrative purposes. On this specific platform, the micro-benchmark's raw processing time for a single cache line is approximately 193 cycles. All the following experiments are single-threaded.

3.2 Quantifying the Costs

So far, the precise costs associated with injecting additional prefetch instructions remain undetermined. Our examination identifies three principal cost factors incurred during execution: the CPU's retirement of the instruction, the translation of the address to be prefetched into a physical one, and the allocation of a slot in LFB. Pinpointing the exact source of costs caused by the prefetch instruction remains an intricate challenge. To gain a deeper understanding, we assess the difference in compute cycles between a sequential execution and one that accesses data randomly but uses software prefetching. Thanks to the efficiency of the hardware's data and TLB prefetchers, the sequential execution encounters almost no stalls, thereby revealing the pure computational cycles of our workload.

Execution and Address Translation. For our first experiment, we configure our micro-benchmark to use blocks of only one cache line, accessing all cache lines in random order. By utilizing performance counters, we can categorize the consumed cycles into compute-related and stalled cycles. Furthermore, we can dissect the stalled

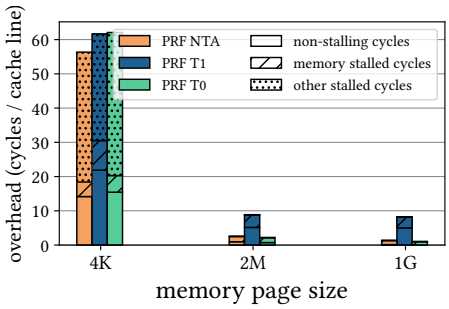


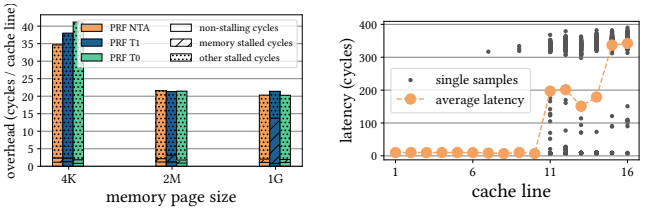
Figure 3: Breakdown of CPU cycles associated with a single prefetch for different memory page sizes and instructions.

cycles into two subsets: those occurring due to memory waiting times (using the `CYCLE_ACTIVITY.STALLS_MEM_ANY` counter) and "other" stalls (tracked via `CYCLE_ACTIVITY.STALLS_TOTAL`[2]). Figure 3 presents the results, showing the cycle *overhead* (beyond the sequential execution) while prefetching single cache lines by software using different memory page sizes. With "common" 4 kB-sized pages, we can observe that a single prefetch induces an additional approx. 60 CPU cycles, most of them related to stalls of unidentifiable sources.

Remarkably, the major amount of these additional cycles diminishes when utilizing Hugepages, simultaneously reducing the number of TLB misses. This observation leads us to deduce that these cycles are primarily caused by stalls while translating the address to prefetch, which is in line with further TLB-focused studies [17]. Only the `prefetcht1` instruction, which directs data into the L2 cache, incurs minimal overhead of up to 9 cycles, even when Hugepages are used. However, the translation (and its related costs) is somewhat inevitable and occurs throughout workloads with random and scattered access patterns; it is only moved forward in time by the execution of the prefetch instruction (and thus associated with the prefetch instead of the actual data access). The reason for stalling lies in the handling of TLB misses during the prefetch execution, which is always processed *synchronous*—compelling the CPU to wait while the PMH takes action. Conversely, the actual data cache misses are processed *asynchronously* with the assistance of the LFB. Note that also memory stalls experience a slight decrease using Hugepages since the PMH can also cause data cache misses.

While the precise categorization of cycles is only possible on a few CPU architectures, we see similar patterns on different CPUs across different hardware manufactures, like Intel, AMD, and ARM, and architectures (in fact, across all CPUs listed in Figure 1). For instance, on an Intel platform from the Sandy Bridge series, we note a comparable overhead using 4 kB memory pages and only 3 cycles of overhead with the adoption of Hugepages. Conversely, on contemporary AMD platforms (Zen 4), the observed overhead is around 40 cycles for 4 kB pages and ranges from 7 (for `prefetchnta` and `prefetcht0` instructions) to 14 cycles (for `prefetcht1`) when utilizing Hugepages.

