Understanding Application Performance on Modern Hardware: Profiling Foundations and Advanced Techniques

Jan Mühlig [1], Roland Kühn [1], and Jens Teubner [1]

Abstract: Engineering performance-sensitive applications necessitates a deep comprehension of the interactions between hardware and software. Although profiling tools are available to assist, they often struggle to precisely analyze specific segments of an application. And, crucial information, such as data object addresses, is challenging to relay from the application to external tools. This tutorial demonstrates how these challenges can be addressed by using libraries that enable performance profiling directly within the application.

1 Introduction

Understanding the interplay between software and hardware is paramount for optimizing the performance of data-intensive applications, such as database management systems (DBMSs). However, the architecture of modern hardware has grown increasingly complex—including vast cache hierarchies, sophisticated memory systems, and advanced CPU cores equipped with features like out-of-order execution, intricate branch prediction algorithms, and simultaneous multi threading. This complexity introduces substantial challenges for engineers trying to decipher the interaction of hardware and software, making *performance profiling* a critical practice to pinpoint and—ideally—mitigate performance bottlenecks.

A variety of tools, such as Intel *VTune* [In24b], AMD *μProf* [AM24], and Linux *Perf* [DM10], support engineers in analyzing execution performance. Under the hood, these tools utilize *performance counters*: dedicated registers baked into nearly all modern processors that monitor low-level hardware events, such as the count of executed instructions or cache hits and misses. By correlating these events with instruction pointers, the tools can help to diagnose resource-intensive lines of code.

However, Linux Perf and suchlike—being external—typically profile the entire application, complicating the targeted profiling of specific application segments. This limitation is particularly problematic when analyzing micro-benchmarks, where the benchmark itself may represent only a fraction of the overall runtime, or when attempting to distinguish between different operational phases, such as the fill phase versus the lookup phase in, for instance, tree benchmarks. Additionally, while these tools can correlate events with higher-level constructs like code lines using the binary's symbol names, they often miss the

[1] DBIS Group, TU Dortmund University, Dortmund, Germany,
jan.muehlig@tu-dortmund.de, https://orcid.org/0009-0008-2226-6367;
roland.kuehn@cs.tu-dortmund.de, https://orcid.org/0000-0002-1485-140X;
jens.teubner@cs.tu-dortmund.de, https://orcid.org/0000-0002-0344-5203

opportunity to leverage deeper, application-specific insights, such as the memory location of data structure instances [No20]. Consequently, while it is possible to identify instructions exhibiting high memory access latency, linking them to specific instances of data structures remains elusive. Consider tree-like data structures as an example: Different tree levels experience unique access patterns; nodes logically close to the root tend to stay in the CPU cache, whereas nodes close to the leaves, accessed less frequently, are more likely to be evicted. Since nodes are typically accessed using the same code, it becomes challenging to pinpoint the *true* source of high access latency, which is primarily the node instance rather than the line of code.

In this tutorial, we demonstrate how to address these challenges by integrating profiling directly into the application. This integration allows for fine-grained profiling and the injection of application-specific insights into the profiling results, thereby enhancing our understanding of the interactions between software and hardware. The tutorial is structured as follows: Section 2 provides background on performance profiling techniques. Section 3 illustrates how applications can manage hardware events, using a B$^+$-tree as a running example. Finally, Section 4 outlines the timeline and structure of this tutorial.

2 Using Performance Counters to Understand Hardware/Software Interaction

Modern processors provide two different methods to gain an understanding of how software interacts with the underlying hardware substrate. First, Performance Monitoring Units (PMUs) can count the occurrence of low-level hardware events like cache misses throughout the software's execution cycle. Second, processors can periodically capture snapshots that offer more fine-grained data, such as the currently executing instruction and the accessed memory address.

