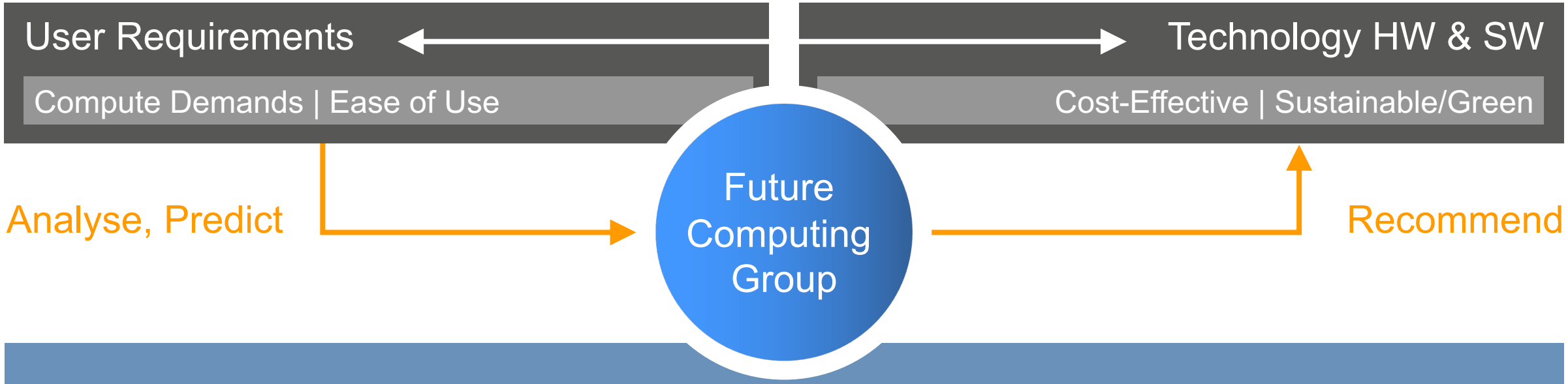
Always-on Application Introspection for Large HPC Systems

15th Parallel Tools Workshop | Sep 20, 2024 | Josef Weidendorfer

Work together with
Amir Raoofy, Michael Ott, Carla Guillien
Julian Scheipl

The Role of the FC Group



Understand best options – not just for the next system

Recommendations internally (for system purchase and operation) and externally (for supporting LRZ users)

Path to Best System for Users

1. Characterization of Application Mix on Current System
2. Estimation of future requirements (Artifical Intelligence, Big Data)
3. Identification of dominant compute kernels defining user requirements
4. Derivation of representative Benchmark-Suite (mix of micro-benchmarks, proxy-apps) for procurements
5. Ensure that benchmark suite is available for upcoming architectures (heterogenous, with accelerators)
6. Benchmark on recent architectures on-site
   (1) validate vendor claims, (2) understand usability, (3) check stability of SW stack

Context: LRZ User Base

- Jobs on SuperMUC NG
  - Around 750 research projects, Munich / Bavaria / German
  - Around 2000 researchers
- Some HPC community codes, but often codes written from scratch
- Top 5 codes only use 17% of CPU hours

For a good understanding
of the performance characteristics of the application mix,
we need **always-on background monitoring**

Motivation for Performance Characterization Monitoring

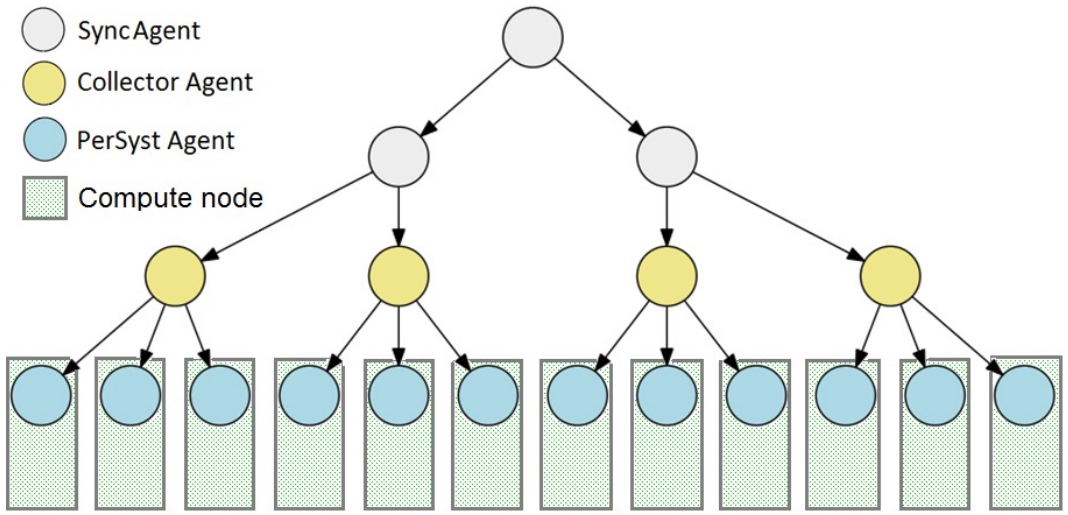
- Identify jobs with performance issues
  - High cache miss ratio, low Flop count, load imbalance
  - Allows to notify user about eventual waste of CPU budget
    - No enforcement of action
      (users already showed performance/scaling figures in project proposals)

