USC University of Southern California

Granular Resource Demand Heterogeneity

Yizhuo (Coulson) Liang, Ramesh Govindan, Seo Jin Park

HotOS' 25

Overview

- **Resource demand heterogeneity exists within applications**
  - Exploit resource-aware scheduling in finner granularity
- **Hiresperf: profile resource usage of each function invocation**
  - at 10 μs resolution, 7% ~ 15% overhead, or even lower
- **Future directions to exploit such fine-grained resource demand knowledge**

Resource Demand Heterogeneity:

Each application exhibits a distinct set of needs for **shared architectural resources**, such as

- DRAM Bandwidth
- Storage Bandwidth
- L3 Caches
- Network bandwidth
- Pipeline
- ...

Knowing Resource Demands Matters

Colocating apps in a single instance is common place, and apps may compete for the shared resources

Resource contentions

- slow programs down
- waste other resources
- hurt QoS & higher tail latency

Demand-Aware Placement & Scheduling

Existing methods:

- **runtime profiling or benchmarking before deploy**
- **schedule when and where to run programs to minimize contentions**

***e.g.* collocating a mem-bandwidth-intensive app with a more compute-heavy app**

Yet prior works focus on monolithic applications.

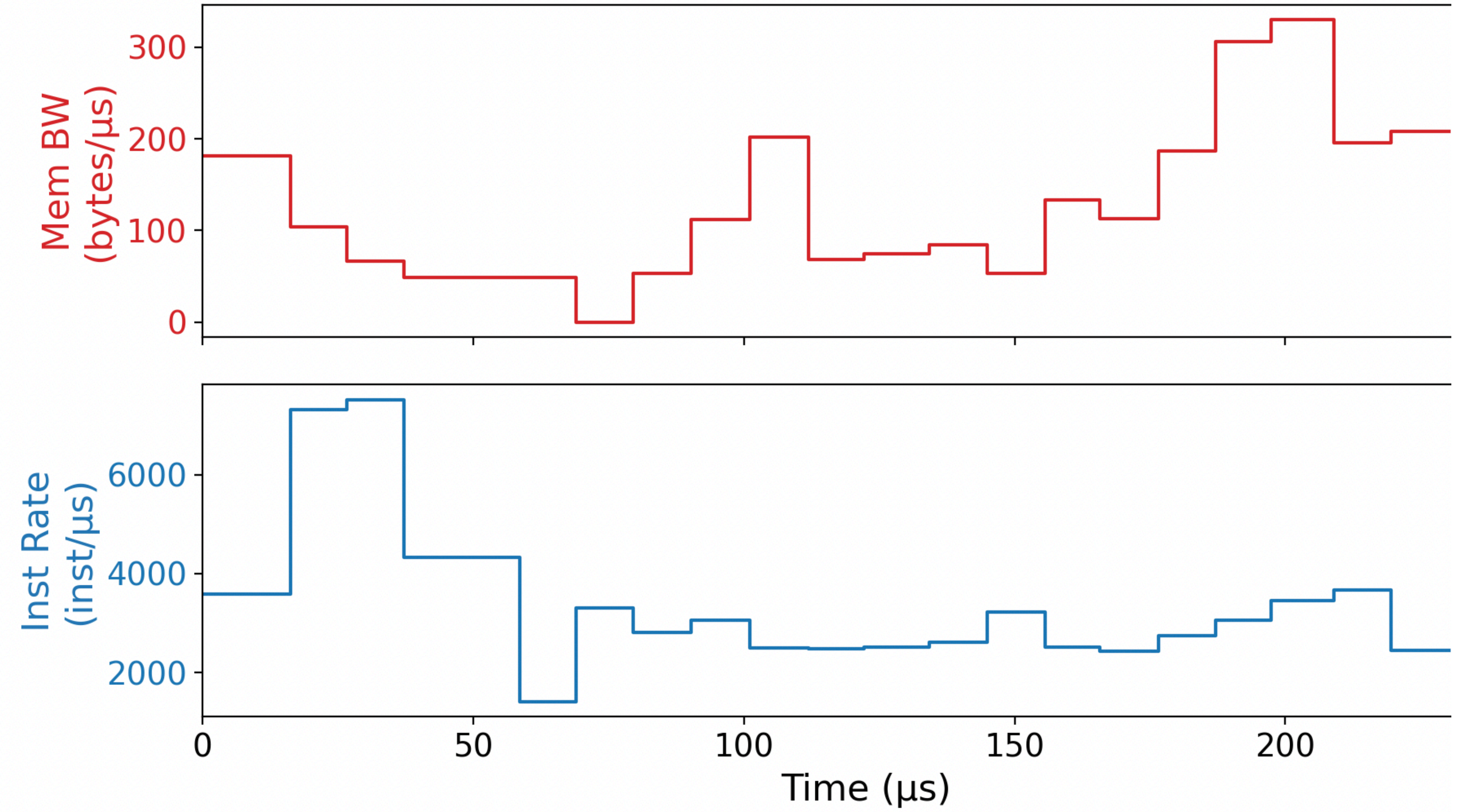
Demand Heterogeneity Within an App?

However, there are also heterogeneous resource needs within a single application

Demand Heterogeneity Within an App?

However, there are also heterogeneous resource needs within a single application

e.g. resource timelines of processing a Lucene++ search request



Demand Heterogeneity Within an App?

However, there are also heterogeneous resource needs within a single application

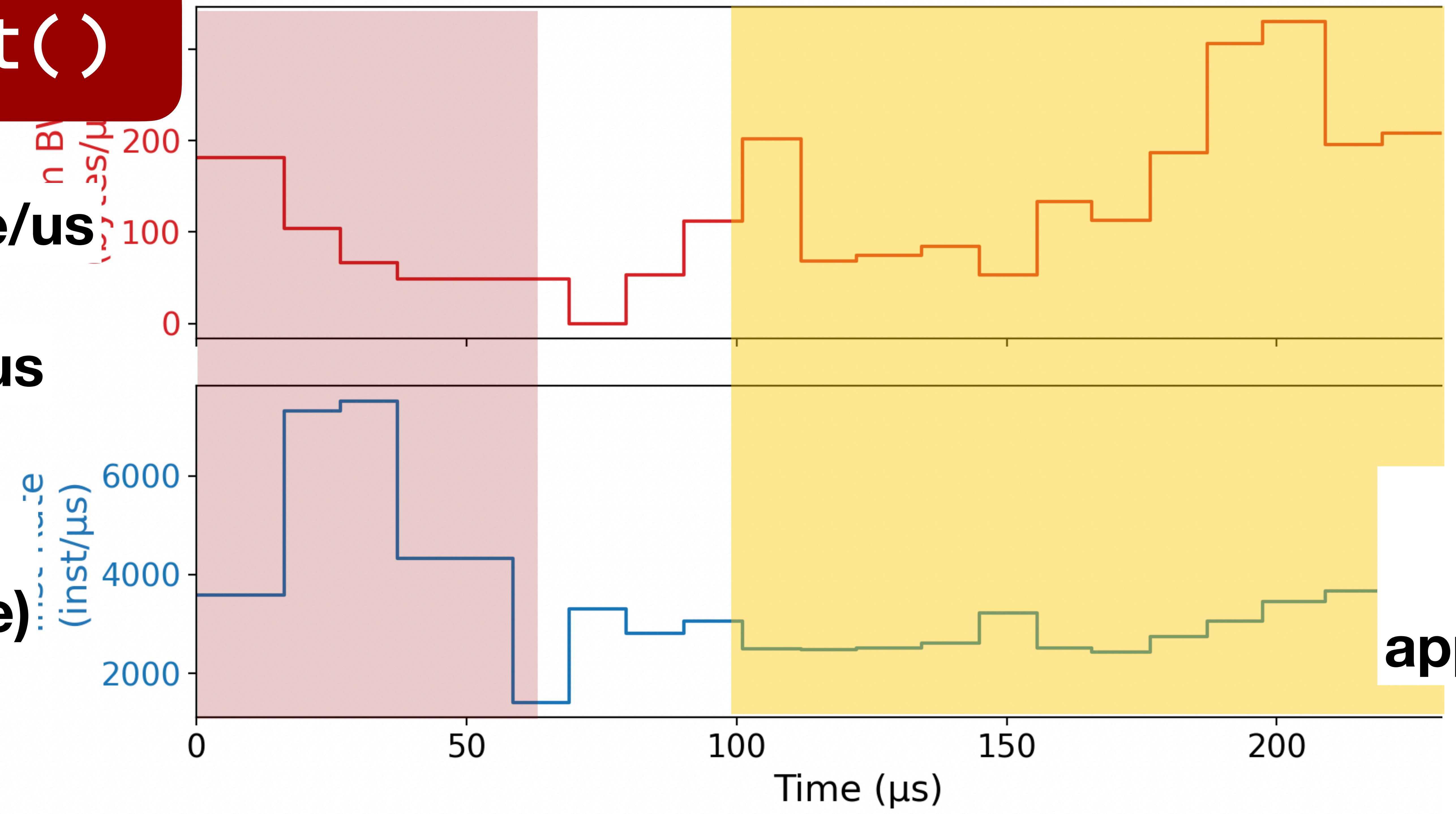
e.g. resource timelines of processing a Lucene++ search request

