Fine-Grained Hardware Profiling – Are You Using the Right Tools?

Aarati Kakaraparthy
University of Wisconsin, Madison
aaratik@cs.wisc.edu

Jignesh M. Patel
Carnegie Mellon University
jignesh@cmu.edu

ABSTRACT

We consider the problem of *fine-grained hardware profiling*, i.e., profiling the hardware while the desired section of the program is executing. Although this requirement is frequently encountered in practice, its importance has not been emphasized in literature so far. In this work, we compare and validate three tools for performing fine-grained profiling on Linux platforms – `perf`, PAPI, and a homegrown tool PMU-metrics. `perf` has been used in the past for fine-grained profiling in an erroneous manner, producing inaccurate metrics as a result. On the other hand, PAPI and PMU-metrics produce accurate metrics for profiling at the ms-scale, while PMU-metrics enables profiling even at the μs-scale. Thus, we hope that our analysis will help systems practitioners choose the right tool for performing fine-grained profiling at different time scales.

1. INTRODUCTION

Hardware profiling is an important part of developing efficient data processing methods. This process frequently involves using fine-grained profiling to drill down different parts of the design to obtain valuable performance metrics such as cache misses, CPU cyces, TLB misses, etc. Fine-grained profiling also enables development of online adaptive algorithms that utilize performance metrics to make decisions at runtime [12, 16]. However, it can be challenging to obtain accurate hardware metrics at sub-ms time scales.

To highlight the challenge of profiling fast data processing methods, consider the example of a hash table. An optimized hash table typically takes a few hundred cycles for each probe operation, i.e., $< 0.1\mu s$. However, existing tools do not support accurate and non-intrusive profiling at such small time scales, thus forcing us to increase the duration of profiling using batching as shown in Listing 1.

We note that batching may not be possible if the target component is deeply integrated in a system with many hierarchical modules, and it may be important to examine the performance of just the target as it interacts with the rest of the system. For developing online adaptive algorithms, it may be necessary to collect metrics at a certain granularity for the technique to be effective. Fine-grained profiling enables isolating the performance of different components of a system, thus making it an essential part of developing and tuning systems.

```
#define N          10^8 // 100M
#define BATCH_SIZE 10^7 // 10M (config.)
#define HM_SIZE    10^6 // 1M

void main() {
  /* Generate 1M random keys */
  Keys keys = genKeys(HM_SIZE);
  /* Populate the hash table */
  HashMap hm = new HashMap();
  hm.populate(keys);
  /* Generate 100M fetch requests */
  Requests reqs[] = genFetchReqs(keys, N);
  for (int i=0; i<N; i+=BATCH_SIZE) {
    /* Collect metrics for each batch */
    hm.processReqs(&reqs[i:i+BATCH_SIZE]);}
}
```

Listing 1: An example highlighting the requirement of fine-grained profiling. Hardware metrics need to be measured for each batch of fetch requests independently.

Given the importance of fine-grained profiling, some questions that arise are: Which tools are available for this task? At what time scales do they work? What trade-offs do they offer? These questions have not been addressed in prior work utilizing fine-grained profiling [13, 14]. To address these questions, we make the following contributions:

- **Applications of fine-grained profiling**: To highlight the importance of fine-grained profiling, we study two applications. First, we profile the performance of a hash table as shown in Listing 1 [12]. Second, we study BLARE, an adaptive log processing framework for regular expression (regex) matching [16] that utilizes fine-grained profiling.

```
struct Perf {
  profile(int id, function<void()> body) {
    /* Launch perf stat to track parent */
    pid_t pid;
    stringstream parent_id;
    parent_id << getpid();
    pid = fork();
    if (pid == 0) {
      exit(execl("/usr/bin/perf",
        "perf", "stat",
        "-e", "L2_RQSTS.REFERENCES",
        "-p", parent_id.str().c_str(),
        nullptr));}
    /* Run body */
    body();
    /* Kill perf stat */
    kill(pid,SIGINT);
    waitpid(pid,nullptr,0);
  }
};
void main() {
  ...
  for (int i=0; i<N; i+=BATCH_SIZE) {
    Perf::profile(i/BATCH_SIZE, [&]() {
      /* Collect metrics for each batch */
      hm.processReqs(&reqs[i:i+BATCH_SIZE]);
    });}
}
```

Listing 2: Fine-grained profiling with `perf`. A child `perf` process is launched to track the parent while the section of code to profile (`body`) is executed.