[2]Note that the `STALLS_TOTAL` number already includes memory stalls; thus, it is necessary to subtract the latter.



(a) Overhead in cycles

(b) Sampled prefetch-latencies for 1 GB Hugepages

Figure 4: Results of the micro-benchmark using 16 cache lines per block.

Line Fill Buffer. Prefetches—and loads in general—missing the L1d are forwarded via the LFB to the lower memory subsystem. The capacity of these buffers is quite limited: Intel's architectures provide 10 (e.g., Sandy Bridge and Haswell [10]) to 16 (e.g., Golden Cove [10]) LFB slots, while modern AMD CPUs (like Zen 4) offer up to 24 MAB entries. When the LFB is at full capacity, no new prefetch or load requests can be accepted, causing the CPU to stall until slots become available as previous requests are completed by the L2 cache. Since the software-prefetch instruction allocates a slot *synchronously*, prefetching large data blocks at once (e.g., an entire tree node) can exceed the buffer's capacity, leading to increased latency when issuing the instruction.

However, precisely breaking down the overhead for LFB-related stalls is not as straightforward as for address translation-related costs since these stalls only occur for requests that flood the buffer—thus, not all prefetches are impacted equally. To still get an impression, Figure 4 shows the results of a scenario where our micro-benchmark prefetches blocks of 16 cache lines at once. More precisely, Figure 4a shows the average additional cycles per cache line in such an execution over a sequential one (analogously to the comparison for single-line blocks in Figure 3). With 4 kB memory pages, the illustrated overhead includes both TLB miss penalties and LFB-related stalls, ranging around 35 additional cycles per prefetch. However, utilizing 1 GB Hugepages eliminates almost all translation-related stalls, leaving approx. 20 stalled cycles per cache line, which we associate with the waiting times for an available LFB slot. To back this further, we also evaluated the number of requests finding an already full LFB (via the counter `L1D_PEND_MISS.FB_FULL`), which increases from almost zero (when prefetching a single cache line) to 1.2 per cache line when prefetching a block of 16 cache lines.

Again, the stalled cycles demonstrated in Figure 4a are an average over all cache lines of an entire block; but not all prefetches experience identical costs. Delving deeper, Figure 4b dissects the mean *latency* in cycles for each prefetched cache line within a block when utilizing 1 GB Hugepases. We recorded these results using address sampling via the *perf* interface [1]. Notably, there is a marked latency escalation starting from the eleventh cache line, where the average latency values leap from 7 to a range of 200 – 340 cycles. This pattern suggests that the underlying Cascade Lake architecture has exactly 10 LFB slots, which is in line with (unofficially published [2]) numbers for the very similar Skylake architecture. Furthermore, we verified these observations on Intel Sandy Bridge and Haswell systems, which officially have 10 LFB slots, showcasing similar latency patterns as depicted in Figure 4b. Similarly to the previous benchmarks, we can confirm results comparable to Figure 4a on AMD's Zen 4, which does not enable cycle breakdowns, by utilizing the "common" cycle counter. However, evaluating the latter hardware necessitates prefetching 32 cache lines to notice a similar latency pattern, reflecting the MAB's larger capacity of 24 slots.

Discussion. Based on our findings, we conclude that the execution of prefetch instructions causes only negligible pressure on the CPU's instruction bandwidth. Most of the costs associated with a single prefetch instruction are attributable to the (unavoidable) penalties of TLB misses. Mitigating these costs demands hardware-level support, such as adopting larger memory page sizes. Looking ahead, an ideal advancement would be the implementation of asynchronous translation mechanisms, allowing prefetch operations to proceed without stalling. Additionally, the introduction of a specialized instruction to prefetch address translations could significantly streamline the process. To circumvent expensive LFB stalls (see Figure 4), prefetching algorithms have to be implemented very carefully (see details in Section 4 below). It is essential to consider the LFB's capacity, which not only varies across hardware manufacturers but also between different generations—requiring *hardware conscious* implementations. For instance, AMD's architectures use up to 24 slots, whereas the latest Intel architectures support up to 16.