createWeight()

DRAM_BW: 117 Byte/us

INST_Rate: 4395 /us

Want better CPU (frequency & pipeline)



score()

DRAM_BW: 218 Byte/us

INST_Rate: 2909 /us

More mem channels, Colocate with other apps consuming less DRAM BW

Demand Heterogeneity Within an App?

However, there are also heterogeneous resource needs within a single application

e.g. resource timelines of processing a Lucene++ search request

`createWeight()`

`score()`

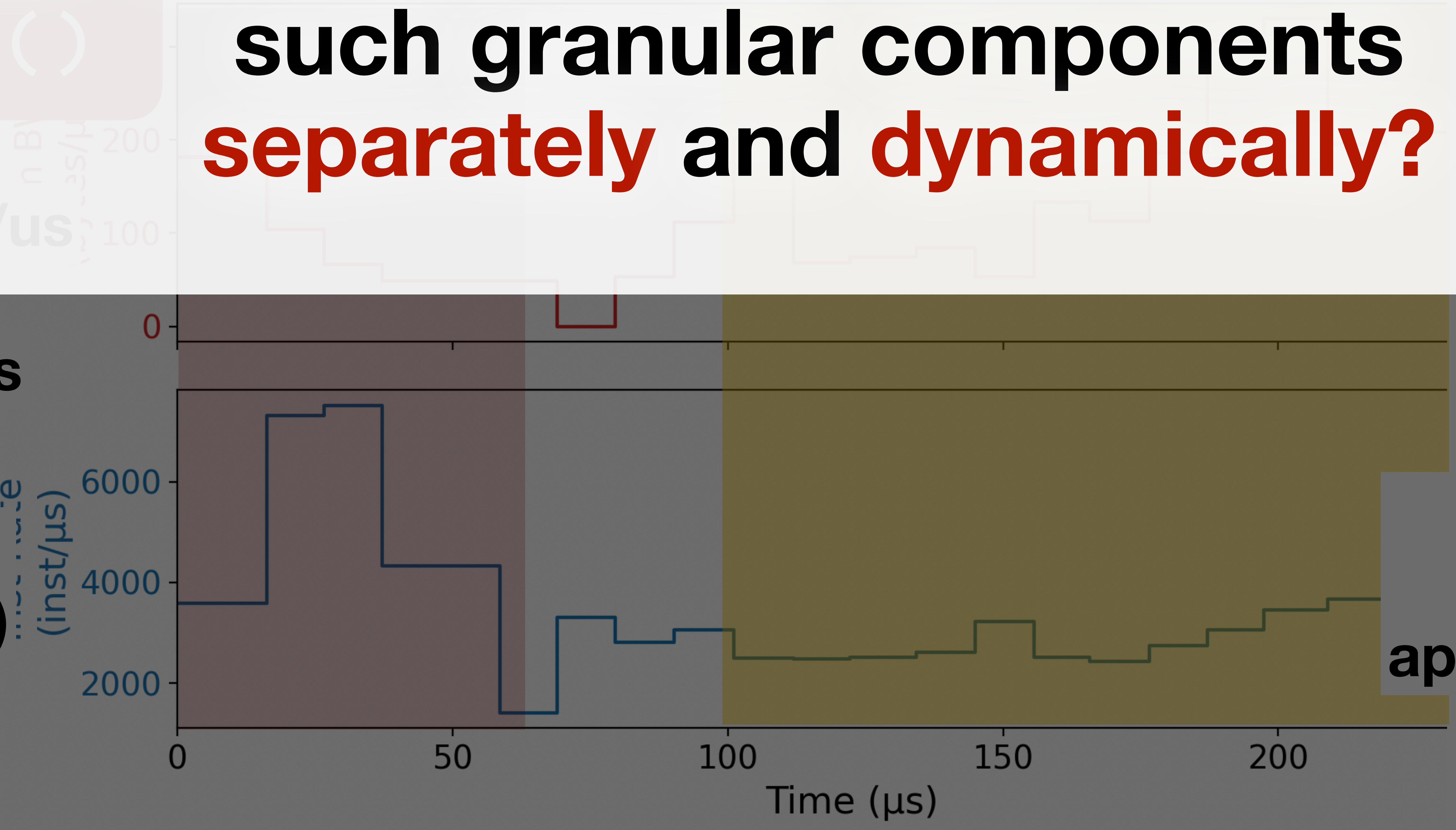
Can we really disaggregate and schedule such granular components separately and dynamically?

