Diagnosing Performance Fluctuations of High-throughput Software for Multi-core CPUs

May 25, 2018, ROME'18@Vancouver

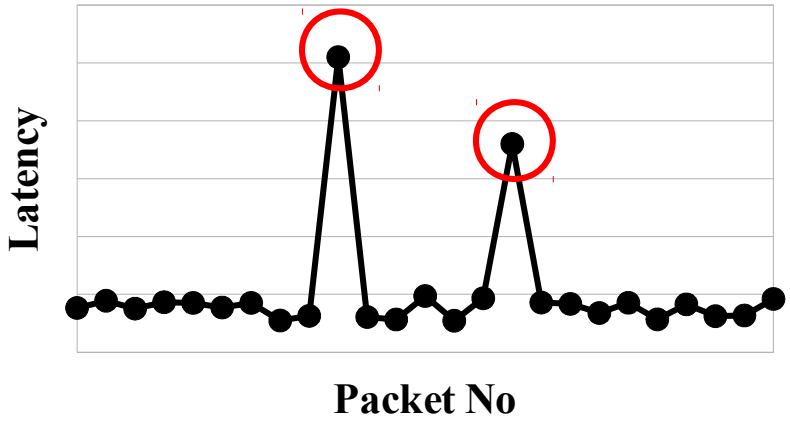
Soramichi Akiyama, Takahiro Hirofuchi, Ryousei Takano
National Institute of Advanced Industrial Science and Technology (AIST), Japan
{s.akiyama, t.hirofuchi, takano-ryousei}@aist.go.jp

Performance Fluctuation

- Performance of high-throughput software
  - Latency of SQL queries on a DBMS (mils of queries/s)
  - Throughput of software networking stack (100s Gbps)
- **Fluctuates for similar of even identical data-items**

*data-item := {query, packet, request}

  - TPC-C: standard deviation is twice the mean (*1)
  - Software-based packet processing: throughput drops by 27% in the worst case (*2)
- Large impact on usr experience



(*1) “A top-down approach to achieving performance predictability in database systems”, SIGMOD’17
(*2) “Toward predictable performance in software packet-processing platforms”, NSDI’12

Causes of Performance Fluctuation

- Cache-warmth
  - The first data-item may take more time than others
- Implementation design
  - Optimizing for the averaged may enlarge tail latency
- Resource congestion
  - Depending on how co-located workload uses competing resources

Performance fluctuations occur due to non-functional states of high-throughput software

Difficulty of Diagnosing Fluctuation

- Fluctuations occur in a complex set of non-functional states of the target software
  - May appear only in a production run / a compound test
- Reproducing non-functional states into a control environment is Infeasible
  - Cannot be quantified easily
  - May change frequently
  - Pinpointing a specific state as the root cause before solving the problem is impossible

Need to diagnose fluctuations online with low overhead

Trace vs. Profile

- Profile: Averaged view for a certain time period
- Trace: A list of performance event + timestamp

Profile

Func-tion	Total Time
A	250 us
B	100 us
C	50 us

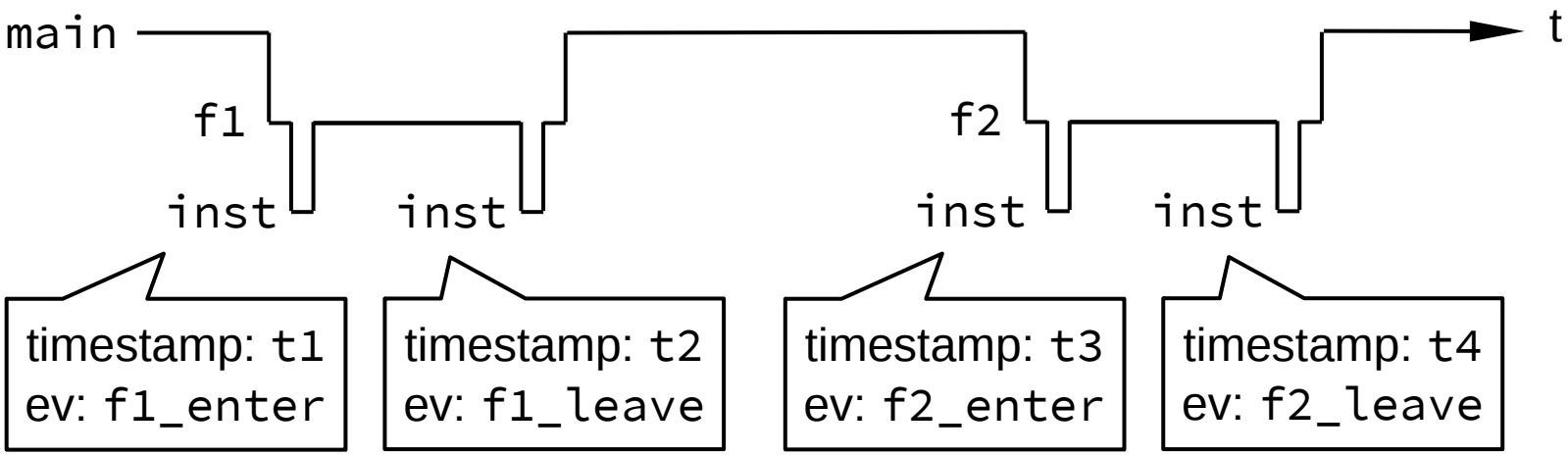
Trace

Request	Func-tion	Event	Time-stamp (us)	
#1	A	Enter	00010	90 us
#1	A	Leave	00100	
#2	A	Enter	00145	10 us
#2	A	Leave	00155	
...	...	...	...	
#50	C	Enter	04918	
#50	C	Leave	04923	