2.1 Counting Hardware Events

For counting event statistics, PMUs are deployed in a timer-like fashion: They are programmed with specific events to count, started before the execution of code to monitor, and stopped subsequently. The difference in recorded values before and after the execution reflects the events' occurrence during the software's operational phase. This practice of recording key performance indicators is both well-known and extensively used in professional system engineering and academic research (e.g., [LLS13, ZF15, Be21, Sc23, Kü23, KMT24]).

While modern hardware provides a handful of PMUs that can record events simultaneously, the spectrum of monitorable events has broadened significantly across recent processor generations. For example, Intel's Sandy Bridge architecture (launched in 2011) supports approximately 1 900 distinct events, a number that has risen to over 16 000 with the introduction of the Cascade Lake architecture in 2019. Notably, most of these events cover *uncore* activities. Nevertheless, the diversity of these events varies across different CPU generations and manufacturers, adding another layer of difficulty in comparing system performance across chipmakers.

2.2 Detailed Analysis through Sampling

Although coarse-grained statistics can highlight inefficient executions, they seldom pinpoint the exact origin of bottlenecks, such as which specific instruction causes cache misses or branch mispredictions. For a more granular analysis, modern processors are equipped with *sampling*-based mechanisms that periodically capture the current state of execution. The general idea is to specify one or multiple *trigger* events along with a threshold; the PMU will then count the trigger event and fire a sample upon reaching the threshold, for instance, every 4 000th CPU cycle. Users can also dictate the specifics included in the sample, such as the pointer of the currently executing instruction, the accessed memory address, details of the access like latency and data origin, register values, and even branch and call stacks.

The most common method of utilizing these samples is *code-based* profiling, implemented by various profiling tools such as *VTune*, AMD *μProf*, and Linux *Perf* (particularly through the Perf *record* subcommand). These tools focus on monitoring instruction pointers and correlating them with higher-level programming constructs, e.g., C++ functions and lines of code. This enables the identification of code that is executed frequently or consumes significant execution time. The derived insights can pinpoint specific sections of code that are critical bottlenecks, such as those causing CPU stalls while waiting for data transfers from the memory subsystem.

2.3 Abstractions

Different CPU manufacturers implement sampling mechanisms in distinct ways [Sa23]. For instance, since the Nehalem architecture, Intel implements Processor Event-Based Sampling (PEBS) [In24a, AH17, Yi20], which allows the PMU to leverage almost any configurable event as a trigger for sampling. When the specified threshold is reached, the CPU captures a snapshot and stores the sampled data into a dedicated hardware-managed buffer. As this buffer fills, the CPU interrupts the kernel to transfer the samples into a user-level buffer for further analysis.

AMD's Instruction-Based Sampling (IBS) [Dr07, Dr10] works differently: Each CPU core has two specific PMUs that can be used to sample either *instruction fetching* or *execution* events—the latter based on executed CPU cycles or micro-operations [Sa23]. To that end, the CPU counts either fetched instructions or executed cycles/micro-operations until the threshold is met. Upon reaching this threshold, the CPU traces the next instruction through

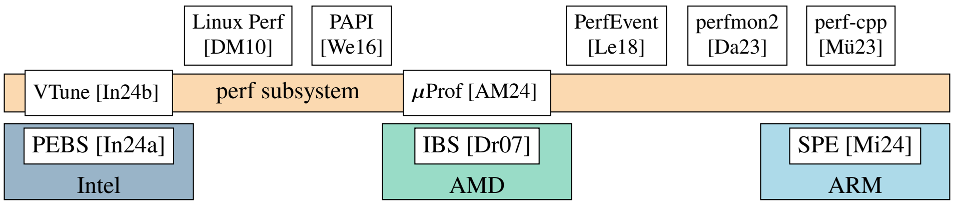


Fig. 1: Abstraction of different profiling approaches. The *perf subsystem* generalizes the utilization of both across different CPUs and manufacturers, enabling several libraries to control PMUs.

the entire fetch or execution pipeline, allowing for the collection of diverse data types, such as TLB statistics from instruction fetches or memory-based statistics from execution monitoring. Since the ARMv8.2 ISA extension, ARM implements Statistical Profiling Extension (SPE), a sampling procedure similar to Intel PEBS and AMD IBS [Mi24]. However, SPE is an optional feature, usually only implemented in high-performance cores like the Neoverse N1 or the Cortex-A78.