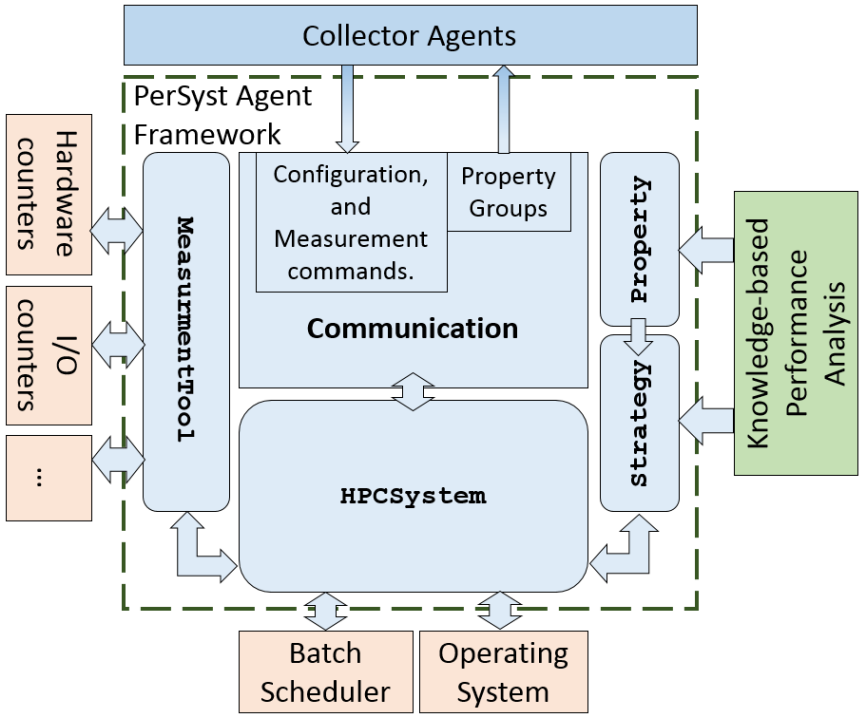
- Statistics to understand demand on resources
  - Focus for next system more on
    - High compute, fast caches, high memory BW, memory capacity, network, storage… ?
  - Embed this requirement in adequate benchmarks for next procurement

Solution (1): Performance Characterization

PerSyst

- Low-frequency sample collection of Performance Counters (every 10 minutes)
- Subset of counters for basic analysis of performance issues / resource demand
- Per-Job data provided to users as web page