DRAM_BW: 117 Byte/us

DRAM_BW: 218 Byte/us

INST_Rate: 4395 /us

INST_Rate: 2909 /us



Want better CPU (frequency & pipeline)

More mem channels, Colocate with other apps consuming less DRAM BW

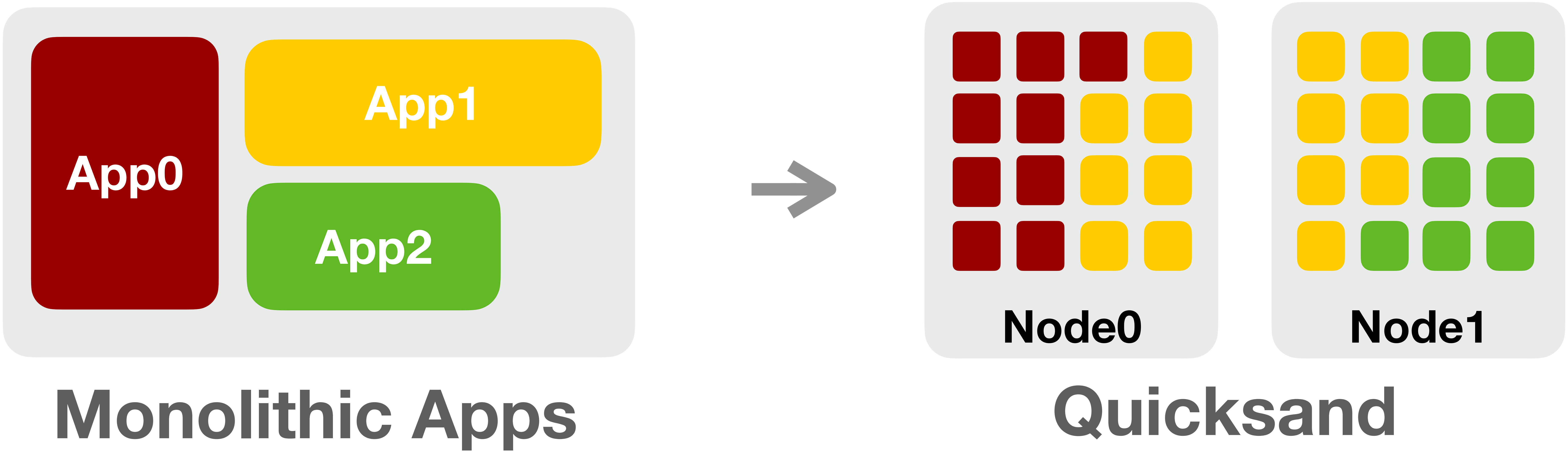
Emerging Granular Computing

Emerging systems with a serverless yet stateful nature, e.g. Quicksand[NSDI25], and Granny[NSDI25], unlock more flexibility to schedule small components

Such systems support fast live migrations and managed local/remote data access

Emerging Granular Computing

Emerging systems with a serverless yet stateful nature, e.g. Quicksand[NSDI25], and Granny[NSDI25], unlock more flexibility to schedule small components



Granular Resource Demand Heterogeneity

How to measure the demands

How to exploit the heterogeneity

Granular Resource Demand Heterogeneity

How to measure the demands

Hiresperf

How to exploit the heterogeneity

Future directions

How to Measure Demands within an App

Breaking an app apart and microbenchmark resource consumptions of individual components?

Engineering efforts 😢

Lock & sync behaviors 😢

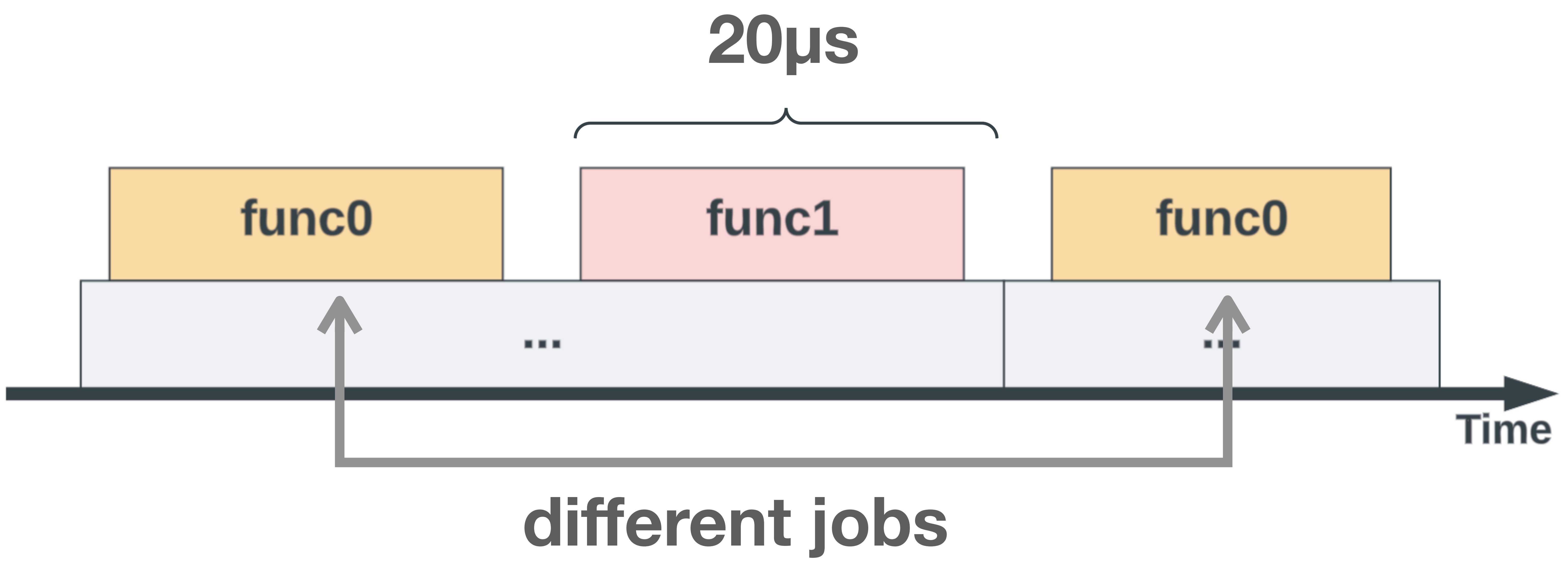
So we want runtime profiling without decomposing the application, with the help of performance counters (PMCs)