- **Comparing multiple profiling tools**: We validate the metrics from three representative tools for profiling on Linux platforms – `perf`, PAPI, and PMU-metrics. The former is a popular command-line profiling tool, while the latter two libraries perform profiling from within the program.
- **Debunking erroneous usage of `perf`**: `perf` can potentially be used for fine-grained profiling as shown in Listing 2. However, we show that `perf` yields incorrect metrics and interferes with execution when used in this manner.
- **Fine-grained profiling at μs-scale**: PMU-metrics enables profiling at the μs-scale while imposing low overhead (+8% execution time) and providing good accuracy (2% error), which facilitates execution of BLARE with minimal overhead (10%).
- **Overall recommendations**: We recommend using either PAPI or PMU-metrics for profiling at the *ms*-scale, while PMU-metrics should be preferred for profiling at the μs-scale.

Thus, we evaluate the validity, accuracy and overhead of `perf`, PAPI, and PMU-metrics at time scales ranging from sub-*s* to sub-*ms* by studying applications to hash tables (Listing 1, [12]) and BLARE [16].

2. BACKGROUND

In this section, we briefly discuss performance monitoring in CPUs (§2.1-§2.2) to highlight the differences between `perf`, PAPI, and PMU-metrics, followed by describing how fine-grained profiling can be performed using each of the three tools (§2.3).

2.1 Performance Monitoring in CPUs

Present-day CPUs come with an integrated *performance monitoring unit* (PMU) that can track hardware events such as cache misses. Typically, PMUs have a fixed number of units that can independently count processor events. The set of supported hardware events and the implementation of the PMU varies across different CPU vendors. Even for the same vendor, the supported hardware events can vary between different models of CPUs.

2.2 Interfacing the PMU

In a nutshell, the interface to the PMU is a collection of registers that can be programmed for tracking hardware events. The low-level interface to the PMU varies across different CPU vendors, and the *perf_event* [4] Linux kernel library provides standardized functions that hide the hardware-specific steps involved in utilizing the PMU. The function *perf_event_open* [5] initializes performance monitoring and returns a file descriptor to access the hardware metrics. Thus, accessing hardware metrics using the perf_event library is equivalent to reading from a file. Some prominent vendors supported by perf_event are Intel, AMD, ARM and IBM.

A cheaper interface compared to perf_event_open is the `rdpmc` [6] x86 assembly instruction that can directly read metrics from the performance counter registers without initiating a system call. The `rdpmc` instruction requires knowledge of which registers contain the output of the programmed PMU units.

2.3 Fine-Grained Profiling on Linux

We compare three different tools for fine-grained profiling – `perf` [11], PAPI [15], and PMU-metrics [10]. Both `perf` and PAPI are built on the perf_event library, whereas PMU-metrics uses a bare metal approach of directly accessing the PMU hardware counters. Below we describe how each of these tools can be used for fine-grained measurement of L1 cache misses on an Intel Xeon Silver 4114 CPU belonging to the Skylake microarchitecture family.

2.3.1 `perf`

Linux `perf` is a popular command-line profiling tool. The `perf stat` command can be used for fine-grained profiling as shown in Listing 2 – a child

```
void main() {
  ...
  int EventSet1 = PAPI_NULL;
  long long metrics[1];
  PAPI_library_init(PAPI_VER_CURRENT);
  PAPI_create_eventset(&EventSet1);
  /* Add PAPI_L1_TCM preset event */
  PAPI_add_named_event(EventSet1,
    "PAPI_L1_TCM");
  for (int i=0; i<N; i+=BATCH_SIZE) {
    /* Start PAPI */
    PAPI_start(EventSet1);
    /* Collect metrics for each batch */
    hm.processReqs(&reqs[i:i+BATCH_SIZE]);
    /* Stop PAPI */
    PAPI_stop(EventSet1, metrics); }
}
```

Listing 3: Fine-grained profiling with PAPI. PAPI provides API functions to program, start, and stop performance event counters.

`perf` process is launched to track the parent when the code to profile is executed, thus (presumably) profiling only the desired section of the program. In Listing 2, L1 cache misses are measured using the event `L2_RQSTS.REFERENCES`.