PerSyst Visualization

Solution (2): Monitoring Infrastructure

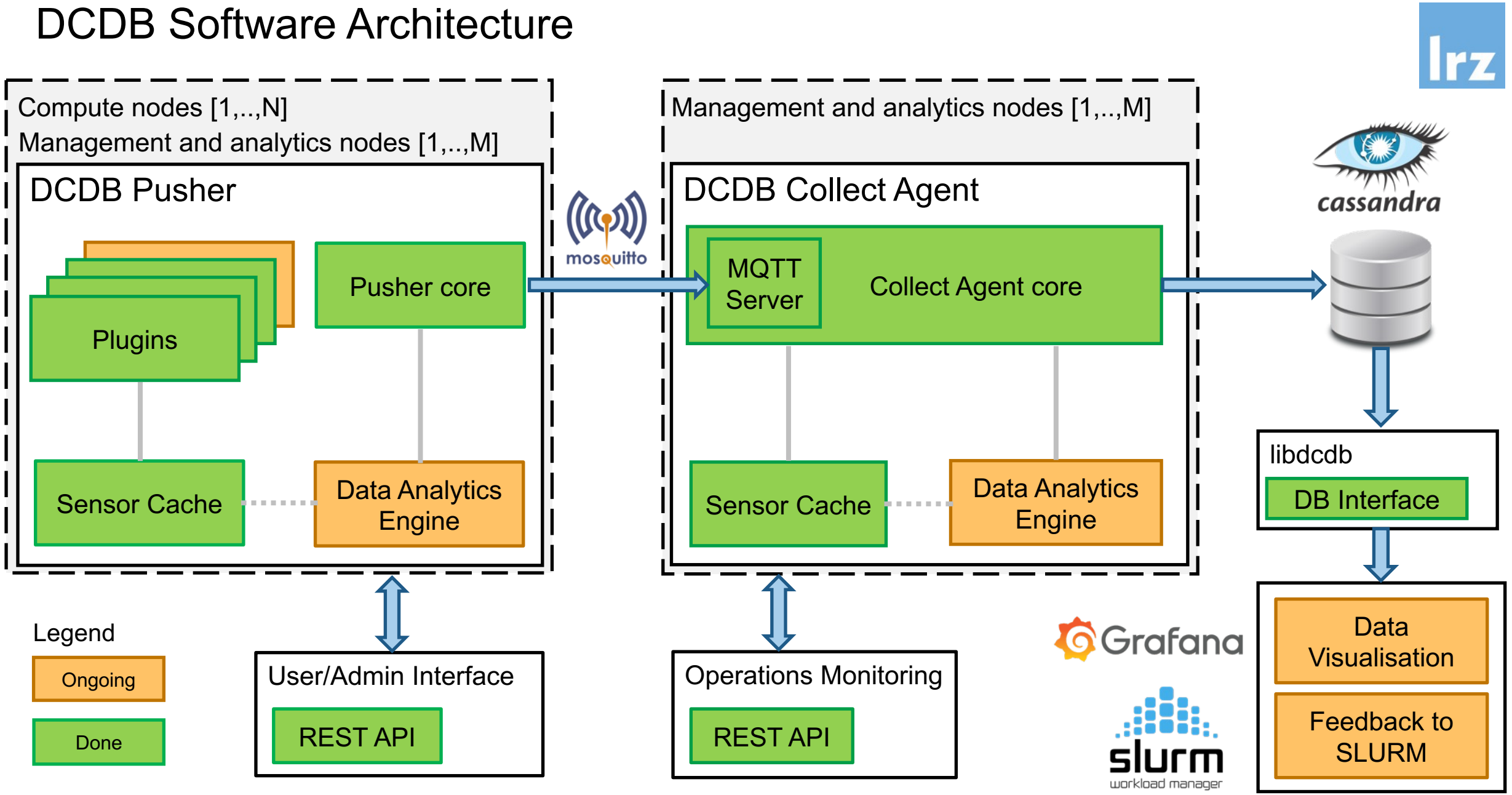
SuperMUC 1/2

- PerSyst used custom agent tree for aggregation

SuperMUC NG

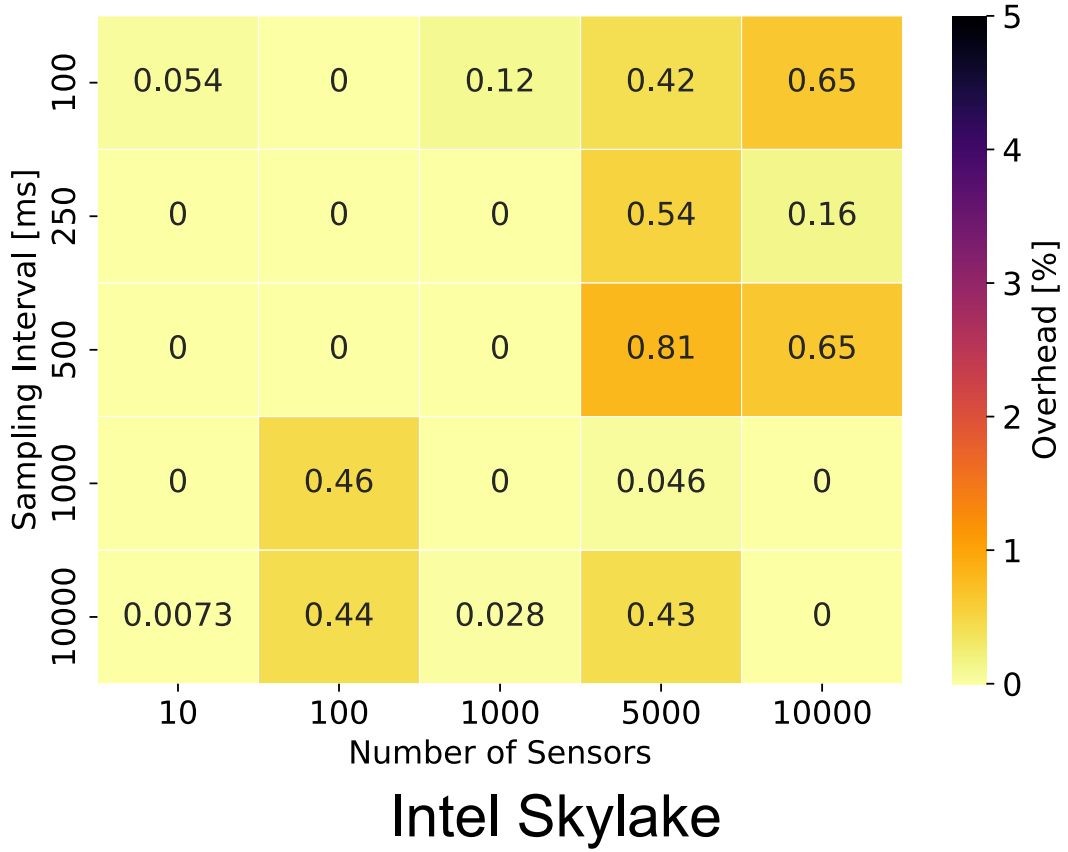
- DCDB: Data Center Data Base
- Developed by LRZ Energy-Efficiency Group within DEEPEST project
- Integrated solution for various monitoring needs
- For sensor data from building infra / cooling infra / HPC system HW / HPC SW …
- Supports sources from perf_events / {proc,sys}fs / GPFS / OPA / IPMI / SNMP / REST …
- Open source (GPLv3): http://dcdb.it
- PerSyst ported to DCDB

DCDB Software Architecture

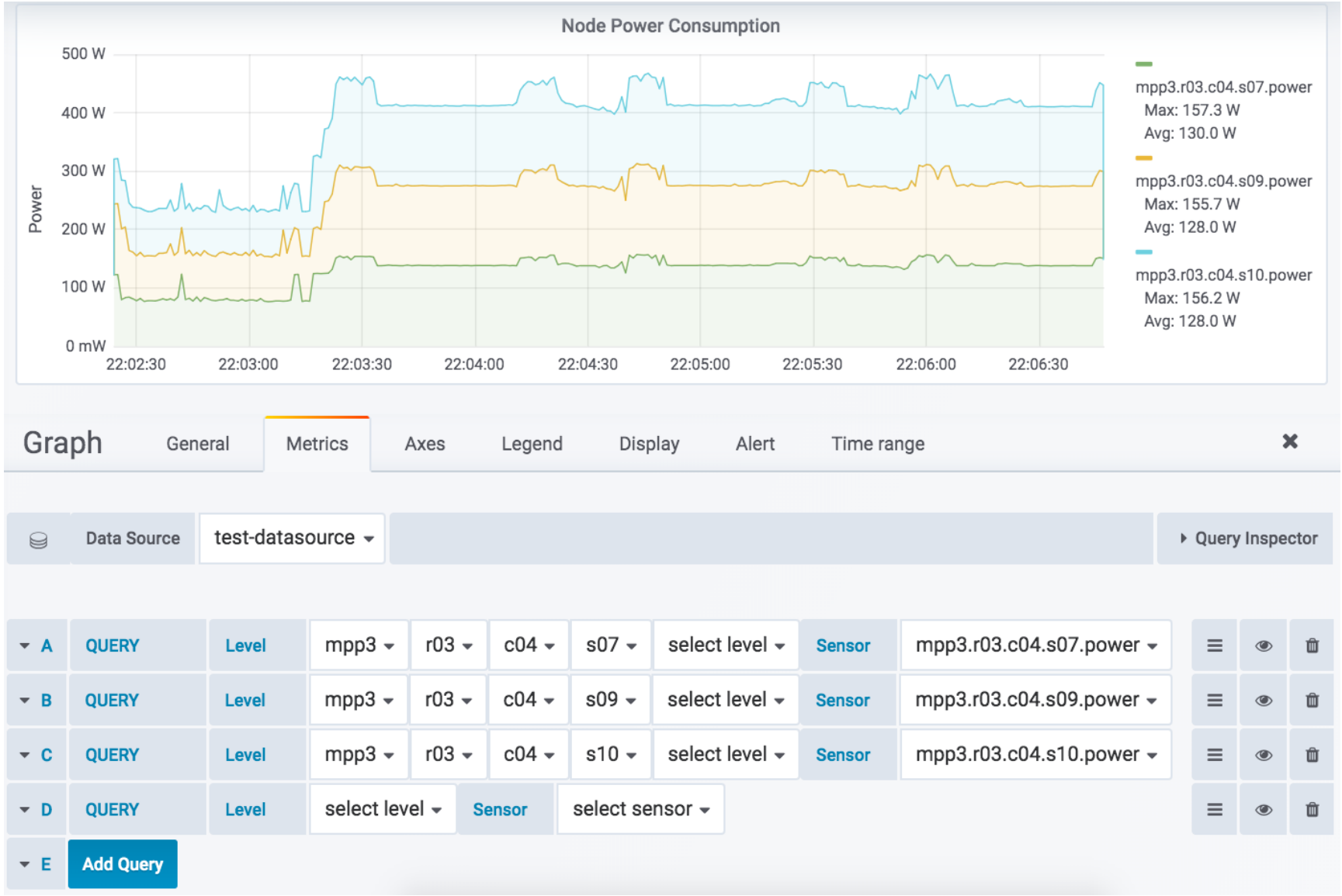


DCDB Overhead

Runtime overhead of DCDB core against High Performance Linpack on Intel Skylake



DCDB: Grafana Integration



We Want More Details!