Requirements for the Profiler

Resolution — Polling PMCs with microsecond-level intervals

Timeline — Distinguish invocations of the same function

Overhead — Not perturb program behavior; JIT profiling



Requirements for the Profiler

Linux perf → No timeline support 😢

VTune → Proprietary & Interrupts? → high overheads 😢

Promblems with Existing Profilers

Linux perf → No timeline support 😢

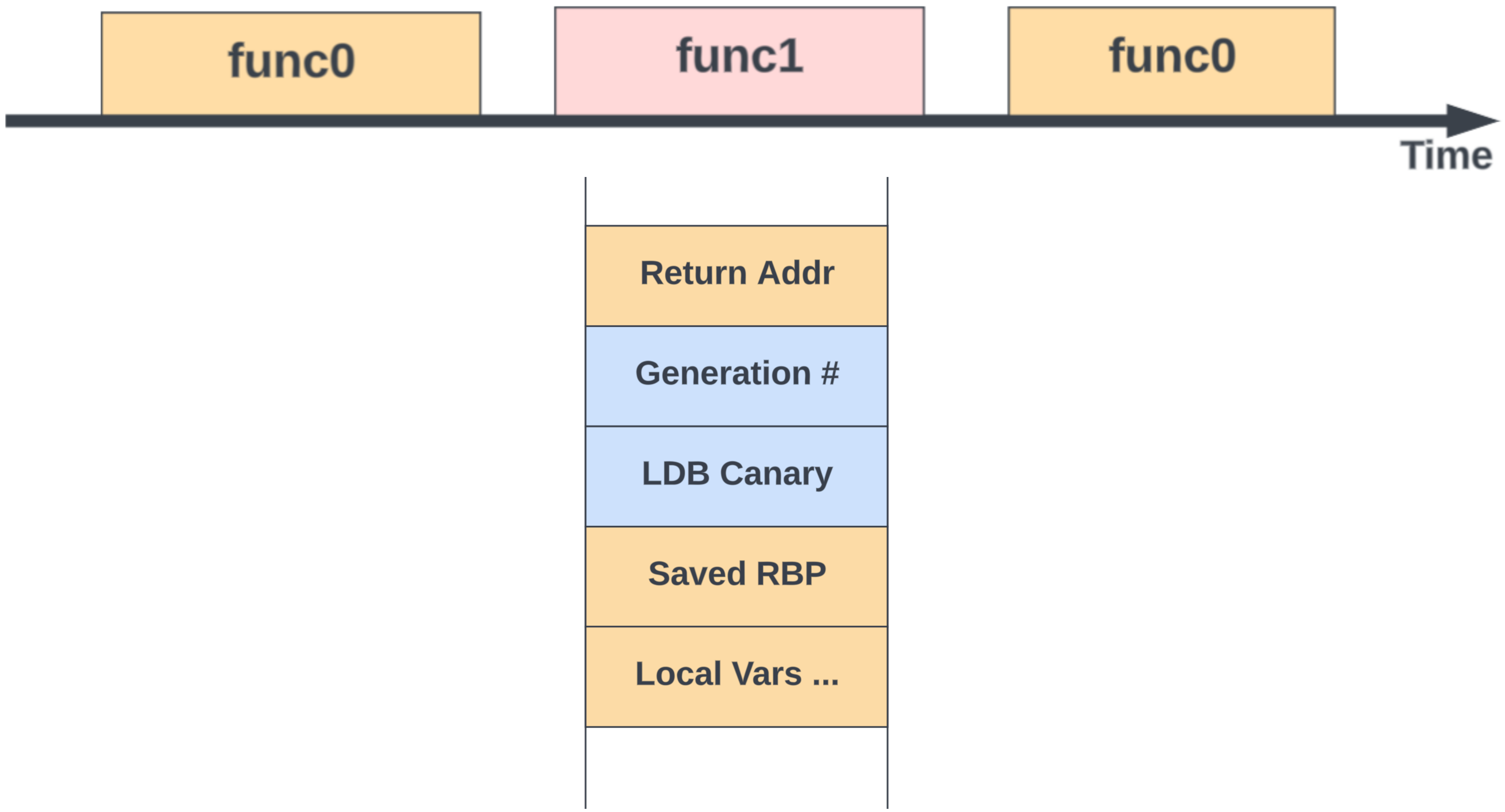
VTune → Proprietary &
Interrupts? → high overheads 😢

Hiresperf → us-level resolution ✅
Timeline support ✅
Low overhead ✅

Hiresperf Design

Func-call Timeline

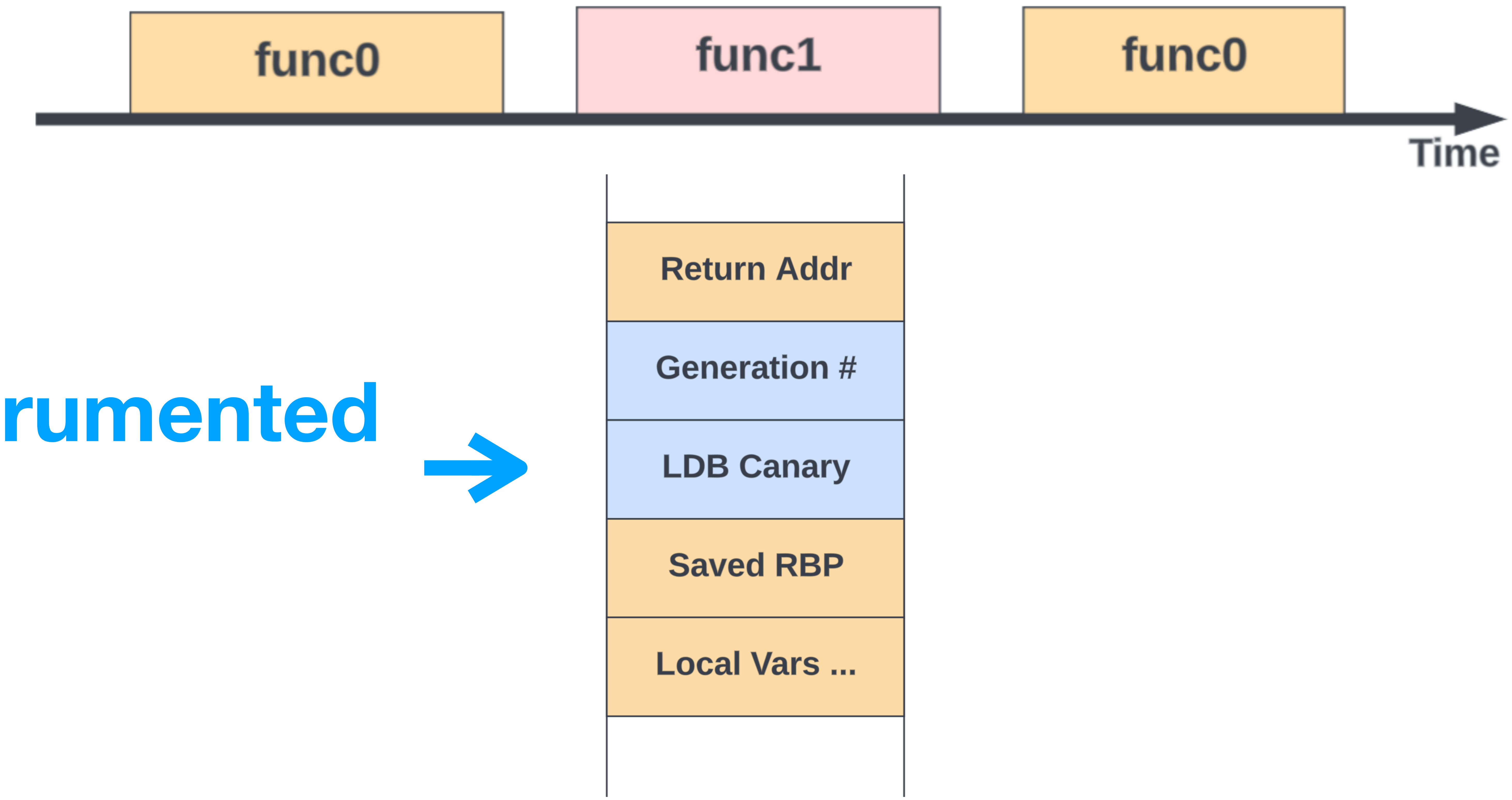
Interrupt-free stack scanning (from LDB)



Hiresperf Design

Func-call Timeline

Interrupt-free stack scanning (from LDB)