3.3 Interplay with Hardware Prefetchers

We saw that software prefetching has its limits when it is necessary to prefetch large data blocks. To circumvent this limitations and associated costs, a collaboration between software and hardware prefetching emerges as a promising solution, forming a symbiotic relationship: Software prefetching takes the initial step by early hinting about unpredictable data accesses; hardware prefetchers can take over, incrementally fetching the remaining data line by line. In this section, we delve into their cooperative dynamics, emphasizing how strategic software prefetching *trains* hardware prefetchers beyond the typical dependence on load misses. This could enhance the efficiency of range scans in tree structures, for instance.

Hardware Prefetcher Patterns. In our in-depth analysis, we concentrate on Intel processors, which offer valuable insights into prefetching dynamics by facilitating the sampling of data addresses and access latencies through the Linux *perf* interface [1]. Intel implements multiple core-local prefetchers that serve the L1 and L2 caches across both modern and legacy processor generations [10, 23]. The L1d prefetchers monitor load streams within a single cache line, subsequently requesting the next line. Meanwhile, L2 prefetchers study the patterns of cache line requests traversing through the L2 cache. Here, the *stream prefetcher* initiates prefetches based on a sequence of consecutive cache line requests, while the *adjacent prefetcher* consistently loads a pair of neighboring cache lines at once. This architectural design prompts two critical lines of inquiry: Do software and hardware prefetchers have any interaction? If yes, what are the impacts of different prefetch instructions?

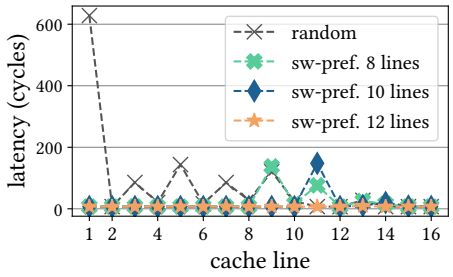


Figure 5: Prefetching several cache lines by software to explore the hardware/software prefetching interaction.

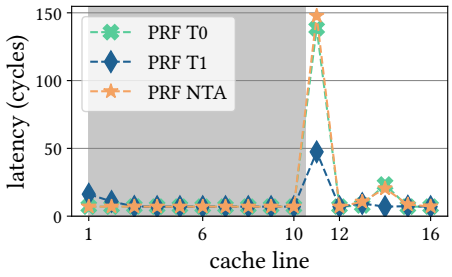


Figure 6: Access latencies when prefetching 10 of 16 cache lines using different prefetch instructions.

Software Prefetches Train the Hardware. Let us revisit the scenario presented in Section 3.2: When software-prefetching is used for randomly accessed data blocks that are 16 cache lines in size, we observed that the smaller-sized LFB becomes overwhelmed. Interestingly, our experiments reveal a compelling interaction wherein software-based prefetching actively trains hardware prefetchers when injecting prefetch instructions only for the first subset of the needed cache lines.

Figure 5 illustrates the access latency for each cache line within a block when varying the quantity of prefetched cache lines: 8, 10, and 12 lines are prefetched using software (with `prefetchnta` for illustrative purposes), leaving the hardware prefetchers to manage the remainder. To contextualize the results, we also present a scenario without software prefetching ("random"), where we observe the hardware prefetchers quickly adapt to the sequential access pattern within a block. However, the hardware prefetchers are initially too late for every second cache line and cannot hide the entire latency. This might be attributed to the small workload (around 193 CPU cycles per cache line) of our micro benchmark.