Per-data-item traces are promising to help diagnosing performance fluctuations, but profiles are not useful

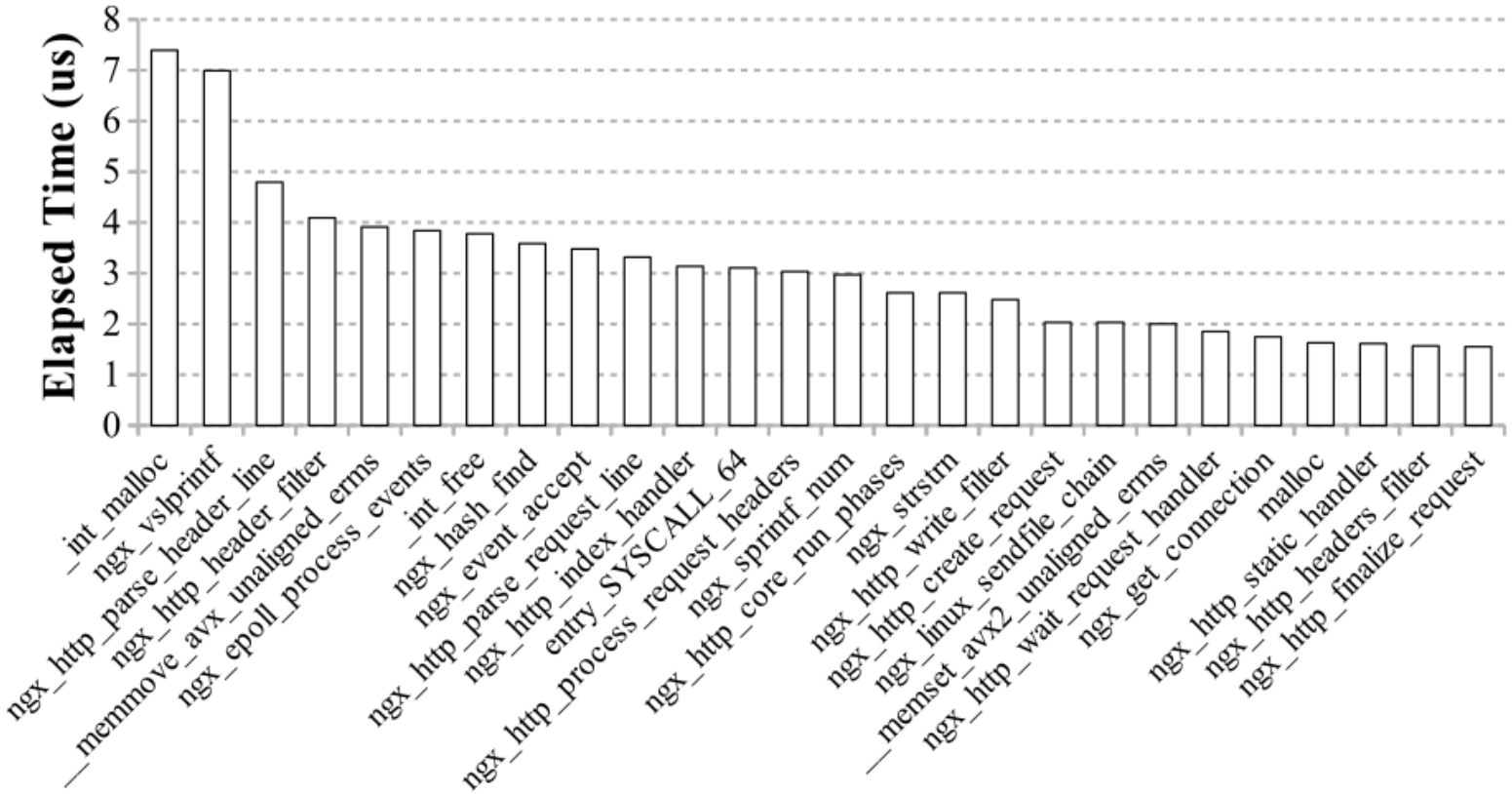
Obtaining Traces: Challenge (1/2)

- Software-based mechanisms to obtain traces
  - Instrumentation at the head and the end of a function to record traces
  - Typical implementation: insert special function calls
  - Examples: `gprof`, `Vampire`, `cProfile`



Obtaining Traces: Challenge (2/2)

- Functions in high-throughput software take a few micro seconds only



- NGINX serves the default index page (612 bytes)
- 1K requests sent simultaneously
- # of cycles for each function is measured by `perf`
- **A lot of them take only a couple of µs**

Instrumenting every function is too heavy for our scenario

Hybrid Approach

- **Main Idea**: use instrumentation only when necessary, and use sampling in other places
- Software-based instrumentation and hardware-based sampling work complementary each other

	Sampling	Instrumentation
Implemented by	hardware	software
Overhead	low	high
Timing	periodic	per each data-item
Adjustable	yes	no
What to trace	pre-defined	software-controlled
Traced data includes	timestamp, instruction pointer	timestamp, data-item ID

