Breaking the Cycle - A Short Overview of Memory-Access Sampling Differences on Modern x86 CPUs

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

As hardware complexity increases, profiling becomes essential for understanding system behavior. This paper compares different x86 sampling implementations for memory access profiling, revealing their complementary capabilities and limitations. Plus, we demonstrate that current abstractions like the *perf subsystem* inadequately expose platform-specific features.

1 Introduction

To fully utilize modern hardware, performance-sensitive applications must be designed with hardware-conscious principles (e.g., [5, 7, 12, 14, 15, 18, 20]). However, sophisticated mechanisms such as out-of-order execution and memory prefetching have transformed hardware into a black box—turning hardware-aware optimizations into an uphill battle.

The silver lining lies in Performance Monitoring Units (PMUs)—specialized components embedded within modern CPUs—which allow engineers to examine software execution under a magnifying glass (e.g., [6, 17]). *Sampling*-based profiling techniques, in particular, offer invaluable insights by revealing critical details such as *memory access patterns* throughout execution. But, PMU implementations vary substantially across hardware vendors and CPU generations: Diverse operating modes and consequently different features complicate the comparison of software executions across heterogeneous hardware platforms [19]. This challenge, however, represents two sides of the same coin: These architectural differences can be leveraged advantageously when properly understood.

This paper presents a comparative analysis of two leading PMU-based sampling techniques from the *memory-access sampling* perspective: Intel's Processor Event-Based Sampling (PEBS) [4] and AMD's Instruction-Based Sampling (IBS) [9, 10]. We equip engineers with critical knowledge about which platform unveils specific execution details, enabling them to select the right tool for each analytical question. Furthermore, we demonstrate how commonly used abstractions—particularly the *perf subsystem* and the command-line interface *perf*—fall short of exposing the full spectrum of capabilities across different hardware platforms, leaving valuable performance insights on the table.

2 Memory Address Sampling on x86 plattforms

Nearly all modern x86 architectures are equipped with *memory-access sampling* capabilities that can periodically generate a snapshot of the actual accessed logical and physical memory addresses [17]. However, the collection of samples by the two prominent vendors—AMD and Intel—differs significantly as already shown in [19]. Intel's PEBS facility explicitly allows to sample load and store instructions (e.g., every x-th load instruction will create a new sample) [11]. AMD employs a different strategy with its IBS, where every x-th micro-operation (μOp), regardless of the type

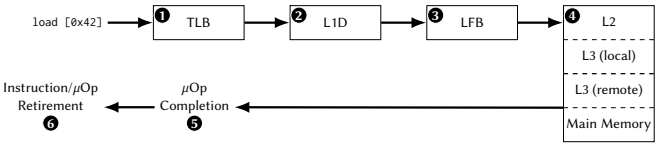

Figure 1: Sequence of a memory access and the hardware components involved [1]. The numbers indicate points at which information can be obtained for the memory samples.

(*arithmetic*, *load/store*, …), will be tagged and traced throughout the entire processing pipeline [1]. In addition to the accessed addresses, both hardware makers provide additional information in their respective samples that allow deductions about the utilization of critical system resources. To highlight which information can be retrieved in IBS and PEBS, we will follow a memory request through the various execution stages and reveal, at each stage, what information both vendors provide in their samples (Figure 1).

TLB Access (❶). If an instruction/μOp that accesses memory is executed, the accessed logical address has to be translated into a physical address by consulting the Translation Lookaside Buffers (TLBs), which then returns the page address on a TLB hit, or issues page walk to retrieve the physical page address from the page table (on a miss). While PEBS merely reports the TLB hit/miss status, IBS reports which level was hit and quantifies TLB refill latency [2], i.e., when the L1 TLB was refilled from the L2 TLB or a page walk was issued due to a miss in both TLB levels.

L1D Cache Access (❷). After the address translation, the caches will be consulted to find the requested data element[1]. If the data element can be found in the L1 data cache (L1d), both sampling implementations report the L1d as the data source. The latency for the cache access, however, will only be reported by PEBS. In contrast, IBS reports the latency when the request missed the L1d.

Line Fill Buffer and Memory Access (❸ and ❹). If the data element cannot be found in the L1d, the address of the cache line that contains the data element will be written to the Line Fill Buffer (LFB) (or Miss Address Buffer (MAB) on AMD systems[2]) and will then be serviced by a higher cache level or the memory subsystem. Once the memory request has been processed, both vendors report information about the latency and the data source from which the data element was retrieved (e.g., the L3 cache or the main memory). However, some key distinctions exist between the sampling implementations from both vendors. If the cache line is already registered in the LFB, PEBS reports the LFB as the data source, while IBS reports the real source from which the data was finally retrieved. In addition, IBS-samples also contain more information

[1] In virtually-indexed-physically-tagged (VIPT) caches, the address translation and cache access can be parallelized to a certain extent.
[2] For the sake of simplicity, we use the term LFB in this paper, although we refer to the MAB on AMD systems.