We can recognize a similar pattern when we software-prefetch only the first 8 cache lines. While the software-prefetched cache lines exhibit L1d-related latencies, we observe a noticeable latency spike in the first cache line not prefetched by software. It takes more accesses—up to the 12th cache line—for the hardware to prefetch the remaining data timely. Remarkably, the configuration where 12 cache lines are prefetched by software emerges as the best performing one. Given that this amount appears to be sufficient for the hardware prefetchers to operate on time, it strikes the balance between minimizing LFB pressure and reducing access latency in our benchmark. While the results indicate that with doing more computational work, fewer software prefetches are sufficient, the primary conclusion remains: Software prefetches trigger the hardware prefetchers. We notice a comparable trend for AMD's Zen 4 platform: The throughput reaches a plateau when software-prefetching only 6 cache lines, indicating that the hardware prefetchers are successfully taking over.

Different Instructions Train Differently. However, not all software prefetch instructions train hardware prefetchers the same way. Our analysis of different instructions reveals that prefetching into the L2 cache (via `prefetcht1`) tends to minimize latency penalties, especially when prefetching fewer than 12 cache lines. Figure 6 illustrates an evaluation where the initial 10 cache lines of a block are prefetched using various instructions, highlighting the differential impacts on latency.

Naturally, hardware prefetchers respond to the access patterns they encounter; prefetching directly into the L1d cache (using `prefetchnta` and `prefetcht0`) circumvents requests to the L2 cache—preventing it from *learning* from these access patterns. Consequently, the L2 cache is only faced with misses when the first non-software-prefetched cache lines are touched. This becomes even more evident when the L1d hardware prefetchers are turned off.

Furthermore, Figure 6 demonstrates a rapid convergence of latencies for data prefetched into the L2 to those associated with L1d accesses. This occurs as a result of the L1d prefetchers, which detect cache misses and subsequently fetch the relevant cache lines from the L2 to the L1d cache. While address sampling is not applicable for AMD architectures, performance counters (in particular `LS_HW_PF_DC_FILLS.LOCAL_L2`) indicate a very similar behavior.

4 ENHANCED APPROACHES TO SOFTWARE PREFETCHING

Based on our findings, it becomes evident that the LFB's limited capacity presents a new bottleneck when prefetching is the software-side response to tolerate memory latency. Given that it is already profitably implemented for tree-like data structures [12, 18, 21, 22], it is somewhat surprising that these LFB limitations have not been extensively studied in such contexts. When baking software prefetching into tree traversal algorithms, the size of the tree nodes becomes a critical factor that directly influences effectiveness. Historically, the node sizes of index structures have often been aligned with the block size of the underlying storage medium, such as HDDs or SSDs. But even in in-memory databases, where the node size does not have to be aligned with a certain storage medium, many tree-based index structures still use relatively large node sizes. One of the reasons is that trees with large nodes have a comparatively smaller depth, which in turn is associated with less time spent on random accesses, including the address translations during a tree traversal. However, when software prefetching is applied without considering the characteristics of the underlying hardware, these large node sizes can quickly become a challenging problem, since fully prefetching a tree node may exceed the capacity of the LFBs within a CPU core as already highlighted in Section 3.1.

To illustrate the impact of node sizes on software prefetching we examined a state-of-the-art B$^+$-tree with Optimistic Lock Coupling as proposed by Leis et al. [15]. We extended the tree to implement prefetching with coroutines similar to the approaches proposed in [12, 22]. The concept of coroutines allows a function to be suspended at a specific point and resumed later, which allows for *interleaved* execution; creating a time frame between the identification and access of a node. We modified the tree so that during a traversal a software prefetch that loads data in all caches (`prefetcht0`) is issued for the succeding node and the traversal is suspended afterward. If a coroutine suspends, the coroutine scheduler can start the execution of another coroutine.

For our experiments, we executed 50M lookups with different node sizes in the original (hereinafter only referred to as *B$^+$-tree*) and the modified version of the B$^+$-tree (*Coro-tree*) and measured the execution in cycles per lookup. We used the YCSB benchmark [6] to fill the tree with 50 M entries (8 bytes for key and value each). As our primary test platform, we used a machine with Intel's Cascade Lake architecture (see section 3.1).