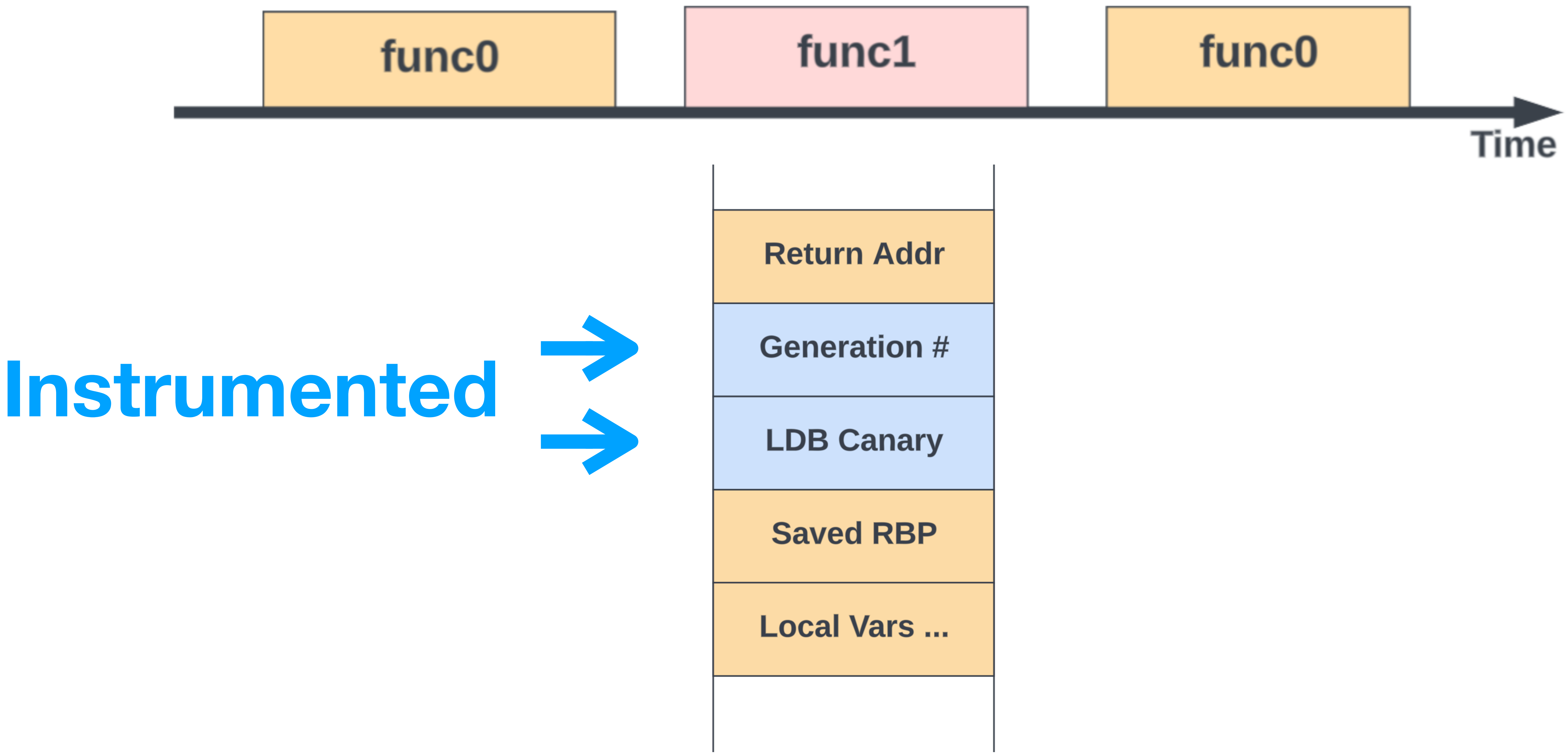


Instrumented

Hiresperf Design

Func-call Timeline

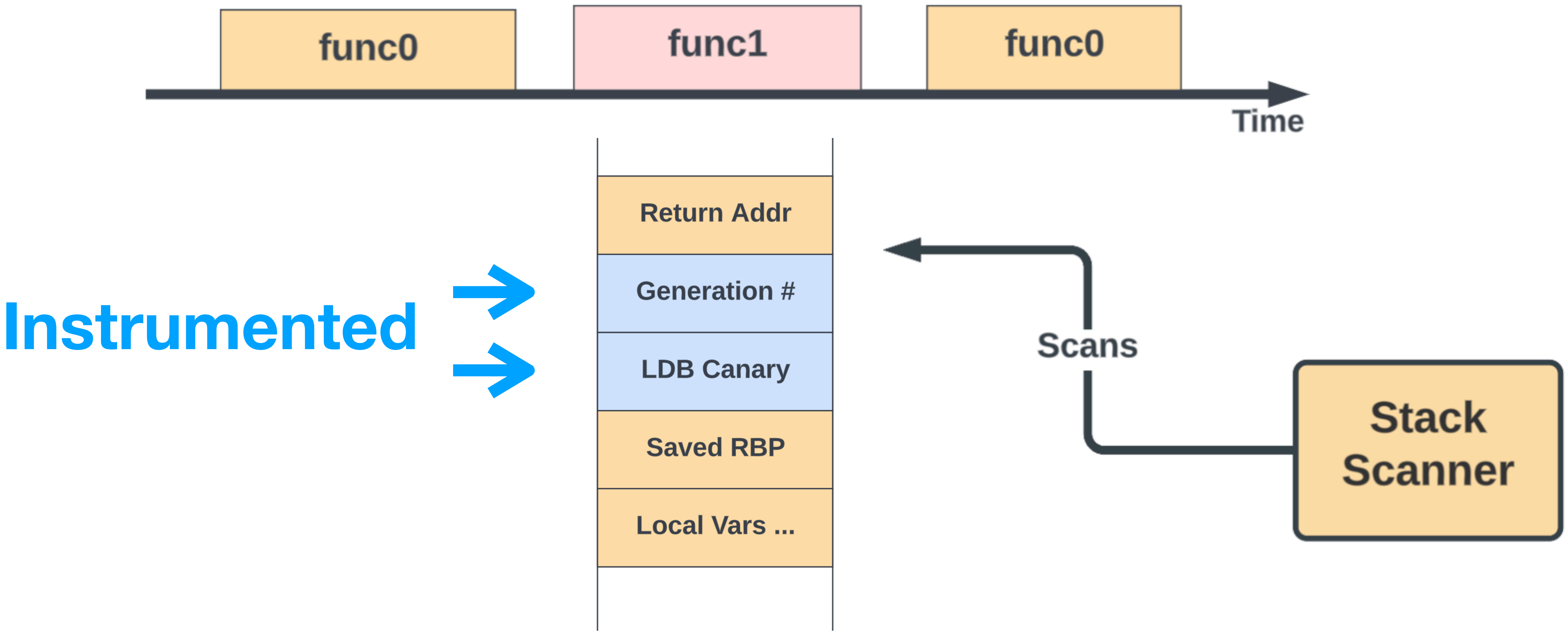
Interrupt-free stack scanning (from LDB)



Hiresperf Design

Func-call Timeline

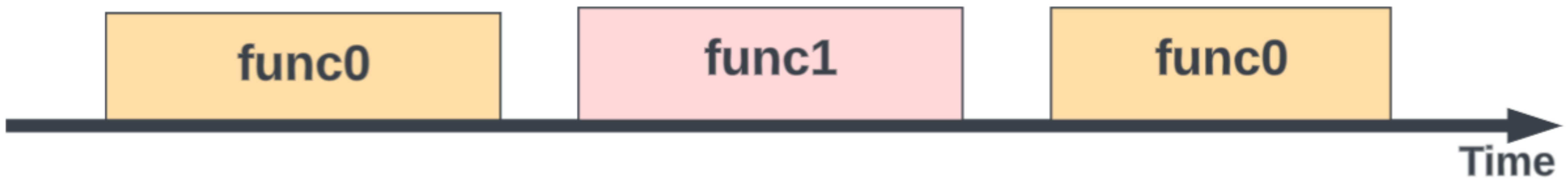
Interrupt-free stack scanning (from LDB)