These methodological differences extend to the configuration of the hardware and interpretation of results. Hence, manufacturers typically develop tailored profiling tools, such as Intel's VTune and AMD's μProf. To bridge these variations, the Linux kernel offers a unified solution through the *perf subsystem*, providing a standardized interface for recording statistics and sampling across various CPUs and manufacturers. This subsystem is deeply integrated within the kernel architecture and forms the foundation of the Linux *Perf tool*, which is designed to handle both the recording of performance statistics and detailed event sampling. Figure 1 illustrates the abstraction levels corresponding to various manufacturer-specific examples.

3 Fine-granular Performance Monitoring

The mentioned instruments primarily wrap the counting of events and the recording of samples throughout an application's entire execution. This broad approach limits their effectiveness in profiling, for example, counting cache misses only within a specific code segment.

3.1 Phase-wise Event Counting

While the perf subsystem allows controlling PMUs from the application, its interface is notably complex and can be challenging to utilize effectively. As a response to these challenges, recent years have seen the development of several more generalized abstractions on top of the perf subsystem to record performance counters and samples, e.g., [We16, Da23, Le18, Mü23] (see Figure 1).

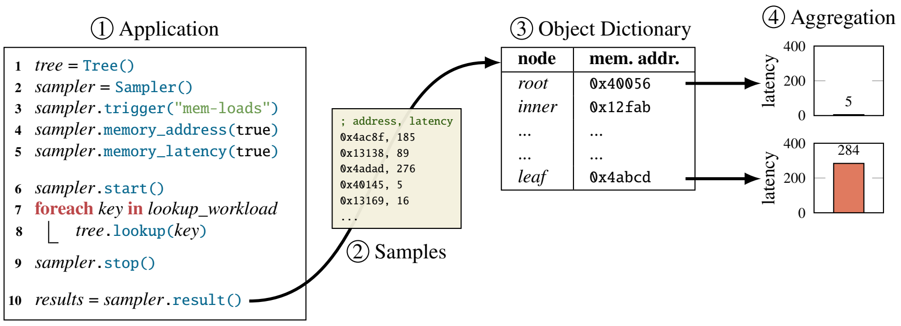
```
1  tree = Tree()
2  event_counter = EventCounter()
3  event_counter.add("cycles","instructions","cache-misses")

4  insert_workload = generate_insert_workload()      // Generate workload data for inserts
5  lookup_workload = generate_lookup_workload()                           // and lookups

6  foreach tuple in insert_workload
7      tree.insert(tuple)                                             // Populate the tree

8  event_counter.start()               // Wrap event counting around the lookup phase
9  foreach key in lookup_workload
10     tree.lookup(key)
11 event_counter.stop()

12 results = event_counter.result()          // Consume results for further processing
```

Fig. 2: Profiling only the lookup-phase of a tree benchmark using perf-cpp.

One example of such an abstraction is the *perf-cpp* [Mü23] library, which will be used as an illustrative application in this tutorial. Similar to the Perf *stat* subcommand in Linux Perf, perf-cpp enables to specify hardware events for counting. However, in contrast to Linux Perf, perf-cpp provides the capability to control profiling directly from within the application, which enables monitoring of specific code regions. For instance, when benchmarking a tree-like data structure, it is common practice to exclude data generation from performance measurements and to monitor individual phases, such as the fill and lookup phases, separately. Figure 2 demonstrates the use of perf-cpp in this context: Following workload generation (lines 4 and 5), the tree is populated with data (lines 6 and 7). Event counting begins just before the lookup phase and ends thereafter (lines 8–11), enabling precise measurement of only this phase. Upon executing all lookups, the results of the monitored hardware events (see line 3) can be consumed and processed for further analysis (line 12).