For single-threaded execution, our results showed that a node size of 4 kB performs best for the *B$^+$-tree*, compared to smaller node sizes (1 kB, 256 B), which we also use as a baseline for the *Coro-tree*. In the *Coro-tree*, however, we could observe the exact opposite. Here, a node size of 256 bytes performed best and outperformed the best-performing *B$^+$-tree* by a factor of 2.25, while we saw less performance with bigger node sizes. While a node size of 1 kB still experiences a performance boost by factor 1.51, a node size of 4 kB performs worse compared to the baseline (factor 0.8). This behavior can be explained by the amount of cache lines that are needed for every node. While a node with a size of 256 B consists of 4 cache lines, a node size of 4 kB results in 64 cache lines, which exceeds the amount of 10 LFB on our primary test platform. As depicted in Figure 7, we could observe a very similar behavior on older Intel architectures, on Intel's latest Sapphire Rapids architecture however, a node size of 1 kB has only a small performance loss compared to a node size of 256 bytes. We attribute both observations to the LFB's capacity since the observed older architectures (Sandybridge and Haswell) also have 10 LFB entries. On the latest Sapphire Rapids platform, the LFB was increased to 16 slots. On recent AMD architectures, we could also see great performance improvements (up to 2.32 on Zen 4). In contrast to the surveyed Intel platforms, it is remarkable that the *Coro-tree* always outperforms the *B$^+$-tree*, even with a node size of 4 kB. We also relate this mainly to the number of fill buffer entries on AMD architectures, which is larger than on Intel platforms (e.g., 24 entries for Zen 4).

Simultaneous Multithreading. Nearly every modern `x86` architecture implements simultaneous multithreading (SMT) to better utilizing the available resources of the CPU. While some CPU components are duplicated (e.g., some registers), other components, like caches, are shared between the threads. As already observed in the previous sections, the LFB represents a critical resource when it comes to utilize software prefetches beneficially; the use of SMT is expected to bring this scarce resource even more into focus. Therefore, we executed our experiments also with two logical threads that run on the same physical core to measure the impact of this critical

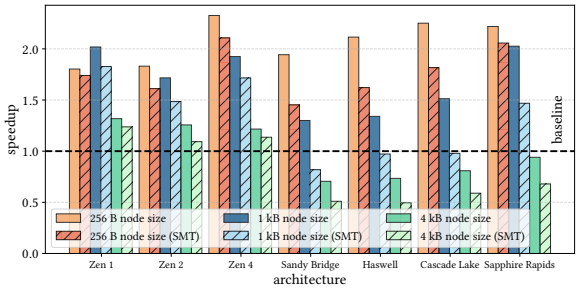


Figure 7: Speedup of a single lookup with different node sizes with and without SMT compared to the best unprefetched baseline.

resource. On our primary test platform, we could still see a performance improvement (factor 1.8) with a node size of 256 bytes, even if the amount of cycles for executing one lookup operation increased, but this accounts also for the *B$^+$-tree* with SMT, that was used as a baseline. Here, we attribute the increased number of cycles primarily to the type of workload we execute, as we do roughly the same kind of work in both threads and, therefore, might use the same shared resources. With a node size of 1 kB, however, we could see a noticeable degradation in performance (factor 0.98) when using SMT on our primary test system. In this case, both threads have to prefetch 16 cache lines, which massively exceeds the number of the available 10 LFB entries. Similar to the execution with a single thread, we could observe this effect as well on older Intel architectures like Sandy Bridge or Haswell. On recent AMD architectures we could also see an increase in cycles per lookup when executing two threads on the same physical core, but in contrast to the Intel, even prefetching of 4 kB nodes performed better than the baseline. Our findings indicate that contemporary AMD architectures are less vulnerable to prefetching larger regions, even when using SMT—in contrast to Intel designs.