Cannot pinpoint at source of performance issue / kernels with given characterization

Performance Issues

- Users are expected to use specific performance analysis tools
- More details allow to
  - Give suggestion to users fix the issue (better service)
  - Identify issues in common libraries where improvements would help everybody

Kernel Identification / Characterization

- Understand usage of installed software packages (get rid of unused packages)
- Help in designing smaller, still representative benchmarks for next procurement
- Allow to estimate porting efforts required towards new architectures / accelerators

Approach: System-Wide Program Counter Sampling

Goal

- Answer questions like “time spent in vendor-provided linear algebra library?”

Solution

- Get distribution of time spent by all jobs in binaries / shared libraries / waiting
- Time-based sampling of PC (1 Hz) with Linux perf_events, extension of PerSyst daemon
  - Own parsing of event stream from kernel with map / unmap / sample events with PC
  - Similar to “perf top -a” without resolving symbols (no DWARF info needed)

Low overhead

- 1 Hz enough for significant statistics on binary / shared lib usage
- Local aggregation into histogram, push to DCDB every few minutes

Pieces of the Solution

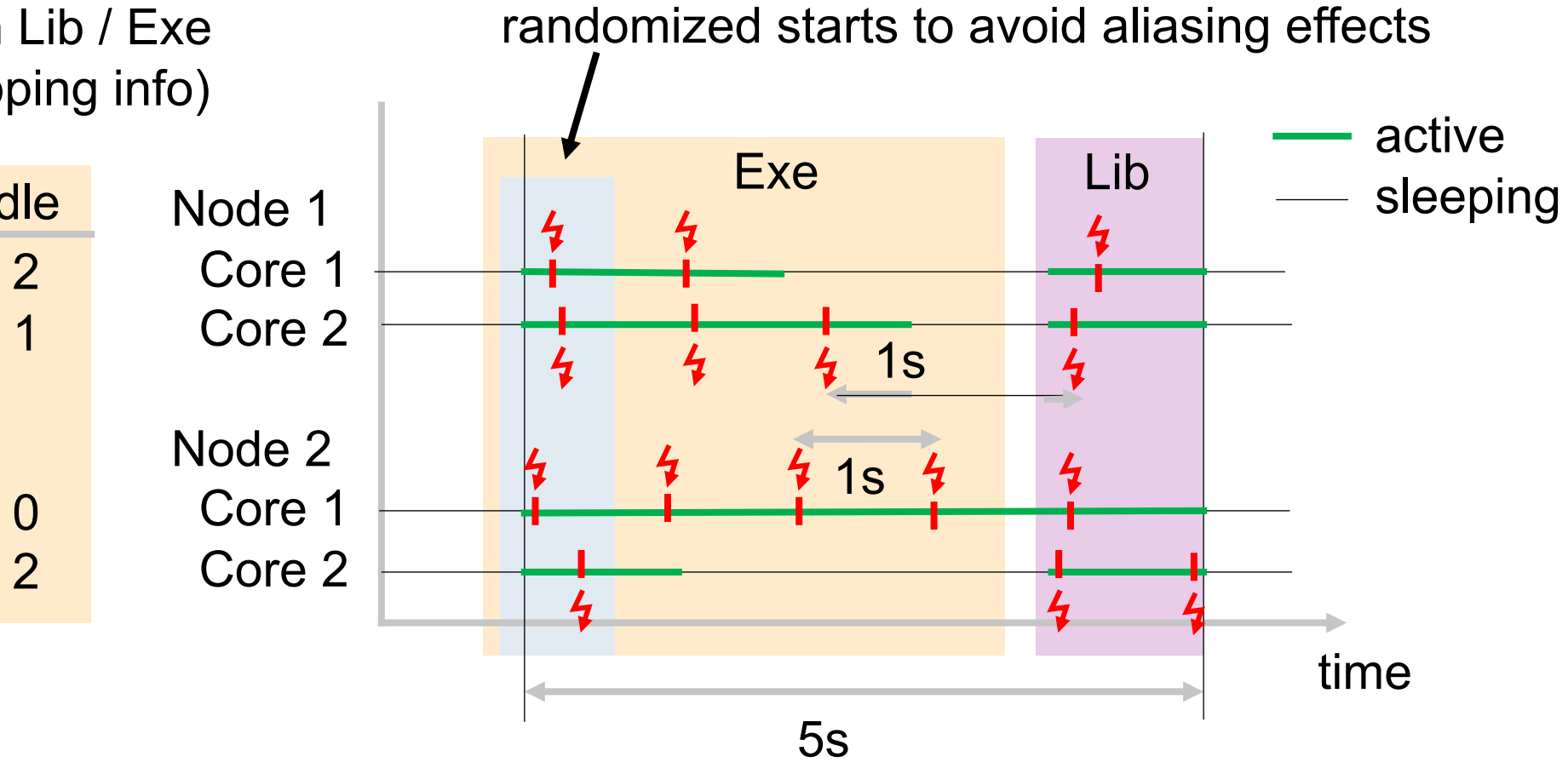
- Full System Sampling Mechanism
  - Linux Kernel-based Sampling (Perf)
- Catch User Knowledge: Instrumentation Library
  - Mark Phases, Relation Phases/Samples, Control of minimal Overhead
- Integration into System Monitoring
  - Collection, Aggregation, Time Series Database, Storage
- Postprocessing
  - Query Interfaces, Grafana, Export to User-Side Visualization Tools

