

Argus

Agentic, Reference-Calibrated, Tree-Guided, System-Software-Level Bottleneck Localization

Vlad-Petru Nitu

Harsh Songara

Konstantinos Sgouras

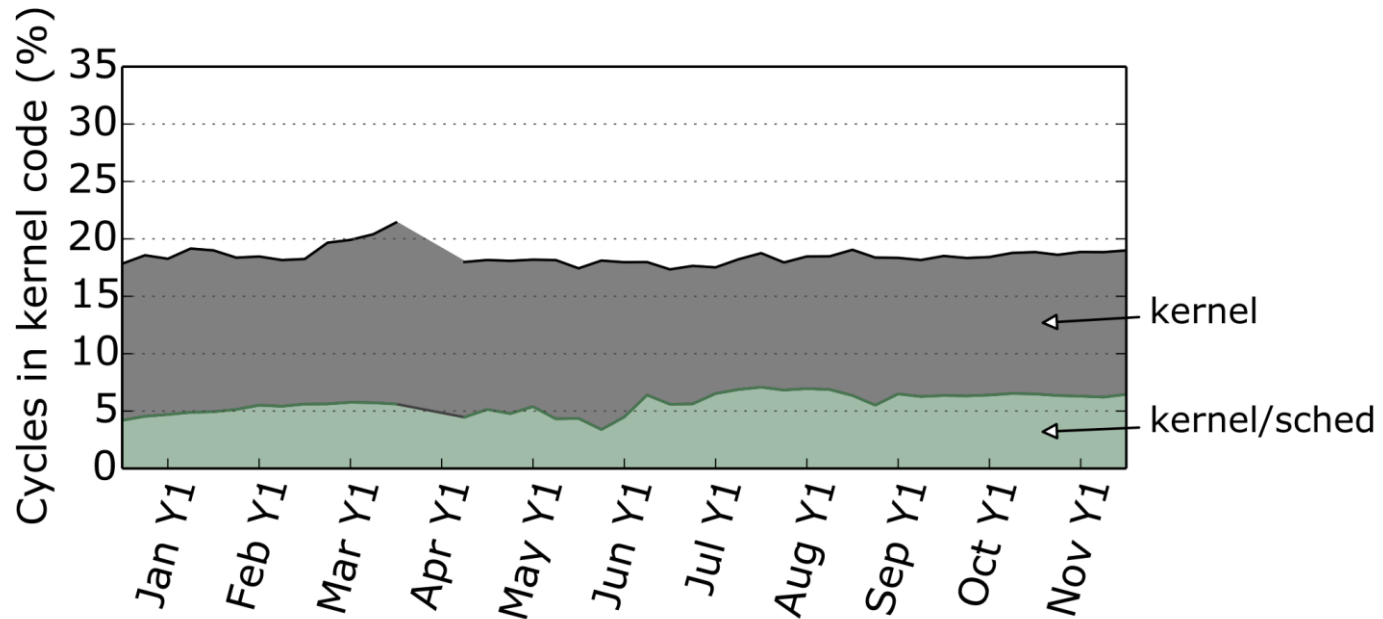
Spiros Galanopoulos

Konstantinos Kanellopoulos

Onur Mutlu

Operating System Routines Consume Many CPU Cycles

Kanev et al., "Profiling a Warehouse-scale Computer", ISCA 2015



**On average ~20% of cycles spent
on executing kernel code**

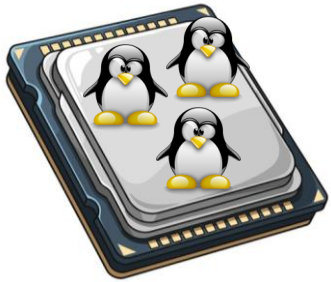
Operating System Overheads Keep Increasing