The results from our study of the coroutine-based *B$^+$-tree* demonstrate that the LFB can indeed become a limitation of software prefetching, also within real-world applications. Especially the use of SMT, which results in sharing this already scarce resource between two (logical) cores, intensifies the competition and potential for stalls when prefetching too large tree nodes. The evaluation shows—once again—the importance of baking the hardware's characteristics into algorithms to achieve optimal performance.

5 SUMMARY

Software-based prefetching holds enormous potential when seeking to hide memory latency behind valuable CPU cycles. In this survey, we took a closer look at the benefits and limitations of software prefetching on several hardware platforms. And we found that its performance depends dramatically on the developer's knowledge of the underlying hardware substrate. Evaluations on synthetic and real-world workloads identified the number of pending memory requests a system can manage as the "next" bottleneck. To leverage the hardware to full performance, engineers need to exercise great caution while designing software-prefetched algorithms.

REFERENCES

[1] [n. d.]. perf_event_open(2) - Linux Manual Page. https://man7.org/linux/man-pages/man2/perf_event_open.2.html. Online; last accessed March 20, 2024.

[2] [n. d.]. Skylake: Intel's Longest Serving Architecture. https://chipsandcheese.com/2022/10/14/skylake-intels-longest-serving-architecture/. Online; last accessed March 22, 2024.

[3] Sam Ainsworth and Timothy M. Jones. 2017. Software prefetching for indirect memory accesses. In *Proceedings of the 2017 International Symposium on Code Generationand Optimization*. ACM, 305–317. http://dl.acm.org/citation.cfm?id=3049865

[4] Pietro Borrello, Andreas Kogler, Martin Schwarzl, Moritz Lipp, Daniel Gruss, and Michael Schwarz. 2022. ÆPIC Leak: Architecturally Leaking Uninitialized Data from the Microarchitecture. In *31st USENIX Security Symposium*. USENIX Association, 3917–3934. https://www.usenix.org/conference/usenixsecurity22/presentation/borrello

[5] Shimin Chen, Anastassia Ailamaki, Phillip B. Gibbons, and Todd C. Mowry. 2004. Improving Hash Join Performance through Prefetching. In *Proceedings of the 20th International Conference on Data Engineering, ICDE*. 116–127. https://doi.org/10.1109/ICDE.2004.1319989

[6] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.

[7] Advanced Micro Devices. 2014. Software Optimization Guide for AMD Family 15h Processors. https://www.amd.com/content/dam/amd/en/documents/archived-tech-docs/software-optimization-guides/47414_15h_sw_opt_guide.pdf. Online; last accessed March 20, 2024.

[8] Advanced Micro Devices. 2020. Software Optimization Guide for AMD EPYC™ 7003 Processors. https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/software-optimization-guides/56665.zip. Online; last accessed March 20, 2024.

[9] Yongjun He, iacheng Lu, and Tianzheng Wang. 2020. CoroBase: Coroutine-Oriented Main-Memory Database Engine. *Proc. VLDB Endow.* 14, 3 (2020), 431–444. https://doi.org/10.5555/3430915.3442440

[10] Intel. 2024. Intel® 64 and IA-32 Architectures Optimization Reference Manual. https://cdrdv2.intel.com/v1/dl/getContent/671488. Online; last accessed March 20, 2024.

[11] Saba Jamilan, Tanvir Ahmed Khan, Grant Ayers, Baris Kasikci, and Heiner Litz. 2022. APT-GET: profile-guided *timely* software prefetching. In *EuroSys '22: Seventeenth European Conference on Computer Systems*. ACM, 747–764. https://doi.org/10.1145/3492321.3519583

[12] Christopher Jonathan, Umar Farooq Minhas, James Hunter, Justin J. Levandoski, and Gor V. Nishanov. 2018. Exploiting Coroutines to Attack the "Killer Nanoseconds". *Proc. VLDB Endow.* 11, 11 (2018), 1702–1714. https://doi.org/10.14778/3236187.3236216

[13] Tsvika Kurts, Zelig Wayner, and Tommy Bojan. 2005. Apparatus and method for bus signal termination compensation during detected quiet cycle. US Patent 6,842,035.