Hiresperf Design

Func-call Timeline

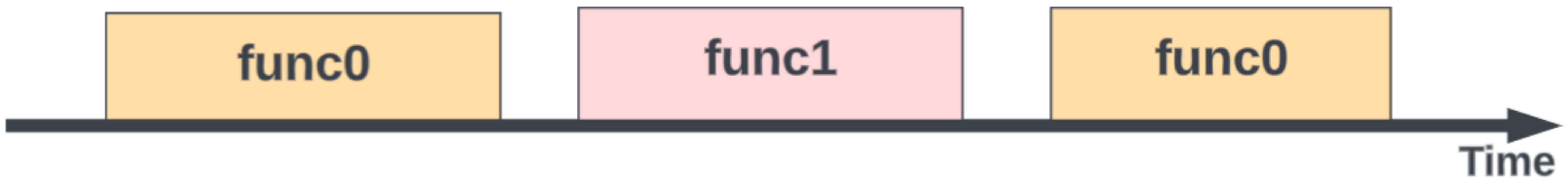
Interrupt-free stack scanning (from LDB)



Hiresperf Design

Func-call Timeline

Interrupt-free stack scanning (from LDB)



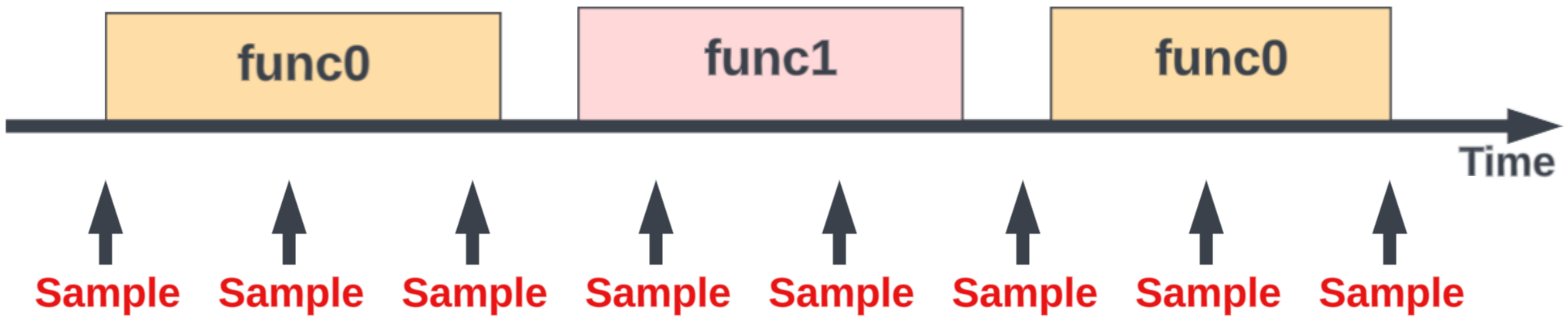
Resource Timeline

Kernel module sending IPIs to poll PMCs

Hiresperf Design

Func-call Timeline

Interrupt-free stack scanning (from LDB)



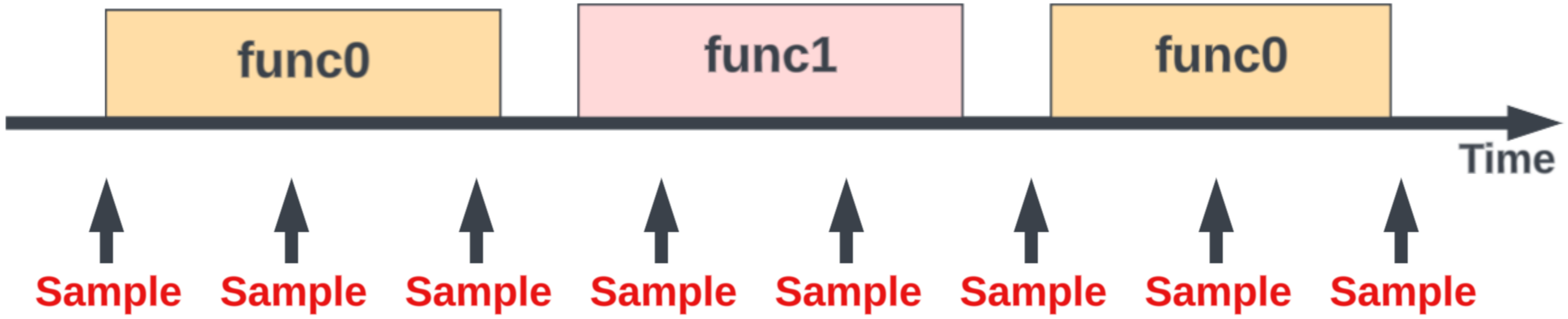
Resource Timeline

Kernel module sending IPIs to poll PMCs

Hiresperf Overhead

Func-call Timeline

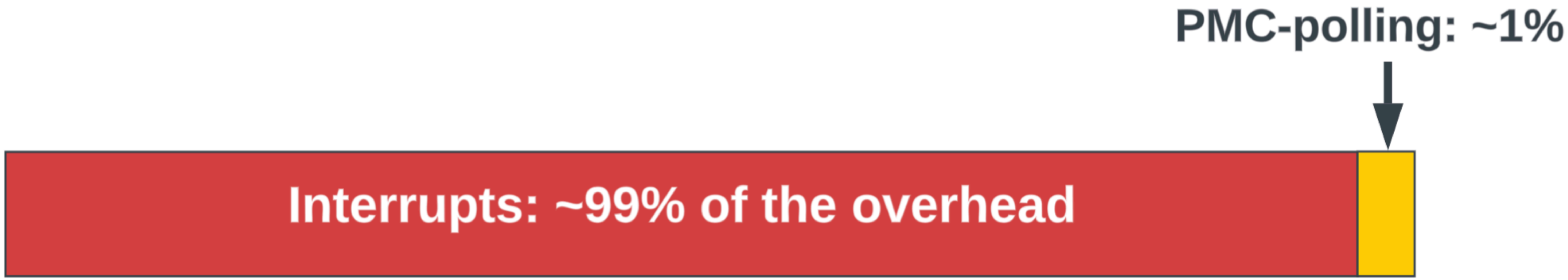
Resource Timeline



VTune at 100us
19 ~ 38% overhead

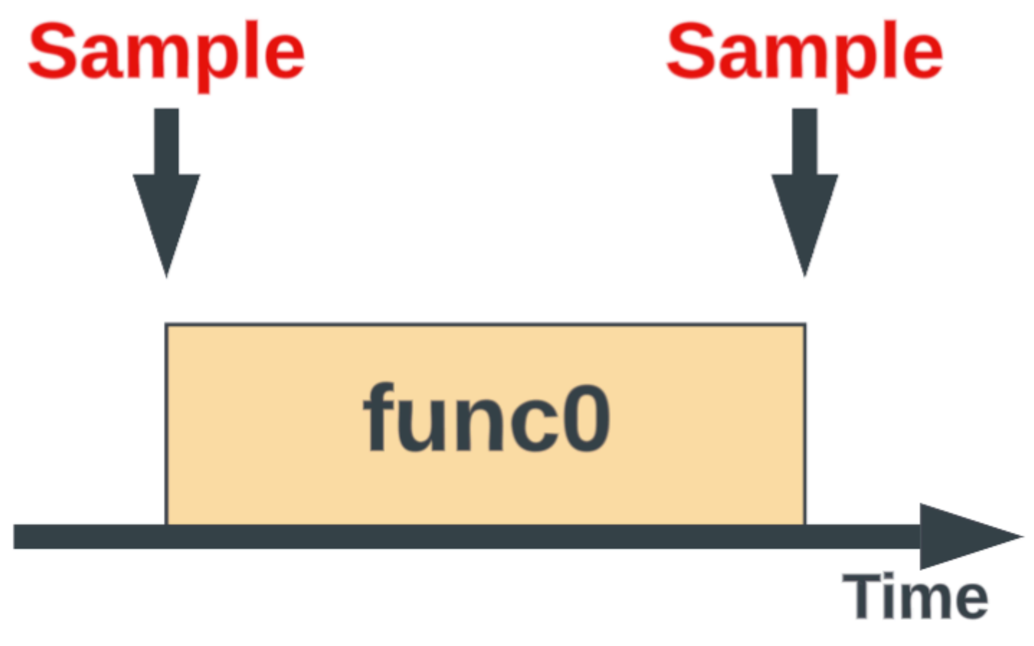
Hiresperf at 10us
0 ~ 8% from stack scans
7% from PMC polling