1

An increasing number of applications is offloaded to specialized compute units

Operating System Overheads Keep Increasing

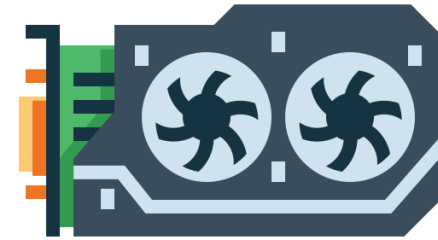
OS running
on CPUS



Offload code

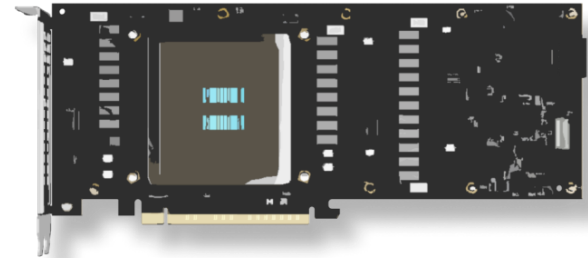


GPUs



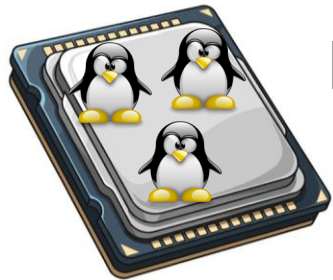
AI Accelerators

Configure &
Offload computation

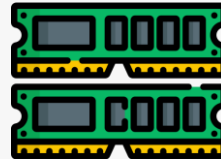
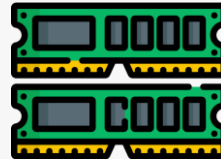
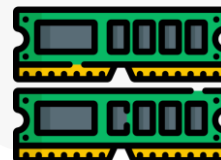