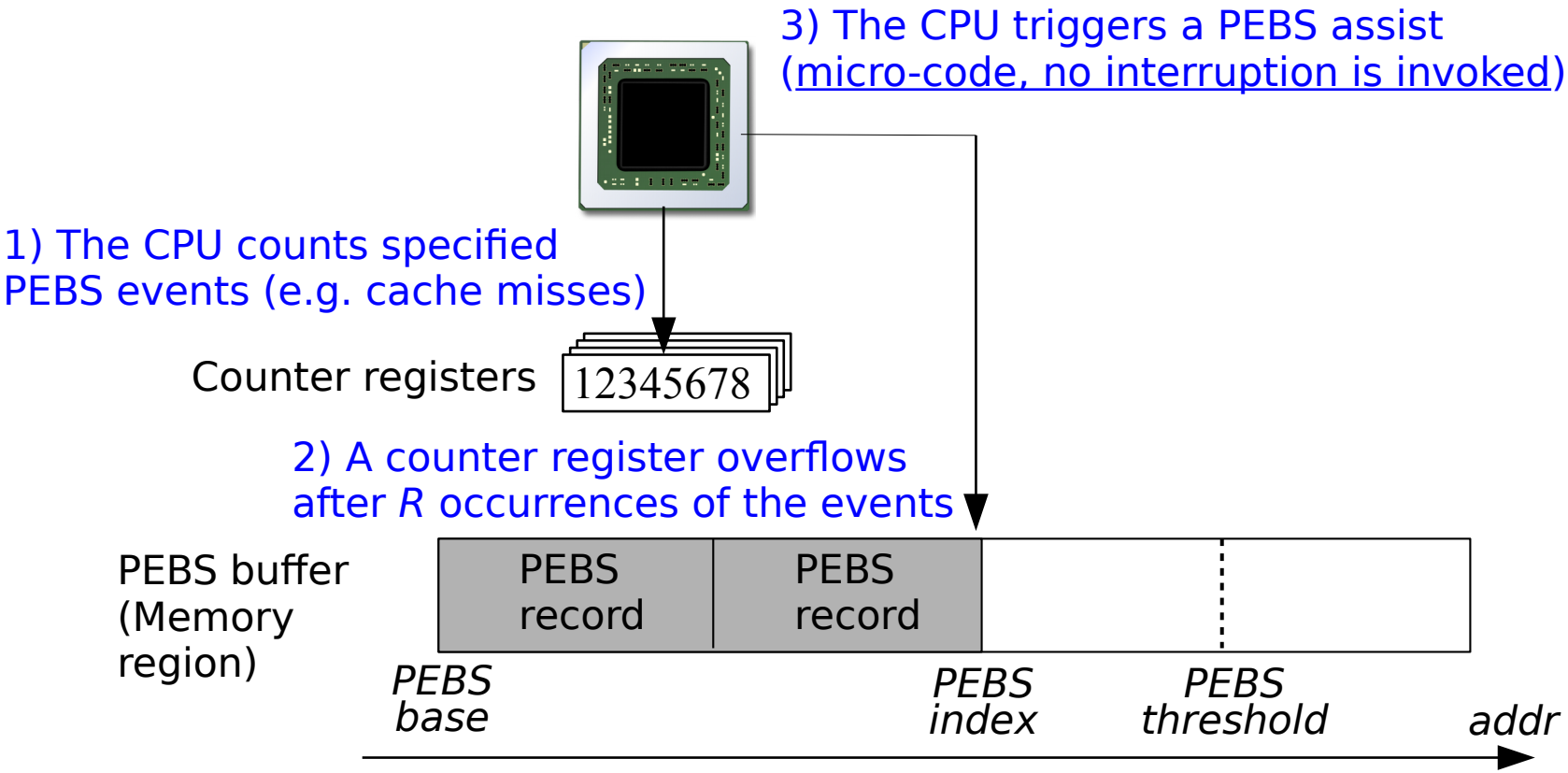
HW-based sampling: PEBS

- Precise Event Based Sampling (PEBS) is leveraged
  - Supported in almost any Intel CPUs
  - Enhancement of performance counters (counts hardware events and records program state at every *R* occurrences)
- PEBS is (almost) all hardware-based
  - Normal performance counters: OS records program states
  - PEBS: CPU (HW) records program states
- Pros: low overhead (less than 250 **ns** / *R* events) (*)
- Cons: can record pre-defined type of prg states

(*) “Quantitative Evaluation of Intel PEBS Overhead for Online System-Noise Analysis”, ROSS’17

How PEBS works

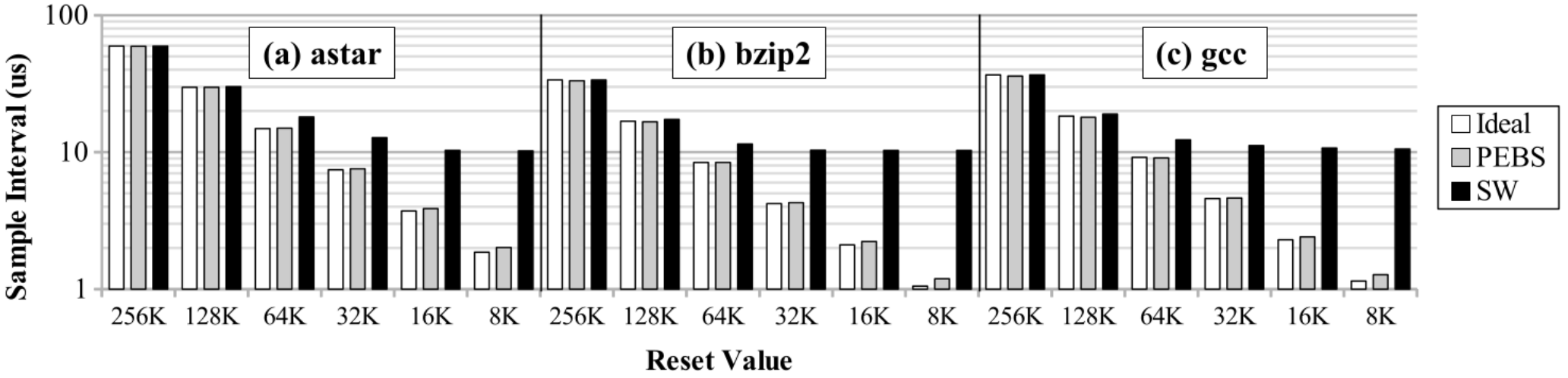
- Looks like normal performance counters, but (almost) everything is done by hardware