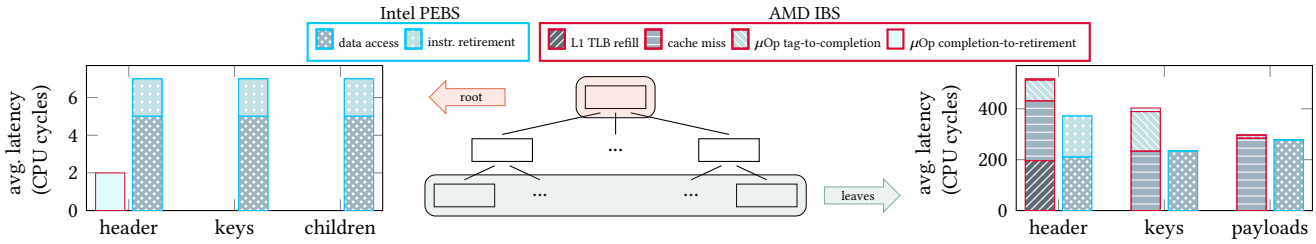

Figure 2: The average access latency during B$^+$-tree lookups at the root node (left) and the leaf nodes (right). Since IBS only reports cache-miss latency, no latency is reported for the root node. The μOp tag-to-completion latency is calculated by subtracting μOp completion-to-retire latency from the μOp tag-to-retire latency.

about the memory request itself, like the page size, the number of requested bytes, a flag if an LFB slot was allocated, and the number of actual allocated LFB slots. This information can be crucial, e.g., to identify bottlenecks caused by requests flooding the LFB, since instructions/μOps stall until an LFB slot becomes available [12].

PEBS distinguishes between loads and stores, counting load prefetches as accesses to the L1d. IBS reports software prefetches as such, although it does not report the cache miss latency.

Instruction/μOp retirement (❺ and ❻). After the memory subsystem has retrieved the requested data element, the instruction/μOp will retire. In contrast to PEBS, which then only reports the total latency for the instruction execution, IBS additionally reports the cycles spent between the completion of the μOp and the point where the μOp is considered as successfully retired.

Overall, IBS reports four latencies: For refilling the L1 TLB, for fulfilling requests that miss the L1d, from tagging the μOp until retirement, and separately from completion to retirement. PEBS provides two latencies: data access and instruction retirement. Additionally, further information such as the occupancy of the LFB is also provided by samples from IBS.

Since *sampling* can introduce significant overhead to the operating system, when many samples are created, Intel's PEBS offers the possibility to filter the samples by latency and keep only samples with latency higher than a configurable threshold. AMD introduced this feature also with the latest *Zen 5* micro-architecture [3].

Perf subsystem. The *perf subsystem*—baked into the Linux kernel—allows to interact with PMUs and builds the foundation for the *perf* command-line-interface. The accessible information from the actual version of the *perf subsystem* seems to be leaned on the details provided by PEBS. On Intel systems, it provides access to nearly all sampled information, whereas on AMD systems many details, such as the latency for TLB refills or the number of occupied LFB entries will not be shown. One way to get this information is to read the raw samples that are provided by the *perf subsystem*. However the samples need to be manually processed to yield the required information, for example, by using libraries like perf-cpp [16].

3 Practical Latency Insights

To briefly illustrate the architectural divergence between these sampling mechanisms, we use a B$^+$-tree [13][3] *lookup* operation as our case study on a AMD Zen 4 system and a machine with Intel's Sapphire Rapids architecture.

[3]We borrowed the implementation from https://github.com/wangziqi2016/index-microbench.

We utilize *memory-access sampling* through the *perf subsystem*, periodically capturing memory addresses and access metrics—including latency details—throughout lookup operations using the *YCS Benchmark* [8] (100 M lookups against a tree populated with 100 M records). On the AMD system we had to instruct the *perf subsystem* to record raw IBS values to access these detailed metrics—particularly the TLB refill latency—as they remain inaccessible through standard interfaces.

Figure 2 visualizes the access latency distributions recorded via AMD's IBS and Intel's PEBS for two critical tree levels: the root node (left) and leaf nodes (right). The plots illustrate the average latency for individual lookups, segmented according to the latency measurement capabilities of each sampling implementation. Unlike instruction sampling, which allows the correlation of performance data with lines of code and functions, *memory-access sampling* enables the direct mapping of samples to specific tree nodes and their structural components (e.g., headers, keys, and payloads). This distinction is crucial, as all nodes share identical code paths, making instruction-based sampling inadequate for analyzing access characteristics across distinct nodes or tree levels.

As one would expect, the root node exhibits minimal access latency due its tendency to reside in the L1d. However, IBS reports no latency measurements in this scenario, as it exclusively captures cache *miss* events. In contrast, PEBS provides granular insights, reporting 5 CPU cycles for cache access and an additional 2 cycles for instruction retirement.