3.2 Access Analysis via Sampling

While event counting provides an initial start-to-finish insight into the interaction between hardware and software of specific code segments, it lacks details on how this interaction evolves throughout the execution. Take, for instance, the case of a tree-like structure: Trees are typically plagued by *pointer-chasing*, where the next node to visit during a traversal is accessed directly after identification—leaving no space for the hardware- or software-prefetcher to jump in. To address this challenge, the use of *coroutines* enables pipelining of node accesses, effectively introducing a time gap between identifying and accessing a node. This interval allows for transferring nodes closer to the CPU via *software-based prefetching* [Jo18, Ps19, HLW20, MT21].



```
tree = Tree()
sampler = Sampler()
sampler.trigger("mem-loads")
sampler.memory_address(true)
sampler.memory_latency(true)

sampler.start()
foreach key in lookup_workload
    tree.lookup(key)
sampler.stop()

results = sampler.result()
```

```
; address, latency
0x4ac8f, 185
0x13138, 89
0x4adad, 276
0x40145, 5
0x13169, 16
...
```

node	mem. addr.
root	`0x40056`
inner	`0x12fab`
...	...
...	...
leaf	`0x4abcd`

Fig. 3: Sampling tree-lookups within the application and mapping samples to tree nodes.

Challenges. Although the combination of pipelining and prefetching is known for reducing the number of CPU cycles spent waiting for data to arrive, these stalls have not been entirely eliminated in former research. For a more in-depth analysis, sampling emerges as an invaluable technique. However, tools like Linux Perf and Intel VTune predominantly rely on code-based profiling that maps sampled instruction pointers to specific lines of code, such as those checking a node's latch or accessing its payload. This profiling can pinpoint which code experiences high memory latency, though it often provides a somewhat skewed view; access penalties vary across the tree, with higher levels near the root experiencing greater cache locality than deeper, leaf nodes. While Linux Perf integrates memory-based sampling, its efficacy is limited by the challenge of correlating sampled memory addresses with application-specific data structures, like individual tree nodes.

Pinpointing Memory Latency. This is where in-application sampling becomes valuable: By capturing access details such as memory addresses and latency directly within the application, this method not only enables precise, fine-grained profiling of specific code segments but also allows to enrich these samples with application-specific knowledge.

Figure 3 outlines this approach using perf-cpp: First, the lookup phase of a tree benchmark is monitored by initiating the sampler prior to executing lookups and stopping it afterward (①). The gathered data (②) are subsequently correlated with specific tree nodes using a dictionary that catalogs all node instances (③). Once all samples are accurately mapped to their respective nodes, the average access latency for each tree node can be computed (④).

Analyzing Software Prefetching for B$^+$-trees. We will now zoom into a detailed analysis driven by in-application sampling, focusing on software prefetching techniques within B$^+$-trees. To take a close look, we augmented a well-established B$^+$-tree implementation [LHN19] with coroutine-based pipelining and software-based prefetching techniques as proposed by Psaropoulos et al. [Ps19]. For our experiments, we utilized the YCSB benchmark [Co10] with 100 million records and lookup requests, executing the benchmark on a single thread. Note that we used different node sizes: 4 kB nodes yield the best