A PEBS record includes:
General purpose registers (eax, ebx, ..), Instruction Pointer (IP), Timestamp (tsc), Data LA, Load Latency, TX abort reason flag

PEBS vs. Software-based sampling

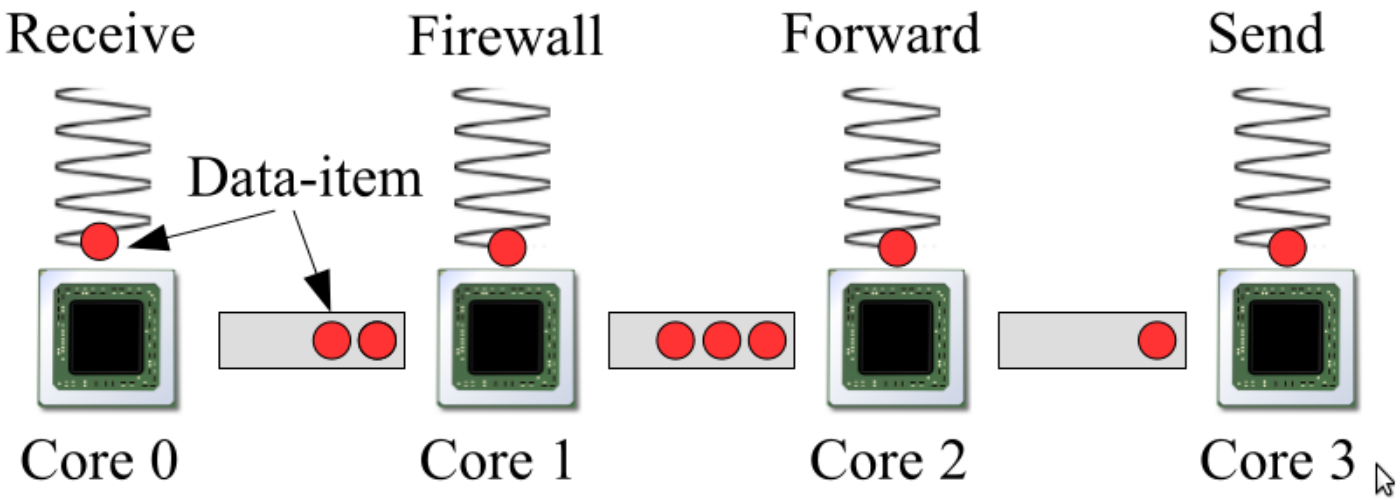
- Overhead of PEBS and normal (software-assisted) performance counters
  - *R* (Reset Value): a sample is taken every time the specified event occurs *R* times
  - Halving *R* results in the sample interval to be also halved, if there is no other bottleneck



PEBS is promising for our purpose while software-assited perf counters are not (Recap: functions to trace take a few second)

Mapping PEBS Data to Data-Items

- PEBS is low overhead, but only records pre-defined set of data (which includes no data-item ID)
  - Q: How to map each PEBS sample to a specific data-item?
  - A: Instrumentation only when target software starts processing a new data-item
- Modern high throughput software (NGINX, MariaDB, DPDK) process one data-item on a core at a time



Instrumentation in Our Approach

- Insert special function calls on *data-item switches*:
  - 1. The target software starts processing a new data-item
  - 2. It finishes processing a data-item
- Self-switching software architecture
  - Data-item switches explicitly written in the code to optimize for throughput → Instrument on these code points

```
while(1) {
    receive_data();
    do_something();
    more_work();
    blahblahblah();
    send_result();
}
```