2.3.2 *PAPI*

PAPI is designed for fine-grained "first person" profiling, i.e., profiling from within the program. PAPI provides API functions to program the PMU (`PAPI_add_named_event`) and start/stop counting events (`PAPI_start/stop`) as shown in Listing 3. The library provides a collection of preset events with the goal of implementing cross-platform compatibility. Internally, the library uses one or more hardware-specific events (referred to as native events) to estimate the output of the preset event. In Listing 3, we use the preset event `PAPI_L1_TCM` to count L1 cache misses. Note that PAPI can also be used to directly count hardware-specific events.

2.3.3 *PMU-metrics*

PMU-metrics provides a fine-grained interface similar to PAPI, as shown in Listing 4. However, there are two key differences between PAPI and PMU-metrics. First, PAPI uses the `perf_event_open` system call to program the PMU at runtime, whereas the PMU-metrics library directly programs the PMU registers using `wrmsr` [9]. Second, PAPI relies on the file interface provided by the perf_event library to access the hardware metrics, whereas PMU-metrics uses the cheaper `rdpmc` assembly instruction to directly read the performance counter registers. Currently, PMU-metrics supports Intel CPUs released since 2007, and can be extended for other vendors.

```
/* Choose performance events */
#define PERFEVTSEL0 SKL_L2_REFS

/* Program the PMU on a chosen core */
$ ./setup.sh -c 1

/* Access counters in the program */
void main() {
  ...
  Metrics m;
  for (int i=0; i<N; i+=BATCH_SIZE) {
    /* Read start value of counters */
    getMetricsStart(m);
    /* Collect metrics for each batch */
    hm.processReqs(&reqs[i:i+BATCH_SIZE]);
    /* Read end value of counters */
    getMetricsEnd(m); }
}
```

Listing 4: Fine-grained profiling with PMU-metrics, with API functions to read performance counters using `rdpmc` x86 assembly instruction. The `setup.sh` script programs the PMU with performance events before running the application.

3. WHICH TOOLS ARE CORRECT?

In this section, we validate the metrics obtained from `perf`, PAPI, and PMU-metrics by profiling a hash table data structure (Listing 1).

3.1 Workload

We consider the example introduced in Listing 1 of measuring hardware metrics for each batch of fetch requests issued to a hash table. We process 100M fetch requests in batches of 10M (approximately 330ms processing time per batch without profiling), i.e., 10 batches of fetch requests are processed overall. The goal is to measure five hardware metrics for each batch – core cycles, instructions retired, L3, L2 and L1 cache misses. The workload executed is the same in all experiments.

3.2 Experimental Setup

All the experiments in this paper have been run on a dedicated server machine with an Intel Xeon Silver 4114 10-core CPU. The running process has been pinned to a specific core (two cores in case of profiling with `perf`) using `taskset` to mitigate the overhead of context switching. The frequency scaling governor has been set to "performance", which drives the cores at maximum possible frequency, i.e., 3GHz on our hardware. All the experiments have been repeated five times, and we compare the median metrics in all cases.

In addition to profiling using the libraries discussed in §2, we use the `clock_gettime` [2] function to measure total time elapsed for processing all

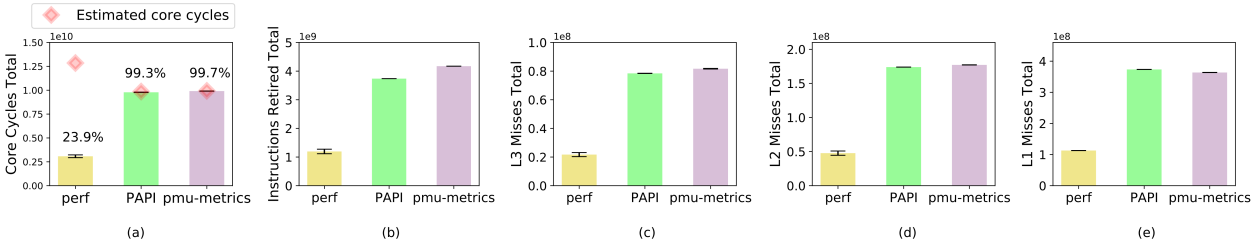

Figure 1: Comparing the aggregate metrics obtained from `perf`, PAPI, and PMU-metrics. The workload involved measuring hardware metrics individually for ten batches of 10M fetch requests each (approximately 330ms processing time per batch without profiling). The metrics obtained from `perf` are significantly different from the remaining two libraries, while PAPI and PMU-metrics report similar metrics in all cases (within 4% for core cycles, L3, L2, and L1 cache misses; within 10% for instructions retired). We validate the core cycles obtained from each of the libraries against estimated core cycles obtained from `clock_gettime`, and the three libraries have 23.9%, 99.3%, and 99.7% accuracy respectively as indicated in (a).