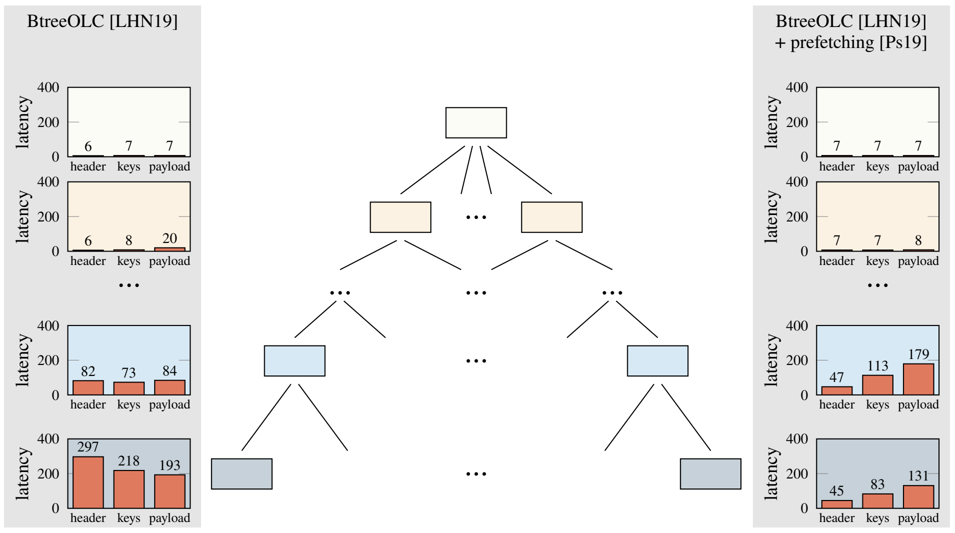


Fig. 4: Memory access latency for different node-segments (header, keys, payloads) in various tree levels derived from sampling—*without* prefetching (left) and *with* software-based prefetching (right).

performance for the unmodified B$^+$-tree, while software-prefetching necessitates smaller nodes to avoid overwhelming the LFB [KMT24]. Therefore, for the prefetching version, we employed nodes sized at 256 B.

As outlined in Figure 3, we configured perf-cpp to sample exclusively during the lookup phase and requested to include memory addresses and latency details in the samples. However, in contrast, each sample was not only mapped to a distinct tree node but to a specific data segment within that node. This allows to aggregate memory latency data for different segments of a node: the *header* (typically accessed first), the *keys* segment, and the *payload* containing the value or child pointer.

Figure 4 presents the results, detailing the average access latency (in CPU cycles) for each node segment at the top two and bottom two levels of the tree. The left side of the figure illustrates the latency for the original B$^+$-tree implementation *without* prefetching, while the right side shows the outcomes with prefetching applied. Unsurprisingly, in both scenarios, the root node exhibits cache locality, with latency comparable to those of the L1 data cache—the same applies to the root's straight child nodes. In contrast, the deeper levels, such as the leaves, exhibit significantly higher access latency, reaching up to nearly 300 cycles when prefetching is not implemented. With prefetching applied, the latency for these deeper node levels is significantly reduced, effectively *doubling* the throughput. Nonetheless, the experiment indicates that many node accesses still experience noticeable latency.

Consequently, these insights can draw a range of optimizations to improve prefetching in tree-like data structures. First, since the upper levels already benefit from cache locality, prefetching these nodes is redundant. By omitting prefetching for nodes already sitting in the cache, we can reduce instruction bandwidth spent on executing prefetch instructions and minimize the overhead associated with coroutines: Nodes visited consecutively without prefetching involved can be accessed within a single coroutine. In subsequent experiments, we saw a modest improvement of approximately 4% when applying this optimization for the first three tree levels. Note that the effectiveness of this optimization is heavily influenced by the cache size and the size of the nodes.

Second, the findings suggest that prefetching is not being applied optimally, potentially due to the *prefetch distance*—the time gap between executing the prefetch instruction and the actual data access. In our implementation, this interval is dictated by a round-robin coroutine scheduler (as described in [Ps17]) and a *fixed* number of coroutines[2] active simultaneously: Each coroutine is executed after all others yield the control flow. However, data movement from memory into caches varies in duration; similarly, coroutines at different stages of a traversal have variable execution times (particularly with the mentioned optimization in place). Accordingly, the interval between prefetch and access varies and is hard to control. This challenge becomes further complex when considering operations beyond lookups that might experience more *heterogeneous* execution times. We believe that, to enhance the effectiveness of prefetching combined with coroutines, adopting a more refined scheduling approach could be beneficial—one that accounts for both the execution duration of coroutines and the memory latency.