Approach – Time Based Sampling

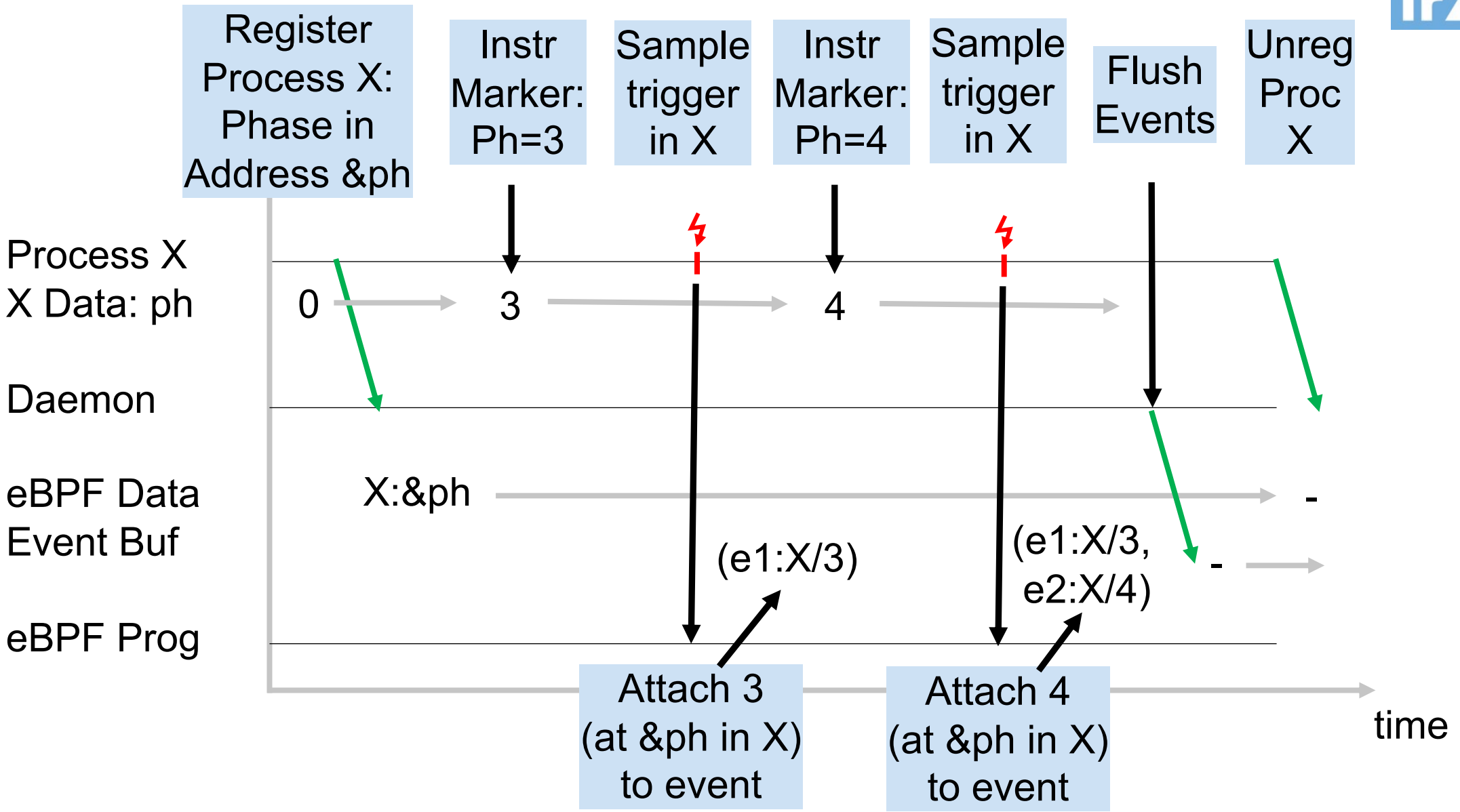
- System-wide, per core, using fixed „reference clock unhalted“
- Collected:
  PID, TID, IP ➔ Sh Lib / Exe
  (via mapping info)