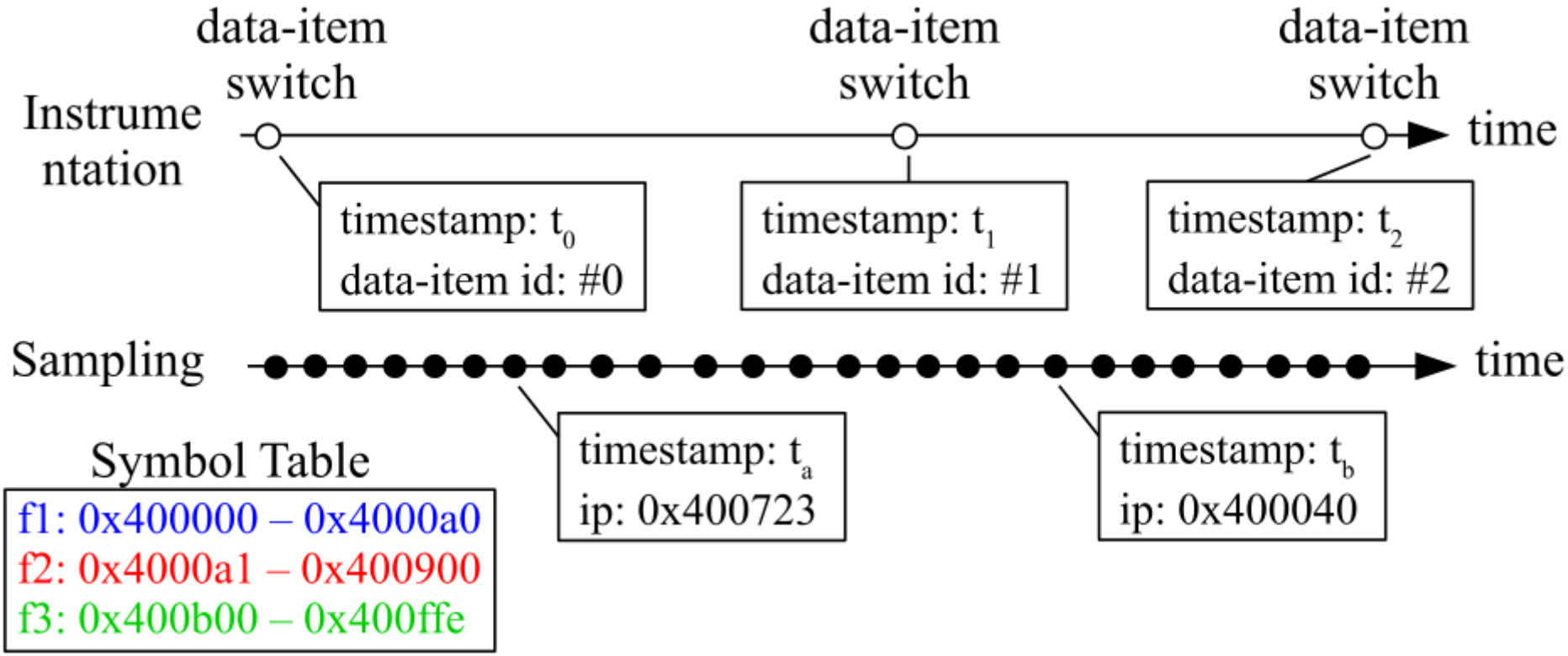
Data-item switch

Data-item switch

- Timer-switching software architecture (future work)
  - Additionally caused by timers to obey latency constraints

Proposed Workflow (1/2)

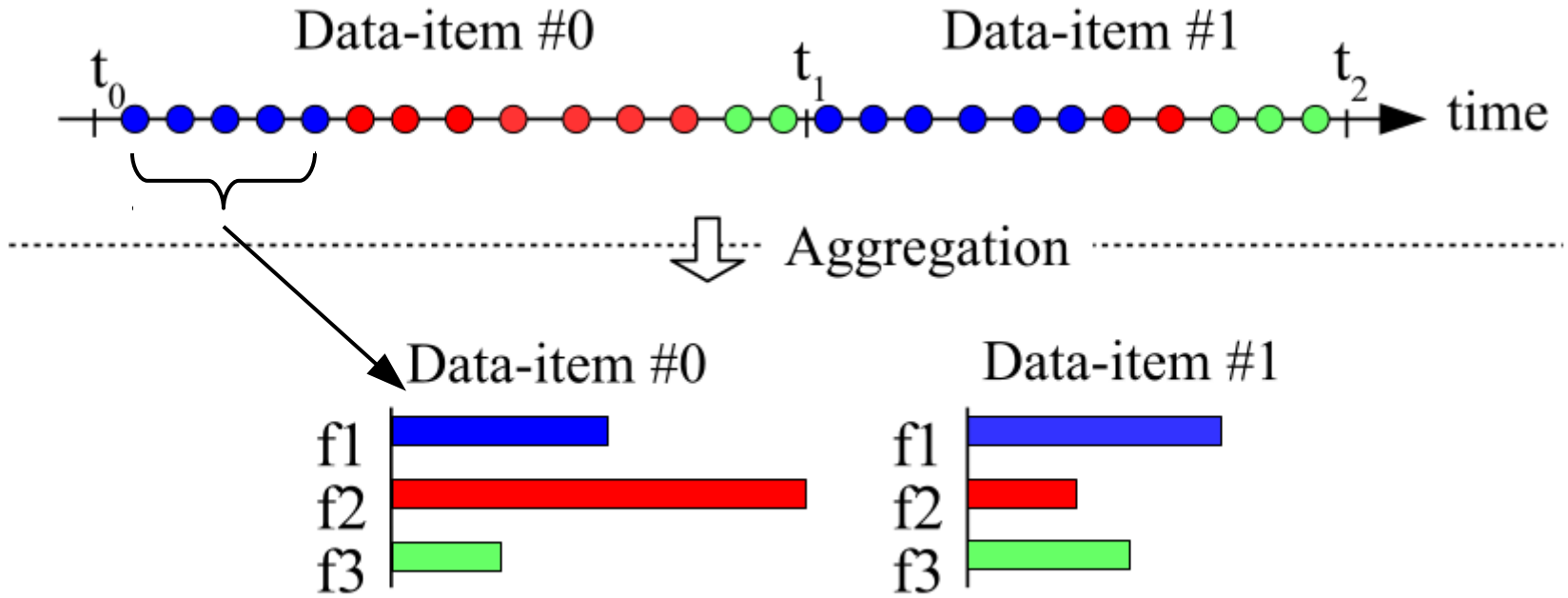
- Step 1: Data Recording
  - Instrument the code on data-item switches
  - Record timestamps and IPs using PEBS (`RETIRED_UOPS`)
  - Acquire the symbol table from the app binary



Proposed Workflow (2/2)