Even Lower Overhead



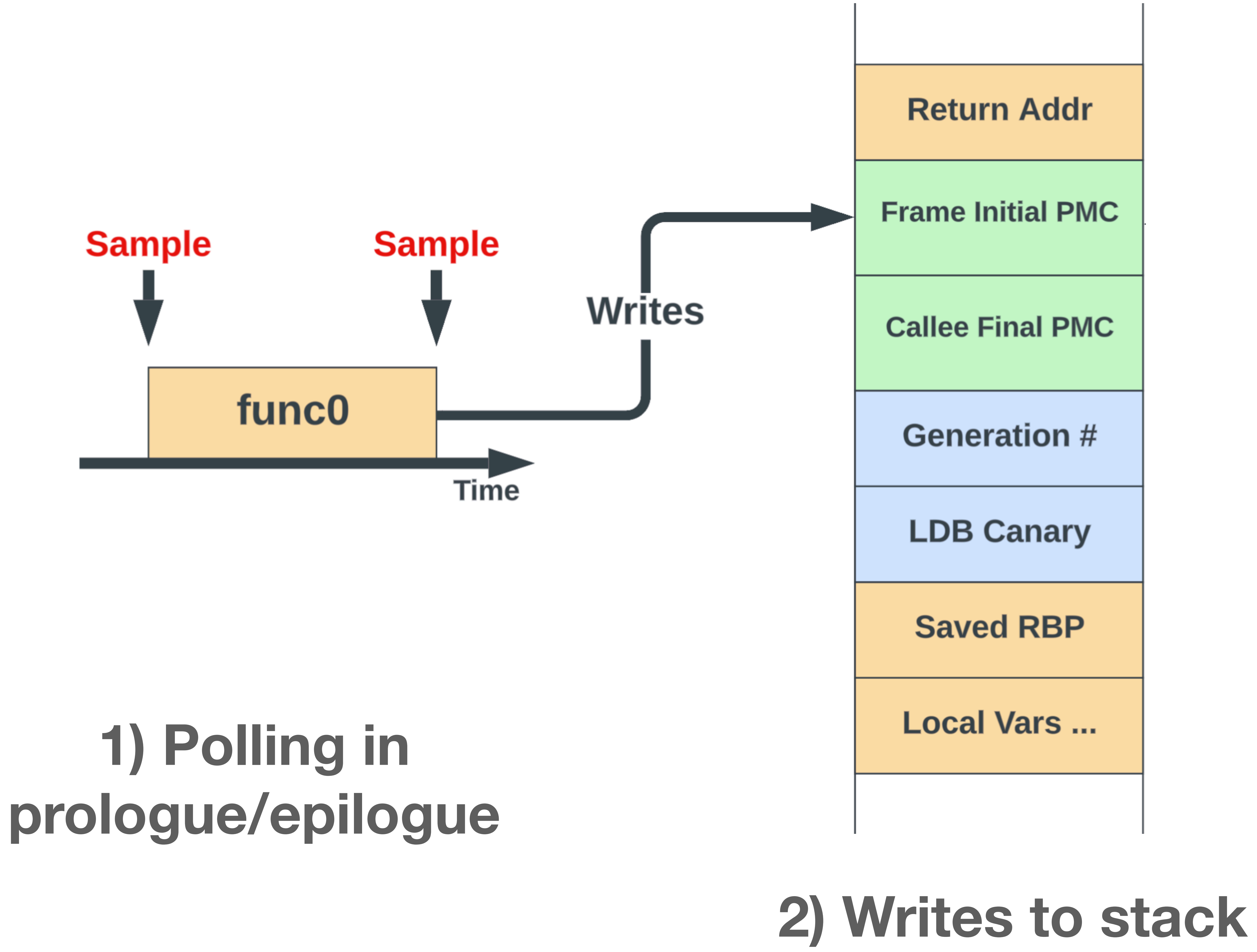
Can we get rid of interrupts? Yes!