Operating System Overheads Keep Increasing

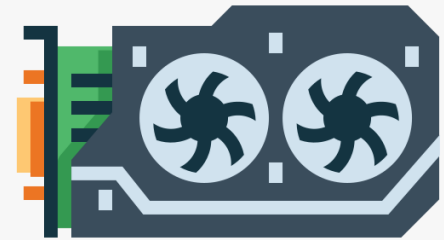
OS running
on CPUS



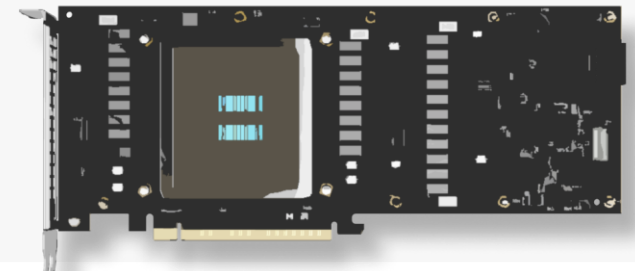
Manage Memory



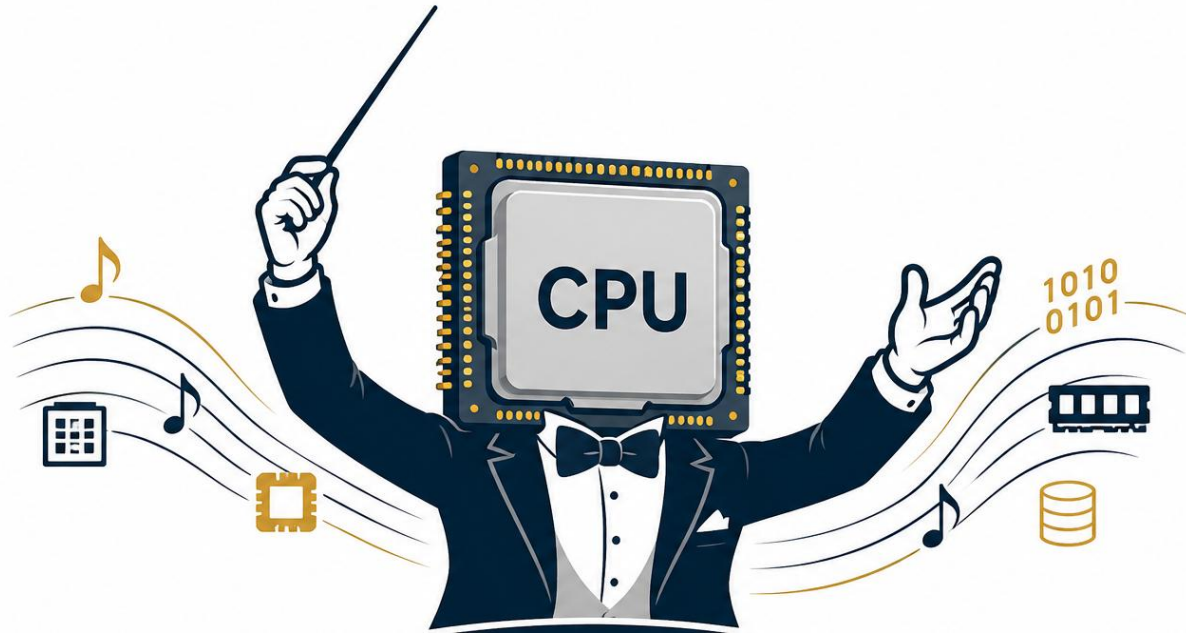
GPUs



AI Accelerators



Operating System Overheads Keep Increasing

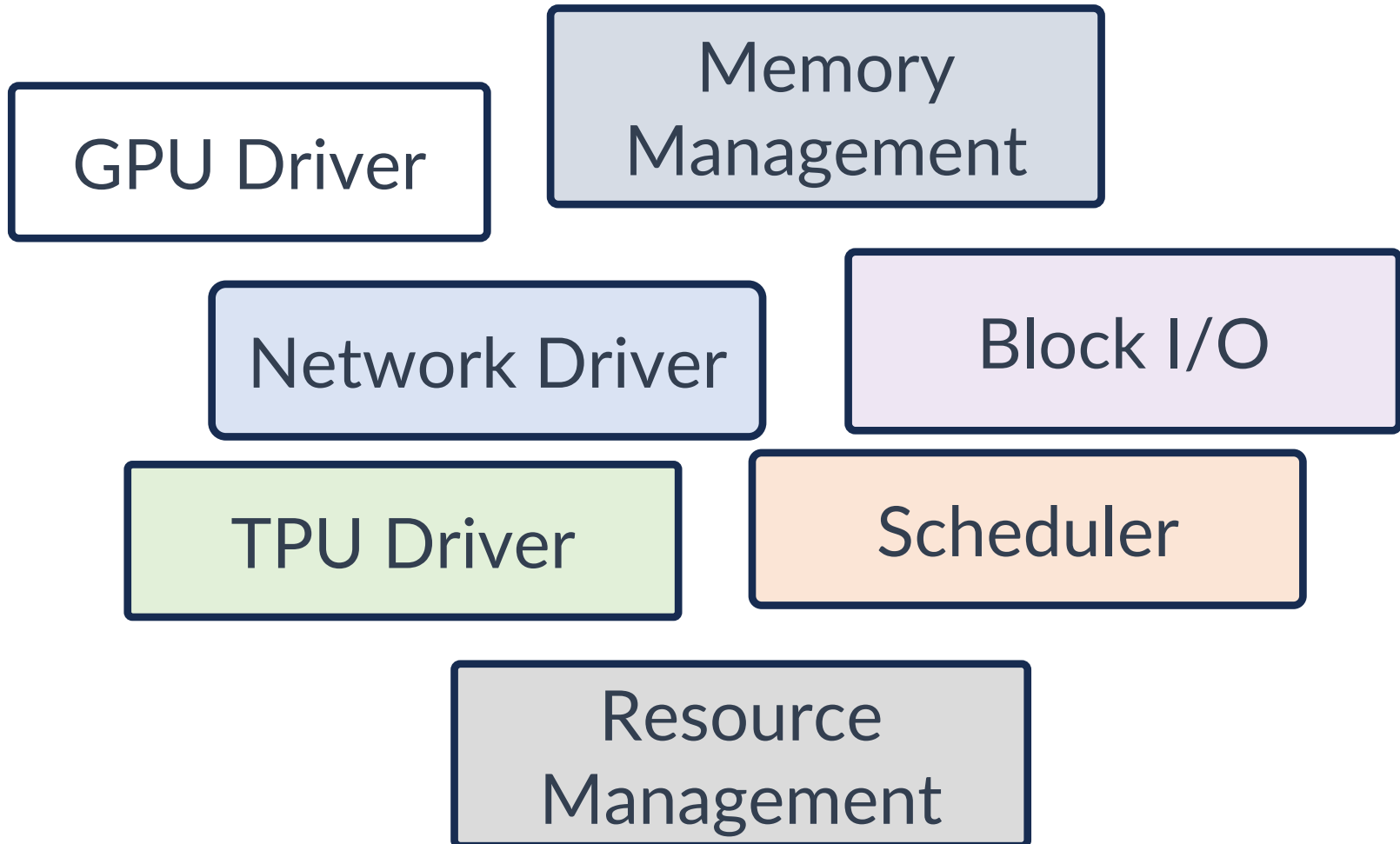


CPU spends an increasing fraction of time
executing OS routines, acting like an orchestrator