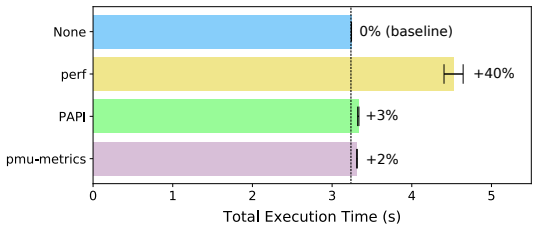

Figure 2: Increase in total execution time of the workload when different libraries are used for profiling. Relative to the baseline where profiling is not performed, `perf` increases execution time by 40% while PAPI and PMU-metrics impose negligible overhead of 3% and 2% respectively.

the 100M fetch requests each time. This function is known to provide good precision and accuracy for measuring time on Linux platforms [8].

3.3 perf *vs* PAPI *vs* PMU-metrics

Listings 2-4 show how to perform fine-grained profiling using `perf`, PAPI, and PMU-metrics respectively. Below we discuss the results obtained.

3.3.1 *Overhead of profiling*

Fig. 2 shows the increase in total running time of the whole workload (100M requests) when profiling using different tools. Compared to the baseline where profiling is not performed, PAPI and PMU-metrics impose a negligible overhead of 3% and 2% respectively. In contrast, profiling with `perf` causes a significant increase of 40% in total execution time of the workload, which suggests interference in the hash table execution from the child `perf` process.

3.3.2 *Comparing the metrics obtained*

We compare the metrics obtained from `perf`, PAPI, and PMU-metrics in Fig. 1. We find that the metrics obtained from `perf` are 3-4× lower compared to PAPI and PMU-metrics, both of which report similar metrics in all cases[1]. Since the same workload is being executed, we can conclude that not all of these tools give the correct metrics. This raises the question which of these tools are reliable.

3.3.3 *Which metrics are correct?*

To establish the correctness of the metrics, we validate the core cycles obtained from a tool against the core cycles estimated using the `clock_gettime` function. Given time t has elapsed while executing `processRequests` on a core with frequency f,

$$estimated\ core\ cycles = t \times f$$

Fig. 1a plots the measured and estimated core cycles for each of the tools[2]. Both PAPI and PMU-metrics attain a high accuracy of 99.3% and 99.7% respectively, indicating that their measurements are correct. Since the same profiling methodology is applied for other hardware events, we conclude that the metrics obtained from these tools are reliable.

On the other hand, the cycles obtained from `perf` are 76% lower than estimated, which suggests that the metrics obtained using `perf` exhibit a large deviation from the true value. This observation is further corroborated by the fact that the aggregate execution time reported by `perf` is 1.2s, which is far lower than the value obtained using `clock_gettime` (4.3s). Thus, we conclude that `perf` produces incorrect metrics and interferes with execution (§3.3.1) when used as shown in Listing 2. We discuss the reasons behind these observations in Fig. 3.

[1]The difference in metrics is within 4% for core cycles and cache misses. A deviation of 10% in instructions executed is encountered because PMU-metrics and PAPI use different hardware events (INST_RETD.ANY vs INST_RETD.ANY_P respectively) on Skylake CPUs.
[2]Note that the time elapsed, and thus the estimated core cycles are different for each tool.

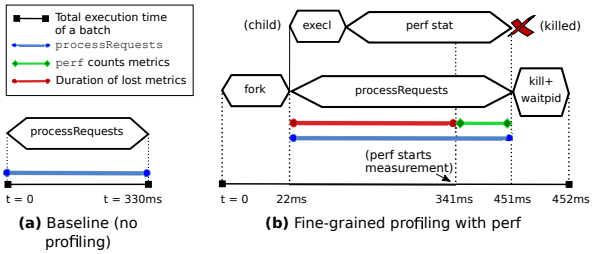

Figure 3: Investigating fine-grained profiling with `perf`. The metrics obtained are lower than estimated because profiling starts at later point compared to the execution of `processRequests` by the parent process. The execution time of `processRequests` increases by 30% (330ms vs 429ms) as forking a child process triggers the copy-on-write mechanism that causes page faults and TLB flushes in the parent. Additional overhead is imposed by `fork`, `kill`, and `waitpid`, which increases the total execution time for a batch by 37% relative to baseline (330ms vs 452ms).