4 Projected Time-frame

The tutorial is intended as an interactive, hands-on tutorial with a duration of approximately 45 minutes. We plan to start with an introduction of the Linux perf subsystem and how perf-cpp interacts with it (roughly 10 minutes).

Following this, we will give a practical introduction to performance profiling, including a demonstration on how to set up the perf-cpp library (10 minutes).

Finally, the B-tree with optimistic lock coupling will serve as the poster child for a more complex data structure. We plan to do a live demonstration on how sampling can be used to e.g., identify the effectiveness of software prefetching as detailed in Section 3.2. This part is explicitly intended for audience interaction, as we plan to provide a script (e.g., via *GitHub*), that will build and execute the benchmark on the devices of the participants. We plan to compare the results of the participants with our results to highlight the differences that can occur on different types of hardware. We will also have some "backup" results from other machines for comparisons (25 minutes).

[2] We found that scheduling 12 coroutines simultaneously yielded the best performance on our system.

Bibliography

[AH17] Akiyama, Soramichi; Hirofuchi, Takahiro: Quantitative Evaluation of Intel PEBS Overhead for Online System-Noise Analysis. In: Proceedings of the 7th International Workshop on Runtime and Operating Systems for Supercomputers, ROSS@HPDC. ACM, pp. 3:1–3:8, 2017.

[AM24] AMD: AMD uProf. https://www.amd.com/de/developer/uprof.html, 2024. Accessed 21 August 2024.

[Be21] Beischl, Alexander; Kersten, Timo; Bandle, Maximilian; Giceva, Jana; Neumann, Thomas: Profiling dataflow systems on multiple abstraction levels. In: EuroSys ’21: Sixteenth European Conference on Computer Systems. ACM, pp. 474–489, 2021.

[Co10] Cooper, Brian F.; Silberstein, Adam; Tam, Erwin; Ramakrishnan, Raghu; Sears, Russell: Benchmarking cloud serving systems with YCSB. In: Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC. ACM, pp. 143–154, 2010.

[Da23] Dang, Weizhen; Yu, Tao; Wang, Haibo; Li, Fenghua; Wang, Jilong et al.: PerfMon: Measuring application-level performance in a large-scale campus wireless network. China Communications, 20(3):316–335, 2023.

[DM10] De Melo, Arnaldo Carvalho: The new linux’perf’tools. In: Slides from Linux Kongress. volume 18, pp. 1–42, 2010.

[Dr07] Drongowski, Paul J: Instruction-based sampling: A new performance analysis technique for AMD family 10h processors. Advanced Micro Devices, 1(3):11, 2007.

[Dr10] Drongowski, Paul J.; Yu, Lei; Swehosky, Frank; Suthikulpanit, Suravee; Richter, Robert: Incorporating Instruction-Based Sampling into AMD CodeAnalyst. In: IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS. IEEE Computer Society, pp. 119–120, 2010.

[HLW20] He, Yongjun; Lu, Jiacheng; Wang, Tianzheng: CoroBase: Coroutine-Oriented Main-Memory Database Engine. Proc. VLDB Endow., 14(3):431–444, 2020.

[In24a] Intel®: Intel® 64 and IA-32 Architectures Software Developer’s Manual. https://cdrdv2.intel.com/v1/dl/getContent/671200, 2024. Accessed 17 October 2024.

[In24b] Intel®: Intel® VTune™ Profiler. https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html, 2024. Accessed 21 August 2024.

[Jo18] Jonathan, Christopher; Minhas, Umar Farooq; Hunter, James; Levandoski, Justin J.; Nishanov, Gor V.: Exploiting Coroutines to Attack the "Killer Nanoseconds". Proc. VLDB Endow., 11(11):1702–1714, 2018.

[KMT24] Kühn, Roland; Mühlig, Jan; Teubner, Jens: How to Be Fast and Not Furious: Looking Under the Hood of CPU Cache Prefetching. In: Proceedings of the 20th International Workshop on Data Management on New Hardware, DaMoN. ACM, pp. 9:1–9:10, 2024.