Operating System Overheads Keep Increasing

- ① An increasing number of applications is offloaded to specialized compute units
- ② System software is becoming more complex and relies on general-purpose compute units

Operating System Becomes More and More Complex

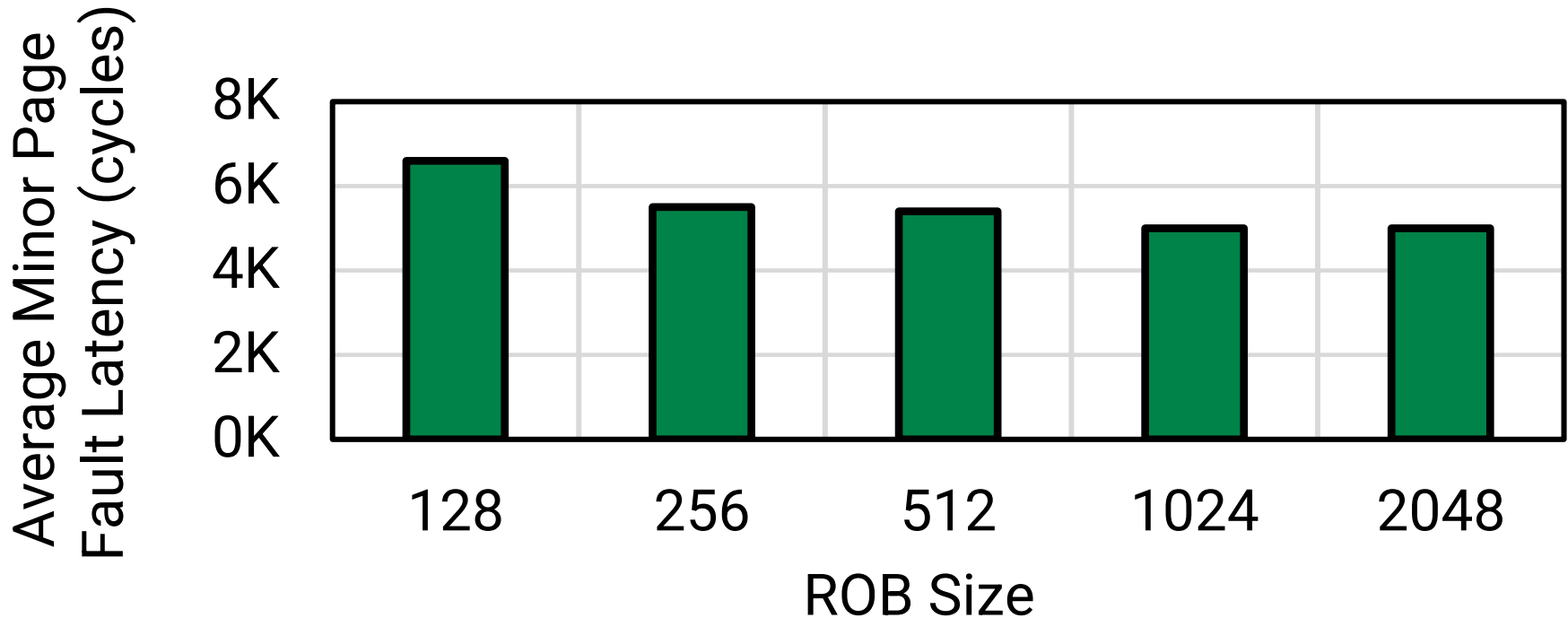


Operating System Becomes More and More Complex

Existing routines require the flexibility that a general-purpose core provides

at a cost..