Interupt-free Hiresperf

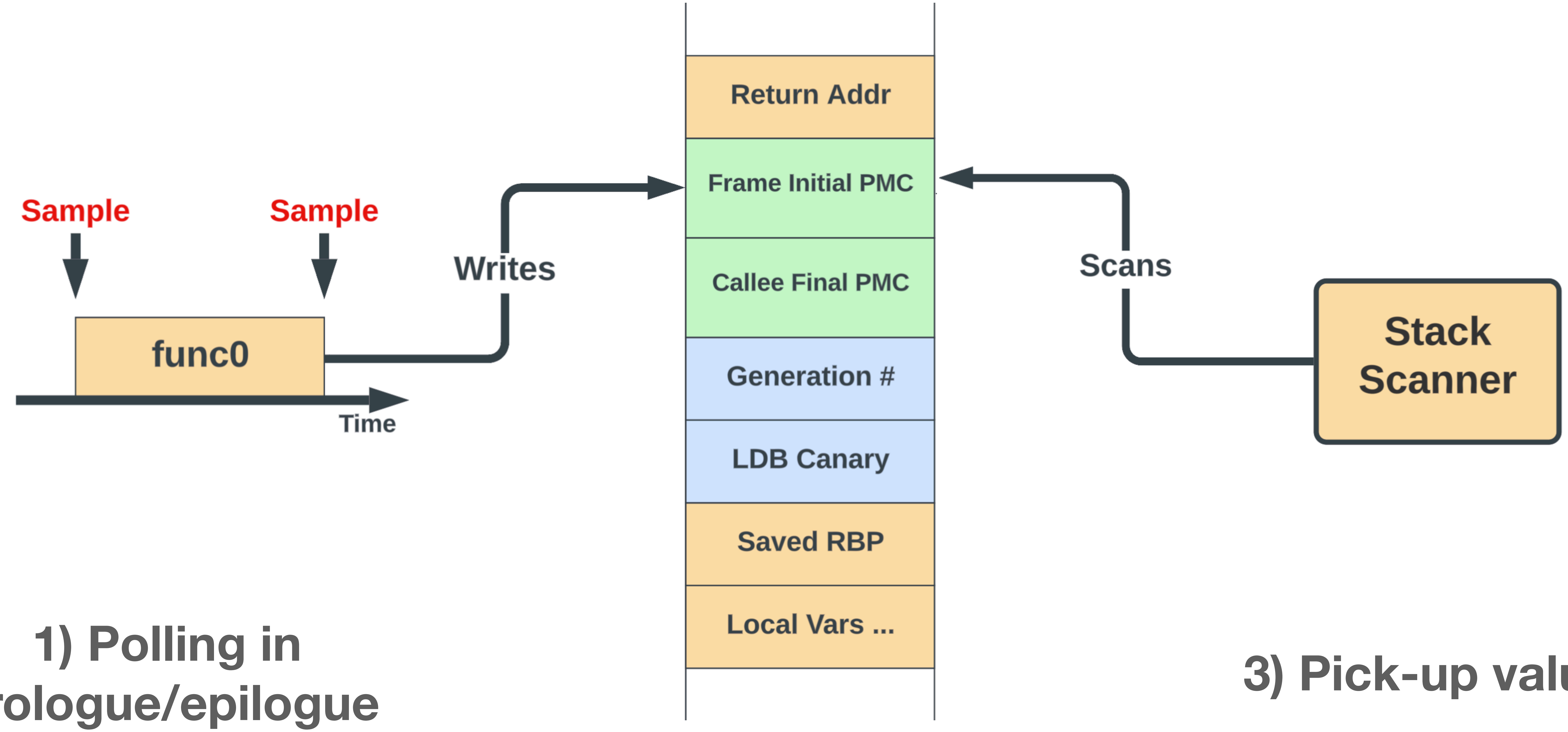


1) Polling in prologue/epilogue

Interupt-free Hiresperf



Interupt-free Hiresperf



1) Polling in prologue/epilogue

2) Writes to stack

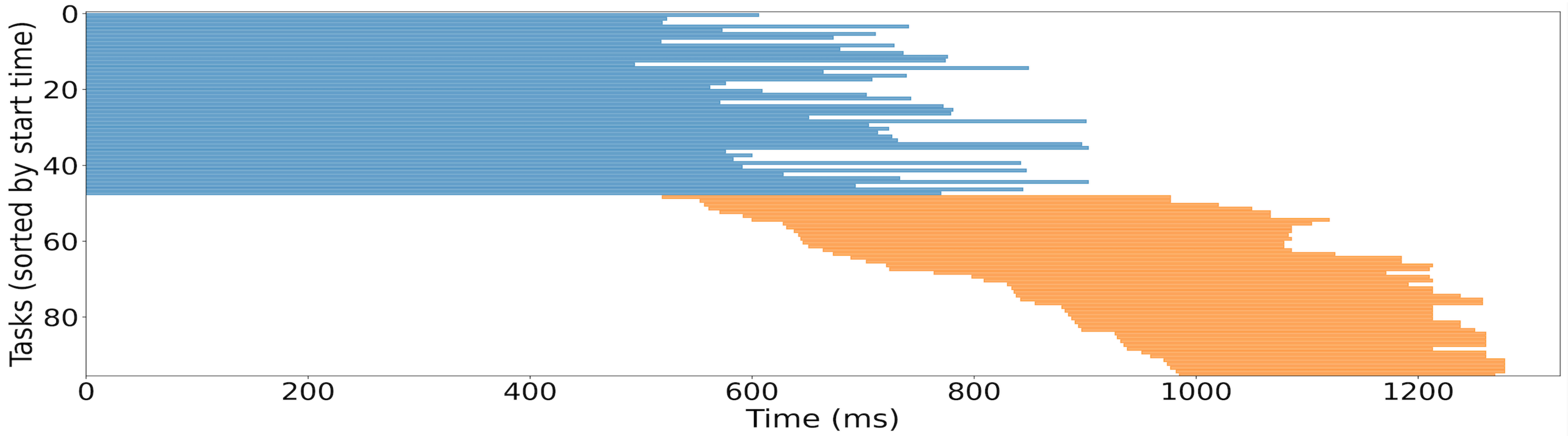
3) Pick-up values

Future Directions

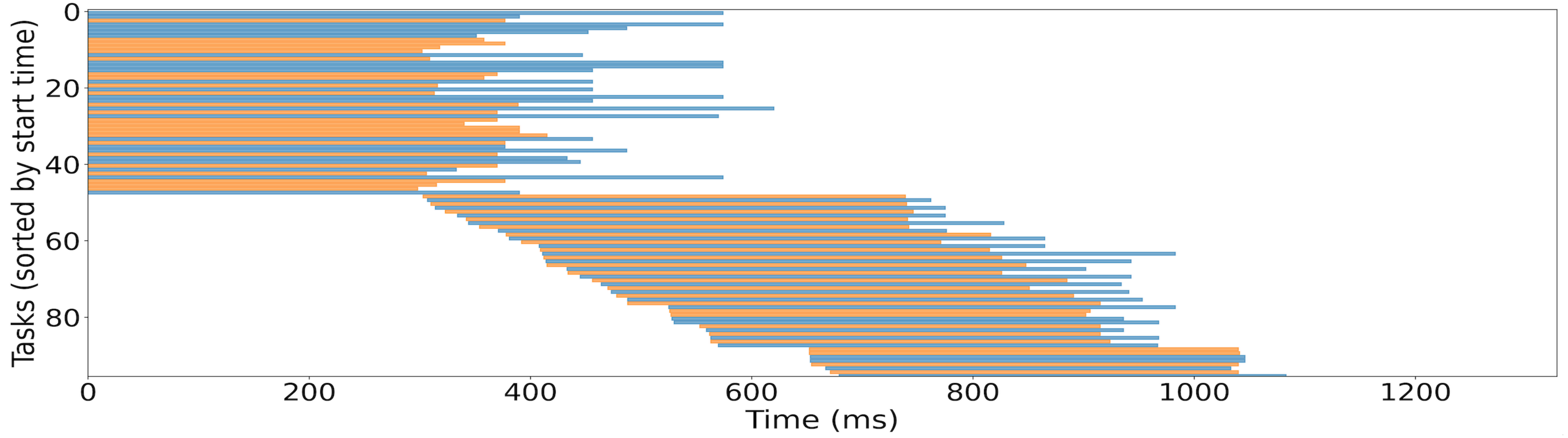
- Batched Processing
- QoS with High Utilization
- Function as a service

Future Direction 1: Batched Processing

Less latency constraints → More scheduling opportunities.
e.g. run a mem-bw intensive operation with a compute-bound one



Non Demand-Aware Scheduling



Interleave Heterogeneous Demands

Future Direction 1: Batched Processing

Less latency constraints → More scheduling opportunities.
e.g. run a mem-bw intensive operation with a compute-bound one

If a single task exhibit different resource needs across different stages, we can also migrate them to always use the best fitting instance.

Prior works developed for DNN-training and Spark. We can generalize this approach to generic programs with profiling.

Future Direction 2: QoS with High Utilizations

- **Current reactive approach** (e.g. Caladan[OSDI20])
  - ban cores for best-effort tasks when increased queuing delay observed for latency critical tasks

- **We want: Proactive Approach**
  - schedule best-effort tasks that does not interfere the latency critical ones
  - utilize resources not consumed by latency critical apps

Future Direction 3: Function-as-a-Service

FaaS containers are natrually suitable for migrations.

- When a function is deployed the first time, low-overhead profiling can learn its resource properties, and adjust later placements.
- When a instance's resource usages shift over time, containers can also migrate.

Summary

Targeting on applications with inherent heterogeneous resource needs, we can leverage on
1) profiled knowledge of fine-grained resource demands
2) granular computing platforms' scheduling flexibility
to improve both performance and utilizations.

Thank you!

Hiresperf is available at
github.com/yizhuoliang/hiresperf

*Interrupt-free version will be released soon