- Step 2: Data Integration
  - Map each PEBS sample to a {data-item, function} pair
  - Estimate the elapsed time for $\{d_i, f_i\}$ by:

Timestamp of the last record for $\{d_i,f_i\}$
– Timestamp of the first record for $\{d_i,f_i\}$



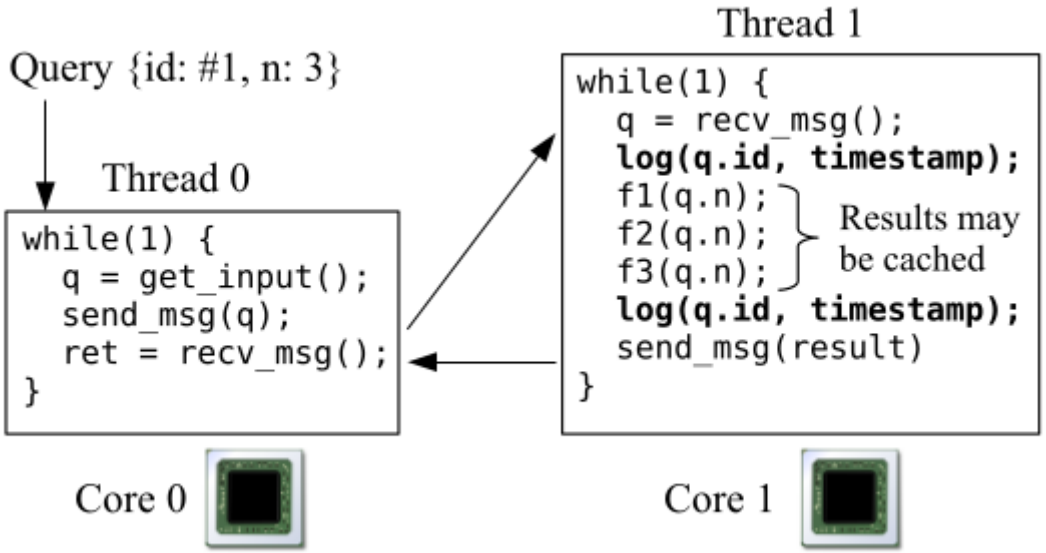
Evaluation

- Sample app
  - Input: query {id, n} → do some work on n data points, returns the results, and caches them
  - Latency fluctuates due to cache warmth
- DPDK-based ACL (access control list)
  - Input: packet → Judge if the packet should be dropped
  - Latency fluctuates due to implementation design
- Environment

CPU	Core i7 6700K (Skylake Micro arch.)
Motherboard	Supermicro X11SAE-F
OS	Debian GNU/Linux 8.9 (Linux kernel 4.9)
NIC	10 Gbps Intel X520-DA2 × 2
Memory	64 GB (16 GB DDR4 × 4)
SSD	512 GB (Crucial M4 CT512M4SSD2)

Sample Application (1/2)

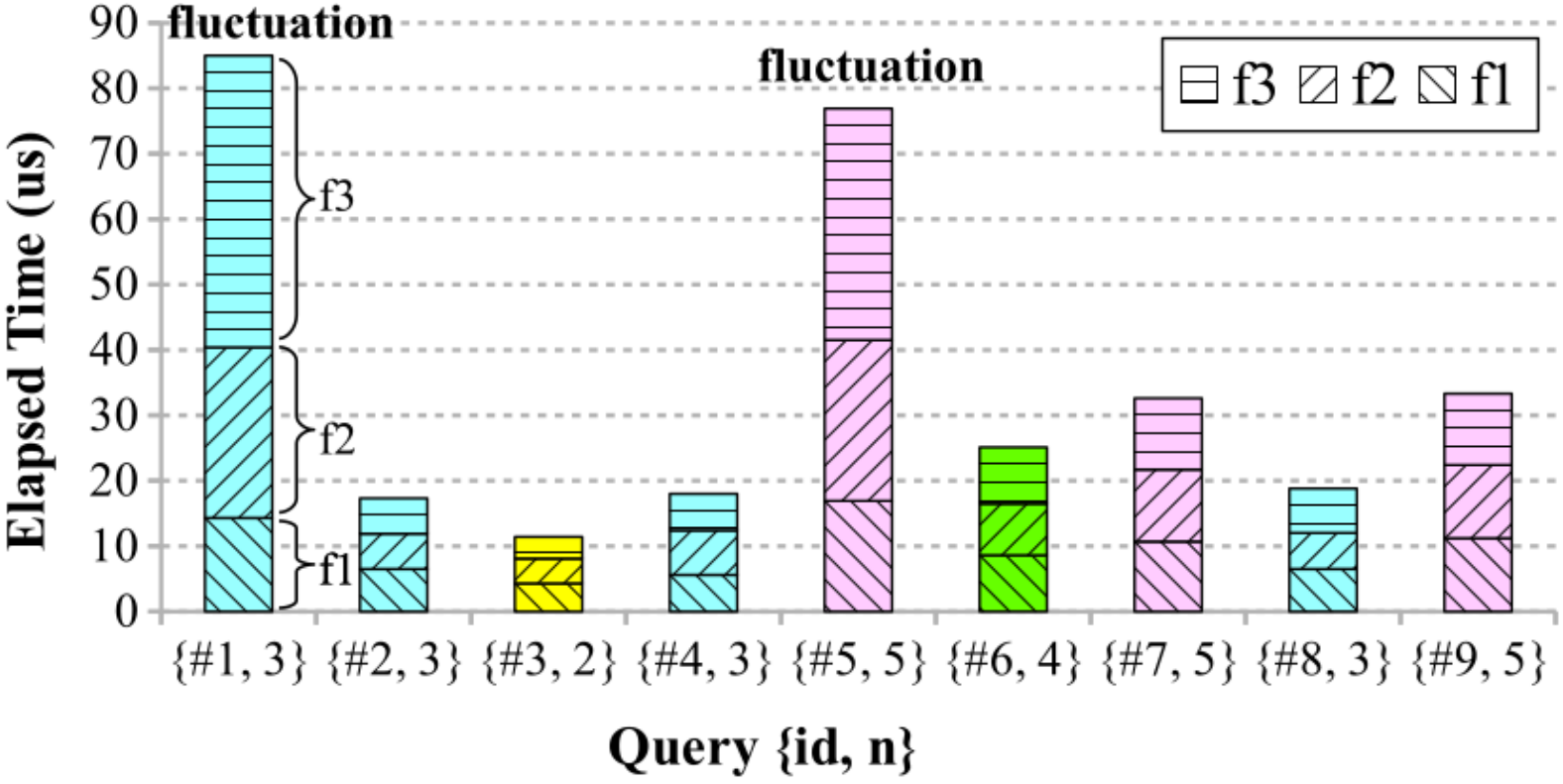
- Consists of two threads, pinned to two cores
  - Thread 0: receives queries and passes them to Thread 1
  - Thread 1: applies linear transformation to *n* points (Xi, Yi) and caches the results
- Instrumentation
  - Thread 1 switches data-items when (and only when) it finishes a query and start a new one