to DCDB every 10 min

Exe	Lib	…	Idle
2	1		2
3	1		1
4	1		0
1	2		2



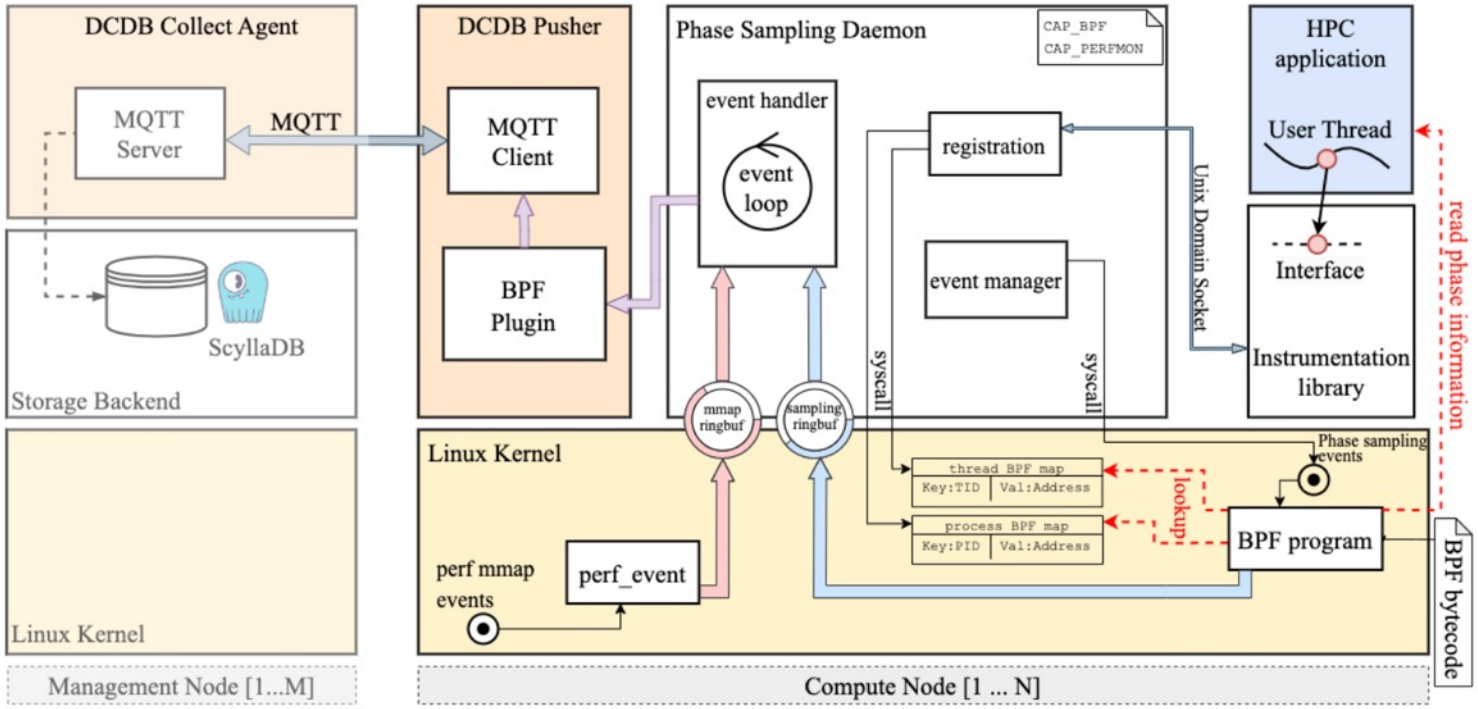
Approach – Marking Phases



eBPF Program

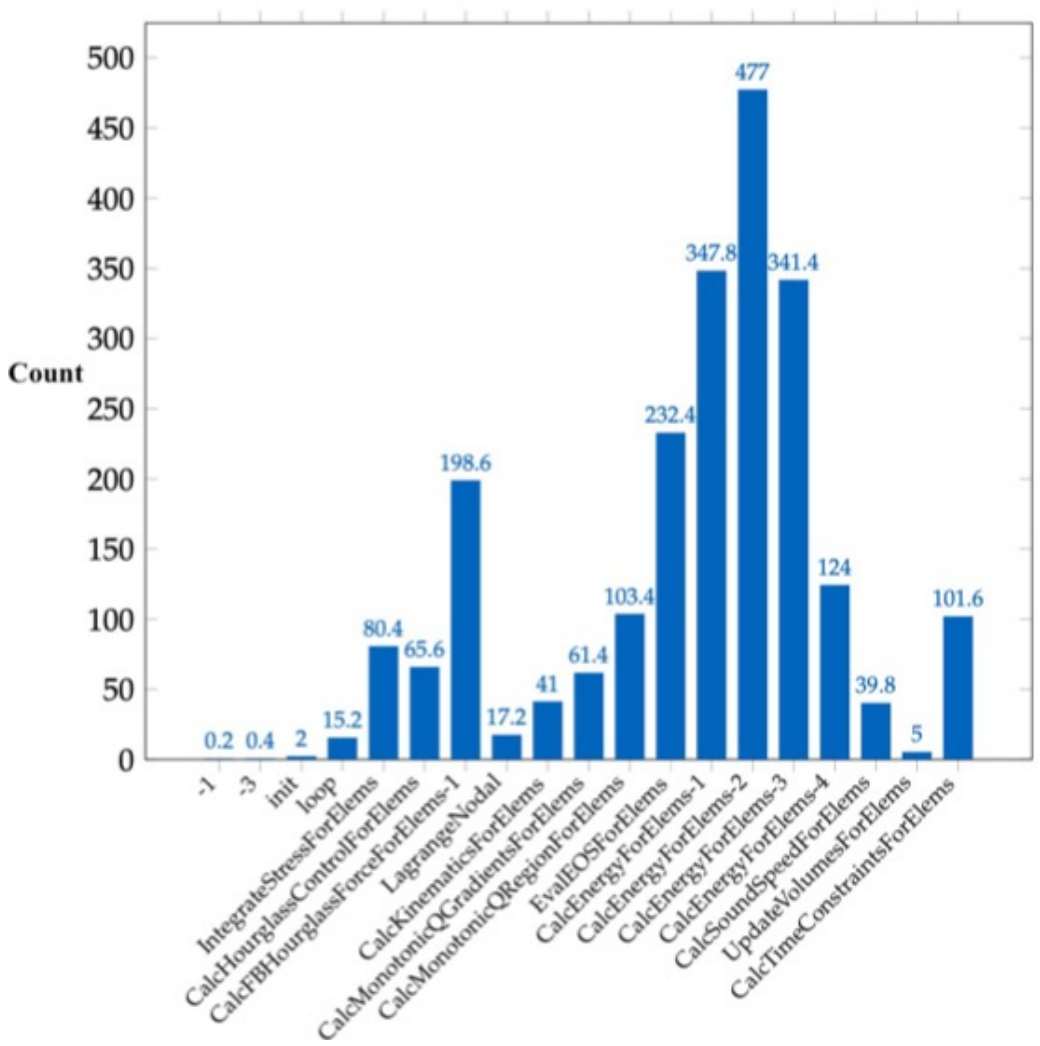
```
int** ret = bpf_map_lookup_elem((struct bpf_map*)&thread_map, &tid);
    if(ret == NULL) {
        ret = bpf_map_lookup_elem((struct bpf_map*)&process_map, &pid);
        if(ret == NULL) { // PID is not instrumented
            // place dummy value into return structure
        }
        // read global phase information
        bpf_probe_read_user(&e.phase_info, sizeof(phase_info_t), *ret);
    } else {
        // read thread-local pointer
        bpf_probe_read_user(&ret, sizeof(void*), *ret);
        // read phase information
        bpf_probe_read_user(&e.phase_info, sizeof(phase_info_t), ret);
    }
```

Architecture



[Poster CF23]

Output of Example: LULESH code with function markers



[BA thesis Scheipl]

Overhead

Must be neglectable: **always on**, running partly on **user budget** for LRZ purposes

How?