[Kü23] Kühn, Roland; Biebert, Daniel; Hakert, Christian; Chen, Jian-Jia; Teubner, Jens: Towards Data-Based Cache Optimization of B+-Trees. In: Proceedings of the 19th International Workshop on Data Management on New Hardware, DaMoN. ACM, pp. 63–69, 2023.

[Le18] Leis, Viktor: PerfEvent. https://github.com/viktorleis/perfevent, 2018. Accessed 23 November 2024.

[LHN19] Leis, Viktor; Haubenschild, Michael; Neumann, Thomas: Optimistic Lock Coupling: A Scalable and Efficient General-Purpose Synchronization Method. IEEE Data Eng. Bull., 42(1):73–84, 2019.

[LLS13] Levandoski, Justin J.; Lomet, David B.; Sengupta, Sudipta: The Bw-Tree: A B-tree for new hardware platforms. In: 29th IEEE International Conference on Data Engineering, ICDE. IEEE Computer Society, pp. 302–313, 2013.

[Mi24] Miksits, Samuel; Shi, Ruimin; Gokhale, Maya B.; Wahlgren, Jacob; Schieffer, Gabin; Peng, Ivy: Multi-level Memory-Centric Profiling on ARM Processors with ARM SPE. CoRR, abs/2410.01514, 2024.

[MT21] Mühlig, Jan; Teubner, Jens: MxTasks: How to Make Efficient Synchronization and Prefetching Easy. In: SIGMOD: International Conference on Management of Data. ACM, pp. 1331–1344, 2021.

[Mü23] Mühlig, Jan: perf-cpp: Access Performance Counters from C++ Applications. https://github.com/jmuehlig/perf-cpp, 2023. Accessed 23 November 2024.

[No20] Noll, Stefan; Teubner, Jens; May, Norman; Böhm, Alexander: Analyzing memory accesses with modern processors. In: 16th International Workshop on Data Management on New Hardware, DaMoN. ACM, pp. 1:1–1:9, 2020.

[Ps17] Psaropoulos, Georgios; Legler, Thomas; May, Norman; Ailamaki, Anastasia: Interleaving with Coroutines: A Practical Approach for Robust Index Joins. Proc. VLDB Endow., 11(2):230–242, 2017.

[Ps19] Psaropoulos, Georgios; Legler, Thomas; May, Norman; Ailamaki, Anastasia: Interleaving with coroutines: a systematic and practical approach to hide memory latency in index joins. VLDB J., 28(4):451–471, 2019.

[Sa23] Sasongko, Muhammad Aditya; Chabbi, Milind; Kelly, Paul H. J.; Unat, Didem: Precise Event Sampling on AMD Versus Intel: Quantitative and Qualitative Comparison. IEEE Trans. Parallel Distributed Syst., 34(5):1594–1608, 2023.

[Sc23] Schubert, Nils L.; Grulich, Philipp M.; Zeuch, Steffen; Markl, Volker: Exploiting Access Pattern Characteristics for Join Reordering. In: Proceedings of the 19th International Workshop on Data Management on New Hardware, DaMoN. ACM, pp. 10–18, 2023.

[We16] Weaver, Vincent M: Advanced hardware profiling and sampling (PEBS, IBS, etc.): creating a new PAPI sampling interface. Technical Report UMAINE-VMWTR-PEBS-IBS-SAMPLING-2016-08. University of Maine, Tech. Rep., 2016.

[Yi20] Yi, Jifei; Dong, Benchao; Dong, Mingkai; Chen, Haibo: On the precision of precise event based sampling. In: APSys '20: 11th ACM SIGOPS Asia-Pacific Workshop on Systems. ACM, pp. 98–105, 2020.

[ZF15] Zeuch, Steffen; Freytag, Johann-Christoph: Selection on Modern CPUs. In: Proceedings of the 3rd VLDB Workshop on In-Memory Data Mangement and Analytics. ACM, pp. 5:1–5:8, 2015.