4. PROFILING AT THE μs SCALE

In this section, we evaluate the accuracy and overhead of PAPI and PMU-metrics as we reduce the granularity of profiling to the μs-scale. We consider two applications – profiling a hash table (§4.1), and adaptive log processing with BLARE (§4.2).

4.1 Reducing the granularity of profiling

4.1.1 *Experimental setup*

We consider the example of profiling a hash table as shown in Listing 1. We decrease the batch size of requests from 10M to 100, which decreases the granularity of profiling from 0.33s to 3.3μs. Total 100M fetch requests are processed, and we compare the median metrics of five runs in each case. For both the tools, the metrics measured at a coarse granularity of 10M requests are used as the baseline, i.e., the performance of a tool is compared to itself.

4.1.2 *Results*

Fig. 4a and 4b respectively show the accuracy and overhead of PAPI and PMU-metrics as we reduce the batch size. At *ms*-scale granularity (batch size 100k), both the tools remain consistent with their respective baselines, while at the μs-scale (batch size 100) we observe a 12% and 2% deviation in instructions reported by PAPI and PMU-metrics respectively. A similar trend is observed for other hardware metrics as well. We note that PAPI imposes a high overhead which increases the total running time by 268% at the μs-scale, while PMU-metrics imposes a low overhead of just 8%.

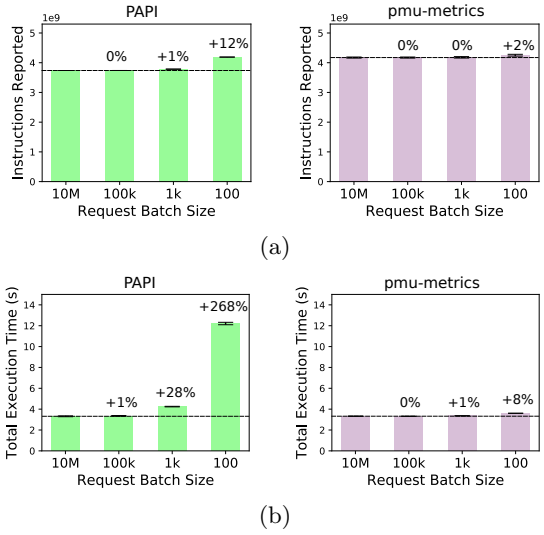

Figure 4: (a) Total instructions reported and (b) total execution time as we reduce the granularity of profiling from 10M requests ($\approx$ 0.33s) to 100 requests ($\approx$ 3.3μs). Total 100M fetch requests are processed in all cases. At μs-scale (batch size 100), PAPI and PMU-metrics show a deviation of +12% and +2% compared to their respective baselines, while imposing 268% and 8% overhead respectively.

Overall, we conclude that either libraries can be used at the *ms*-scale, while we recommend using PMU-metrics at the μs-scale as it imposes low overhead (8%) and maintains good accuracy (2% error).

4.2 Adaptive log processing with BLARE

4.2.1 *Experimental setup*

Regex matching is a frequently encountered task while processing unstructured log data. BLARE [16] uses an adaptive algorithm to choose an efficient strategy for regex evaluation at runtime. In a nutshell, BLARE uses a multi-armed bandit approach which involves measuring the "reward" for different attempts of a given strategy, which informs future attempts. We measure the reward as the number of core cycles elapsed for a given attempt. To this end, we employ three different methods for counting cycles – `perf_event` [5], PAPI (Listing 3), and PMU-metrics (Listing 4). We refer the reader to [3] for details of using `perf_event` for counting cycles. Experiments are run on a dataset [7] of size 100MB as detailed in [1]. Each experiment is repeated ten times, and we report the median metrics in all cases.