[14] Jaekyu Lee, Hyesoon Kim, and Richard W. Vuduc. 2012. When Prefetching Works, When It Doesn't, and Why. *ACM Trans. Archit. Code Optim.* 9 (2012), 2:1–2:29. https://doi.org/10.1145/2133382.2133384

[15] Viktor Leis, Michael Haubenschild, and Thomas Neumann. 2019. Optimistic Lock Coupling: A Scalable and Efficient General-Purpose Synchronization Method. *IEEE Data Eng. Bull.* 42, 1 (2019), 73–84.

[16] Chi-Keung Luk and Todd C. Mowry. 1996. Compiler-Based Prefetching for Recursive Data Structures. In *ASPLOS-VII Proceedings - Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM Press, 222–233. https://doi.org/10.1145/237090.237190

[17] Daniel Lustig, Abhishek Bhattacharjee, and Margaret Martonosi. 2013. TLB Improvements for Chip Multiprocessors: Inter-Core Cooperative Prefetchers and Shared Last-Level TLBs. *ACM Trans. Archit. Code Optim.* 10, 1 (2013), 2:1–2:38. https://doi.org/10.1145/2445572.2445574

[18] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *European Conference on Computer Systems, Proceedings of the Seventh EuroSys Conference 2012*. ACM, 183–196. https://doi.org/10.1145/2168836.2168855

[19] Prashanth Menon, Andrew Pavlo, and Todd C. Mowry. 2017. Relaxed Operator Fusion for In-Memory Databases: Making Compilation, Vectorization, and Prefetching Work Together At Last. *Proc. VLDB Endow.* 11, 1 (2017), 1–13. https://doi.org/10.14778/3151113.3151114

[20] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. 1992. Design and Evaluation of a Compiler Algorithm for Prefetching. In *ASPLOS-V Proceedings - Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM Press, 62–73. https://doi.org/10.1145/143365.143488

[21] Jan Mühlig and Jens Teubner. 2021. MxTasks: How to Make Efficient Synchronization and Prefetching Easy. In *Proceedings of the 2021 International Conference on Management of Data, SIGMOD*. ACM, 1331–1344. https://doi.org/10.1145/3448016.3457268

[22] Georgios Psaropoulos, Thomas Legler, Norman May, and Anastasia Ailamaki. 2017. Interleaving with Coroutines: A Practical Approach for Robust Index Joins. *Proc. VLDB Endow.* 11, 2 (2017), 230–242. https://doi.org/10.14778/3149193.3149202

[23] Till Schlüter, Lorenz Hetterich, Leon Trampert, Hamed Nemati, Ahmad Ibrahim, Michael Schwarz, Christian Rossow, and Nils Ole Tippenhauer. 2023. FetchBench: Systematic Identification and Characterization of Proprietary Prefetchers. In *Proceedings of the 2023 ACM SIGSAC*. ACM, 975–989. https://doi.org/10.1145/3576915.3623124

[24] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 753–768. https://doi.org/10.1145/3319535.3354252

[25] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2019. RIDL: Rogue In-Flight Data Load. In *IEEE Symposium on Security and Privacy*. IEEE, 88–105. https://doi.org/10.1109/SP.2019.00087

[26] Georgios Vavouliotis, Lluc Alvarez, Vasileios Karakostas, Konstantinos Nikas, Nectarios Koziris, Daniel A. Jiménez, and Marc Casas. 2021. Exploiting Page Table Locality for Agile TLB Prefetching. In *48th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA*. IEEE, 85–98. https://doi.org/10.1109/ISCA52012.2021.00016

[27] Xiangyu Zhi, Xiao Yan, Bo Tang, Ziyao Yin, Yanchao Zhu, and Minqi Zhou. 2024. CoroGraph: Bridging Cache Efficiency and Work Efficiency for Graph Algorithm Execution. *Proc. VLDB Endow.* 17, 4 (2024). https://doi.org/10.14778/3636218.3636240