- sampling + collection: overhead controllable
- time spent for instrumentation
  - must be small for good statistics + not make users angry (their budget)
  - issue: user may put markers in inner loops

Solution

- instrumentation points can be de-activated
- dynamic check of overhead: if too high, de-activate!
  (1) visible in samples (2) counters + threshold

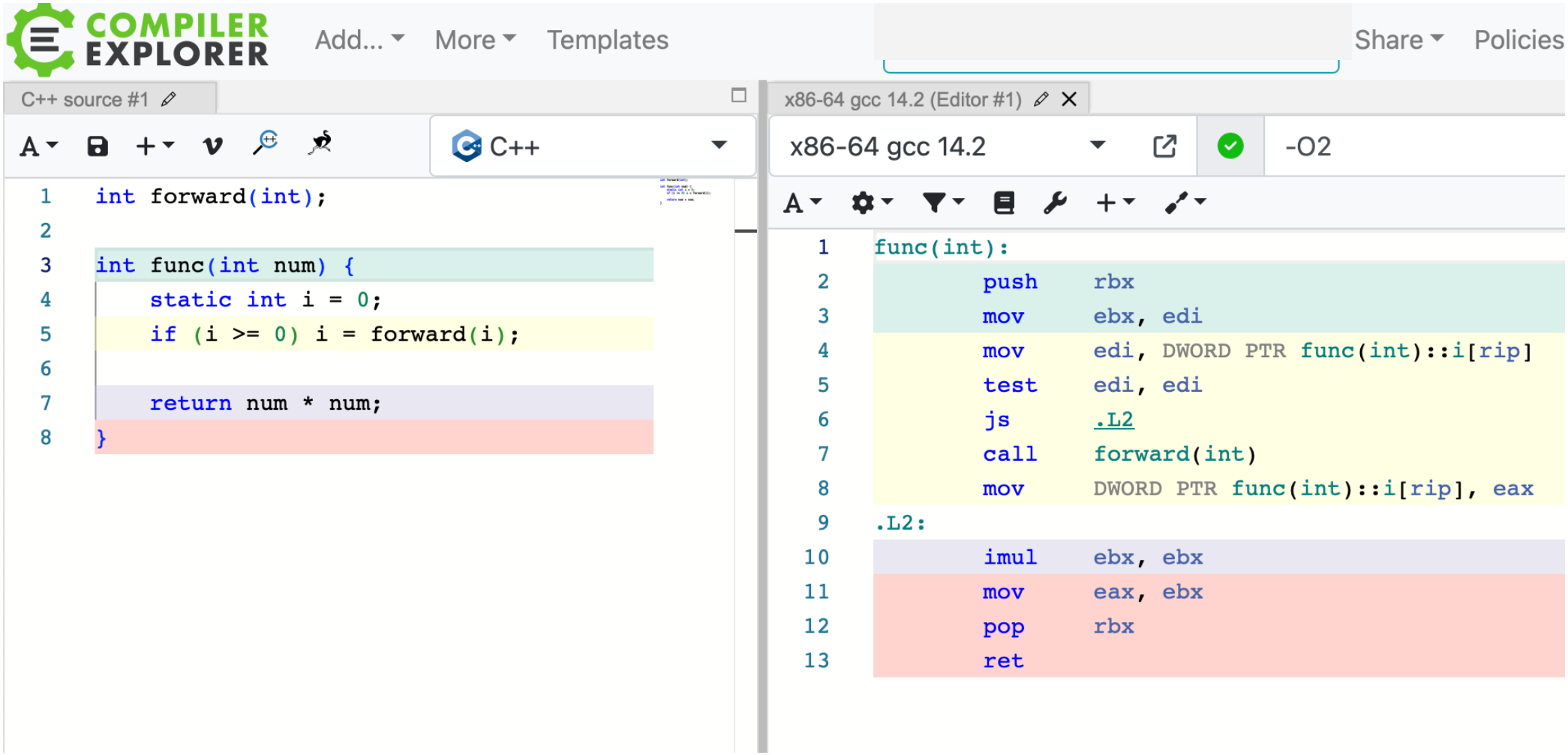
Marker Implementation: LIIF – leight weight instrumentation Interface

- C macro for inlining
- Registration on 1st use
  - pass static info
  - Notify about address of disable flag („id“)
- Regular use (id >= 0)
  - pass dynamic info
- Disabled state: id < 0
  - mem access + compare + branch (no reg on x86)
  - other option: DynInst with dynamic patching