Latency of two identical queries differ due to different cache warmth

Sample Application (2/2)

- Fluctuations due to different cache warmth are clearly observed
- Function level information → useful to mitigate the fluctuation (cf. Query-level logging)



DPDK-based ACL (1/3)

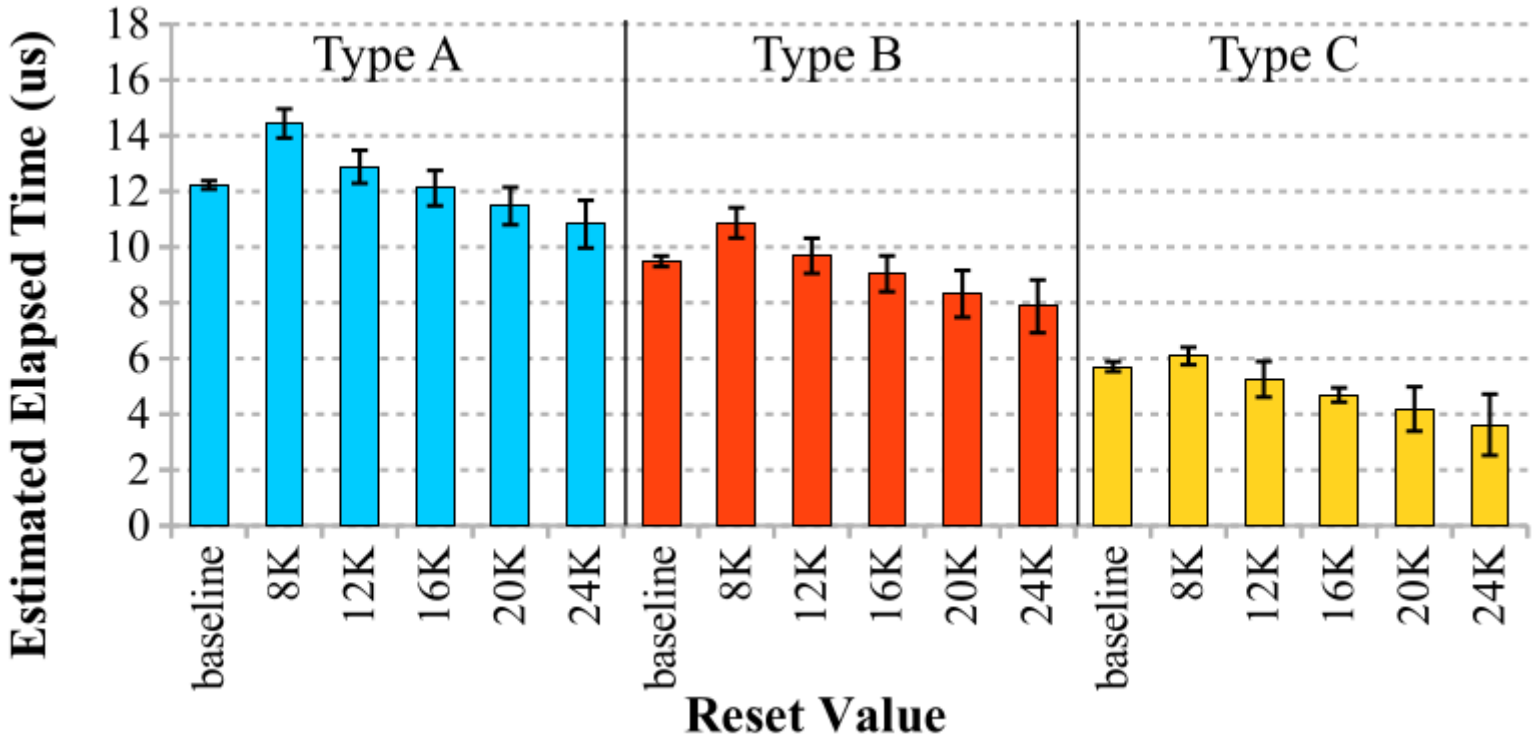
- Consists of three threads, pinned to three cores
  - RX/TX threads: receives packets / sends filtered packets
  - ACL thread: filters packets according to the rules
- Latency of very similar packets differ due to implementation design (details are in the paper)