4.2.2 *Results*

Fig. 5 shows the running time of Blare with different cycle counting methods and the overhead compared to the best regex evaluation strategy. Un-

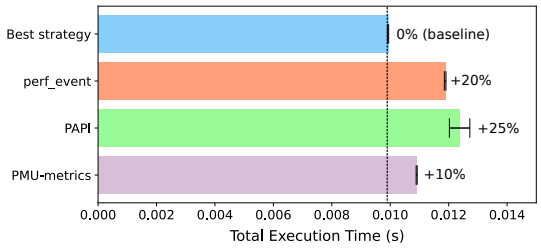

Figure 5: Running time of BLARE with different cycle counting methods. The overhead compared to the best strategy is indicated. Profiling with PMU-metrics attains a minimal overhead of 10% compared to `perf_event` (20%) and PAPI (25%).

surprisingly, PMU-metrics imposes the least overhead (10%) given that it utilizes an the `rdpmc` instruction to directly access performance counters. Cycle counting using `perf_event` (20% overhead) involves issuing system calls (`ioctl`) since performance counting is exposed via a file interface [5]. PAPI also utilizes the `perf_event` interface. However, it is a dynamically linked library, and imposes higher overhead (25%) compared to `perf_event`.

5. PAPI OR PMU-METRICS?

PAPI utilizes the PMU through the perf_event kernel library. Thus, the OS manages the hardware and automatically reprogrammes the PMU in the presence of context switches. However, the file descriptor interface of the perf_event library for accessing performance counters becomes expensive at the μs-scale (+268% execution time, Fig. 4b) and also affects the accuracy (12% deviation in instructions retired, Fig. 4a). On the other hand, PMU-metrics uses `rdpmc` to access performance counter registers, that enables the collection of metrics at μs-scale with good accuracy (2% deviation in instructions retired, Fig. 4a) and low overhead (+8% execution time, Fig. 4b). However, PMU-metrics directly programs the hardware, thus requiring privileged and exclusive access on part of the user.

6. CONCLUSIONS

Fine-grained profiling is an important requirement encountered by systems practitioners. Our analysis reveals that not all tools are suitable for this requirement. In particular, we discourage performing `fork`-based profiling using command-line tools such as `perf`, as it can result in incorrect metrics and impact the performance of the parent process due to the overhead of the copy-on-write mechanism. Tools designed for "first-person" fine-grained profiling such as PAPI and PMU-metrics should be preferred. Both of these libraries provide accurate metrics at the ms-scale, while we recommend using the lighter PMU-metrics library at the μs-scale.

7. ACKNOWLEDGMENTS

This research was supported by the National Science Foundation under grant OAC-1835446 and a David DeWitt Fellowship.

8. REFERENCES

[1] BLARE codebase. `github.com/mush-zhang/Blare/tree/main/original_codebase`.
[2] clock_gettime(3) – Linux manual page. `https://tinyurl.com/yyvkc2wz`.
[3] Counting CPU cycles with perf_event in C. `https://tinyurl.com/46azwvn6`.
[4] perf_event source code. `https://tinyurl.com/2bc557nj`.
[5] perf_event_open(2) – Linux manual page. `https://tinyurl.com/29f64vsm`.
[6] RDPMC – Read Performance-Monitoring Counters. `https://tinyurl.com/6rc495ud`.
[7] US Accidents dataset (2016-2019). `https://tinyurl.com/2n4rv5cd`.
[8] Use Linux's high resolution clock – clock_gettime. `https://tinyurl.com/3emmhdm5`.
[9] WRMSR – Write to Model Specific Register. `https://tinyurl.com/nfund4fe`.
[10] The PMU-metrics Library. `https://github.com/UWHustle/pmu-metrics`, 2022.
[11] A. C. De Melo. The New Linux perf tools. In *Slides from Linux Kongress*, volume 18, 2010.
[12] A. Kakaraparthy, J. M. Patel, B. P. Kroth, and K. Park. VIP Hashing – Adapting to Skew in Popularity of Data on the Fly (extended version). arXiv, 2022.
[13] T. Kersten, V. Leis, A. Kemper, T. Neumann, A. Pavlo, and P. Boncz. Everything You Always Wanted to Know about Compiled and Vectorized Queries but Were Afraid to Ask. *Proc. VLDB Endow.*, 11(13), 2019.
[14] T. Mühlbauer, W. Rödiger, R. Seilbeck, A. Reiser, A. Kemper, and T. Neumann. Instant Loading for Main Memory Databases. *Proc. VLDB Endow.*, 6(14), 2013.
[15] D. Terpstra, H. Jagode, H. You, and J. Dongarra. Collecting Performance Data with PAPI-C. In *Tools for High Performance Computing 2009*. Springer Berlin Heidelberg.
[16] L. Zhang, S. Deep, A. Floratou, A. Gruenheid, J. M. Patel, and Y. Zhu. Exploiting Structure in Regular Expression Queries. *Proc. ACM Manag. Data*, 1(2), June 2023.