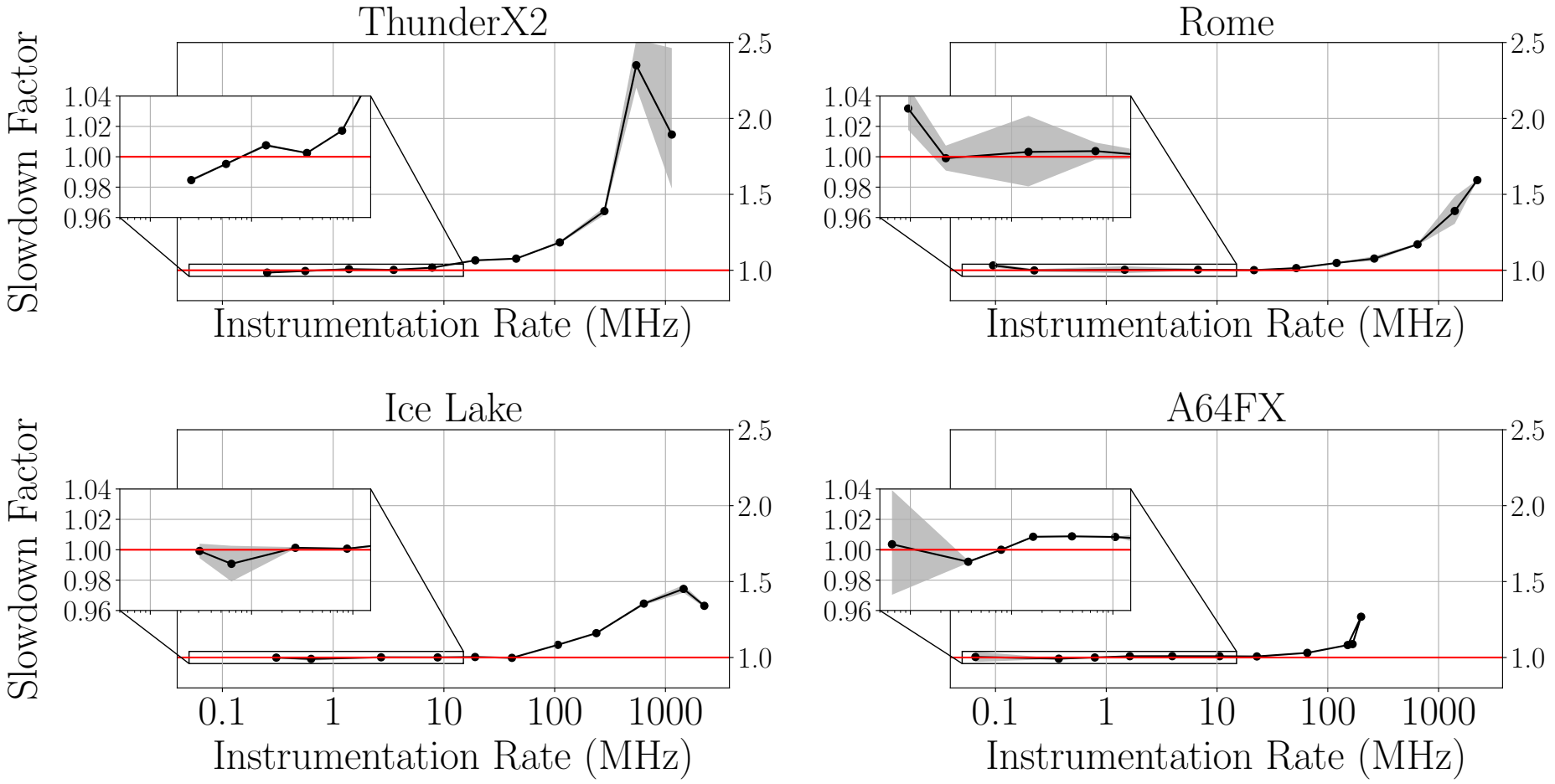
_Atomic

```
1   typedef void (*liif_fptr)(int id, ...);
2   #define LIIF_INST(type, name, format, ...)                \
3    { static int id = { 0 };                                 \
4      static liif_fptr fp = { 0 };                           \
5      if (id >= 0) {                                         \
6        if (!id)                                             \
7          fp = liif_reg(1, &id, LIIF_TYPE_ ## type,          \
8                        LIIF_MODULE, __func__, name, format); \
9        fp(id, __VA_ARGS__);                                 \
10     }                                                      \
11   }
12  #define LIIF_ENTER(name) LIIF_INST(ENTER, name, "")
```

Example: Code for Instrumention with Ability for Deactivate

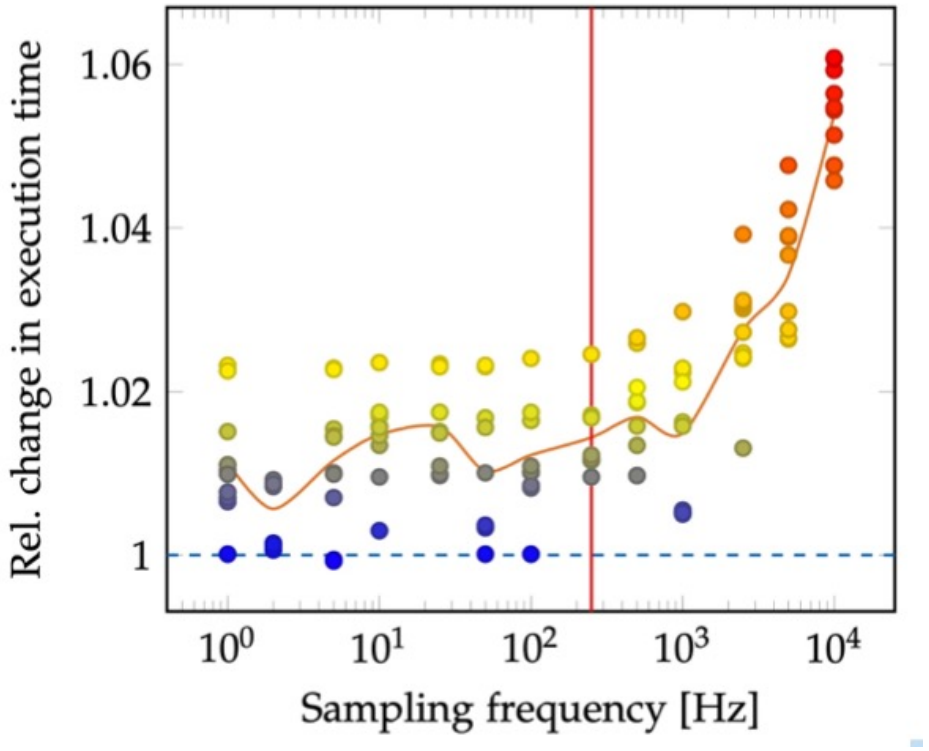


Overhead of De-activated LIIF Instrumentation



[Poster CF22]

Overhead of Sampling + Collection



Conclusion: Sampling for HPC Systems

- System-side monitoring for
  - operation: track utilization + tune operation + feedback to user
  - future procurement: user requirements → benchmark selection

Improved monitoring: Statistics with relation to Code

- Low-frequency sampling across full system (~ 1 Hz): capture relevant compute kernels
- User-provided phase markers: capture coarse-grained developer knowledge

Implementation

- Use sampling feature of „Performance Counters for Linux“
- Attach user-provided phase IDs to sample points via eBPF

Future Work

Sampling for more details on resource contention

- FLOP rates, memory bandwidth
- currently: “free running“ performance counters read every 10 mins
  - this only gives average usage across each 10 min interval
- idea via sampling: do 3 samples in a row to derive rates, attach to IP of middle sample

Other use of phase markers provided by users

- Guide energy-aware system tools: clock up/down when entering phase

„Application Mentors“ can use profiling results to detect issues, contact users

Questions ?