Leaf node accesses present a different profile, frequently triggering cache and TLB misses. For the header segment—typically the first component accessed—IBS delivers detailed timing breakdowns: approximately 200 cycles for TLB refill operations plus 230 cycles for cache miss resolution and ~ 90 cycles for retiring the μOp. PEBS, however, presents a more consolidated view, reporting 210 cycles for cache operations and roughly 160 cycles for instruction retirement, with the latter inherently incorporating TLB latency.

4 Conclusion and Outlook

This paper provided a condensed overview of the differences in *memory-access sampling* on recent `x86` architectures. We showed that hardware makers provide valuable additional information in their memory samples. However, not all information is clearly communicated through the *perf subsystem*. This preliminary work serves as a foundation for further research, as we intend to investigate other hardware architectures in greater detail, including the Statistical Profiling Extension (SPE) in recent ARMv8 systems.

References

[1] Advanced Micro Devices, Inc. 2024. *AMD64 Technology. AMD64 Architecture Programmer's Manual - Volume 2: System Programming.* Santa Clara, CA, USA.

[2] Advanced Micro Devices, Inc. 2024. *Processor Programming Reference (PPR) for AMD Family 19h Model 11h, Revision B2 Processors.* Santa Clara, CA, USA.

[3] Advanced Micro Devices, Inc. 2024. *Processor Programming Reference (PPR) for AMD Family 1Ah Model 44h, Revision B0 Processors.* Santa Clara, CA, USA.

[4] Soramichi Akiyama and Takahiro Hirofuchi. 2017. 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, 3:1–3:8. doi:10.1145/3095770.3095773

[5] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. 2013. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *29th IEEE International Conference on Data Engineering*. IEEE Computer Society, 362–373. doi:10.1109/ICDE.2013.6544839

[6] Alexander Beischl, Timo Kersten, Maximilian Bandle, Jana Giceva, and Thomas Neumann. 2021. Profiling dataflow systems on multiple abstraction levels. In *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*, Antonio Barbalace, Pramod Bhatotia, Lorenzo Alvisi, and Cristian Cadar (Eds.). ACM, 474–489. doi:10.1145/3447786.3456254

[7] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. 1999. Database Architecture Optimized for the New Bottleneck: Memory Access. In *Proceedings of 25th International Conference on Very Large Data Bases*. Morgan Kaufmann, 54–65. http://www.vldb.org/conf/1999/P5.pdf

[8] 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*. ACM, 143–154. doi:10.1145/1807128.1807152

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

[10] Paul J. Drongowski, Lei Yu, Frank Swehosky, Suravee Suthikulpanit, and Robert Richter. 2010. Incorporating Instruction-Based Sampling into AMD CodeAnalyst. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS*. IEEE Computer Society, 119–120. doi:10.1109/ISPASS.2010.5452049

[11] Intel®. 2024. Intel® 64 and IA-32 Architectures Software Developer's Manual. https://cdrdv2.intel.com/v1/dl/getContent/671200. Accessed: March 20, 2025.

[12] Roland Kühn, Jan Mühlig, and Jens Teubner. 2024. 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, 9:1–9:10. doi:10.1145/3662010.3663451

[13] 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. http://sites.computer.org/debull/A19mar/p73.pdf

[14] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for new hardware platforms. In *29th IEEE International Conference on Data Engineering, ICDE*. IEEE Computer Society, 302–313. doi:10.1109/ICDE.2013.6544834

[15] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. 2002. Optimizing Main-Memory Join on Modern Hardware. *IEEE Trans. Knowl. Data Eng.* 14, 4 (2002), 709–730. doi:10.1109/TKDE.2002.1019210

[16] Jan Mühlig. 2023. perf-cpp: Access Performance Counters from C++ Applications. https://github.com/jmuehlig/perf-cpp. Accessed: March 20, 2025.

[17] Stefan Noll, Jens Teubner, Norman May, and Alexander Böhm. 2020. Analyzing memory accesses with modern processors. In *16th International Workshop on Data Management on New Hardware, DaMoN 2020, Portland, Oregon, USA, June 15, 2020*, Danica Porobic and Thomas Neumann (Eds.). ACM, 1:1–1:9. doi:10.1145/3399666.3399896

[18] Michael L. Samuel, Anders Uhl Pedersen, and Philippe Bonnet. 2005. Making CSB+-Tree Processor Conscious. In *Workshop on Data Management on New Hardware, DaMoN*. http://www-2.cs.cmu.edu/%7Edamon2005/damonpdf/2%20making%20csb+%20trees%20processor%20conscious.pdf

[19] Muhammad Aditya Sasongko, Milind Chabbi, Paul H. J. Kelly, and Didem Unat. 2023. Precise Event Sampling on AMD Versus Intel: Quantitative and Qualitative Comparison. *IEEE Trans. Parallel Distributed Syst.* 34, 5 (2023), 1594–1608. doi:10.1109/TPDS.2023.3257105

[20] Ambuj Shatdal, Chander Kant, and Jeffrey F. Naughton. 1994. Cache Conscious Algorithms for Relational Query Processing. In *Proceedings of 20th International Conference on Very Large Data Bases*. Morgan Kaufmann, 510–521. http://www.vldb.org/conf/1994/P510.PDF