Type	Src Addr	Dst Addr	Src Port	Dst Port	
A	192.168.10.4	192.168.11.5	10001	10002	slowest
B	192.168.10.4	192.168.22.2	10001	10002	
C	192.168.12.4	192.168.22.2	10001	10002	fastest

- Instrument `rte_acl_classify()` in ACL thread
  - Other threads are almost idle

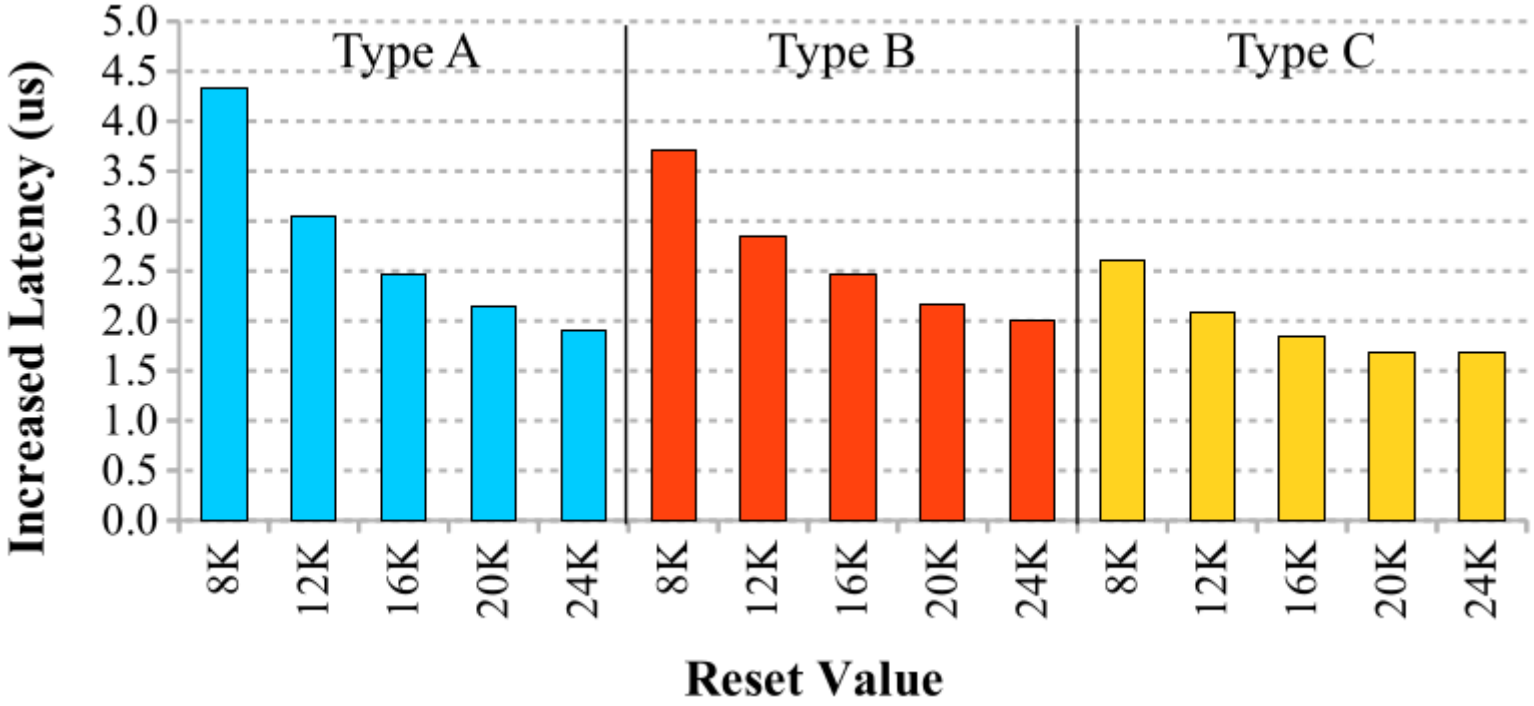
DPDK-based ACL (2/3)

- Baseline (ground truth): inserting logs before and after `rte_acl_classify()`
- Fluctuations for different packet types are clearly and accurately observed



DPDK-based ACL (3/3)

- Overhead is reduced with larger reset values (== smaller sampling rates)
  - But reduces accuracy by nature
- A good balance is required (see the paper for more discussion)



Related Work

- Blocked Time Analysis (*1)
  - Instrument Spark by adding logs → record how long time a query is blocked due to IO
  - Need to specify which function to insert logs
- Vprofiler (*2)
  - Starts instrumenting form large functions and gradually refines the profile
  - Need to repeat the same experiments many times
- Log20 (*3)
  - Automatically find where to insert logs that is enough to reproduce execution paths, but not each data-item

(*1) K. Ousterhout *et al.*, “Making sense of performance in data analytics frameworks”, NSDI’15
(*2) J. Huang *et al.*, “Statistical analysis of latency through semantic profiling”, EuroSys’17
(*3) X. Zhao *et al.*, “Log20: Fully automated optimal placement of log printing statements under specified overhead threshold”, SOSP’17

Conclusions

- Performance fluctuations is a common and important problem
  - Tail latency matters a lot on user experience
- Diagnosing them is challenging
  - Must obtain traces to observe a single occurrence online
  - Instrumenting every single function is too heavy
- Hybrid approach
  - Light-weight sampling + Information-rich instrumentation
  - Can observe fluctuations on a real code base