Example: Scaling CPU Resources



Beyond 1024 ROB entries,
minor page fault latency does not improve

Performance overheads caused by the OS are increasing

We need intelligent profilers to detect and
eventually alleviate the overheads



We Want an OS-level Profiler that is ...

Low overhead: to not distort the measurements

Full kernel visibility: instrument arbitrary kernel code paths

Does not require manual instrumentation

Does not require manual reasoning about profiling data

Existing Profilers

Profiler Category	Low Overhead	Full Kernel Visibility	No Manual Instrumentation	No Manual Reasoning	Expertise
Time-based samplers	✓	✗	✓	✗	★☆☆☆
Hardware-event samplers	✓	✗	✓	✗	★★★★☆
Tracepoint-based profilers	✗	✗	✓	✗	★★★☆☆
Continuous sampling profilers	✓	✗	✓	✗	★★☆☆☆
Profilers for accelerators (e.g. GPUs)	✗	✗	✓	✗	★★★★☆
Dynamic instrumentation profilers	✓	✓	✗	✗	★★★★★
eBPF-based profilers	✓	✓	✗	✗	★★★★★

Existing Profilers

Profiler Category	Low Overhead	Full Kernel Visibility	No Manual Instrumentation	No Manual Reasoning	Expertise
eBPF-based profilers	✓	✓	✗	✗	★★★★

What is eBPF?

eBPF (Extended Berkeley Packet Filter)

Linux mechanism for running
sandboxed programs inside the kernel



eBPF Programs

- Collect & aggregate hardware-sampled metrics
- Export the aggregated result to user-space

eBPF Verifier

- Checks eBPF programs at load-time to ensure safety (e.g., to avoid infinite loops)

Existing Profilers

Profiler Category	Low Overhead	Full Kernel Visibility	No Manual Instrumentation	No Manual Reasoning	Expertise
eBPF-based profilers	✓	✓	✗	✗	★★★★

eBPF runs **arbitrary user-defined code** safely in **kernel space**: compute, filter, and aggregate at the **hook point**

eBPF-based profilers are **more flexible** than dynamic instrumentation, since they support **bounded loops** and **in-kernel logic**

**Still require
manual code writing and
manual reasoning over the profiling data**

LLM-based Coding Agents create an opportunity



Claude Code



Gemini



OpenAI Codex

Enable quick and
accurate code
production

Enable autonomous
reasoning

Opportunity

LLMs can now...	... which unlocks	... closing the following gap
Author eBPF code	Automated eBPF code generation	Authoring Gap: Expertise
Interpret results (e.g., which subsystem causes the bottleneck)	Automated reasoning	Interpretation Gap: Human-in-the-loop

Using Agents Out-of-the-Box is Challenging

1

Cannot localize bottlenecks without a **reference measurement**

2

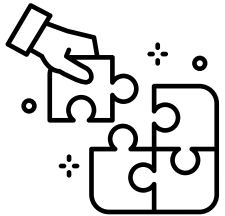
Lack kernel knowledge:
oscillates between hypotheses and hallucinating function names

Using Agents Out-of-the-Box is Challenging

Out-of-the-Box Agent fails to localize bottlenecks 45% of the time

Our Goal

Design an **LLM-based agentic profiler** that



Autonomously authors profiling code



Autonomously reasons about bottlenecks

Profile fast and minimize hallucinations

Argus

An **LLM-based** agentic profiler that autonomously discovers OS-level bottlenecks and relies on two components:

1 **Reference calibration** provides a baseline measurement to confirm bottlenecks

2 **Tree-based data structure** creates a structured view of the kernel codebase

Terminology

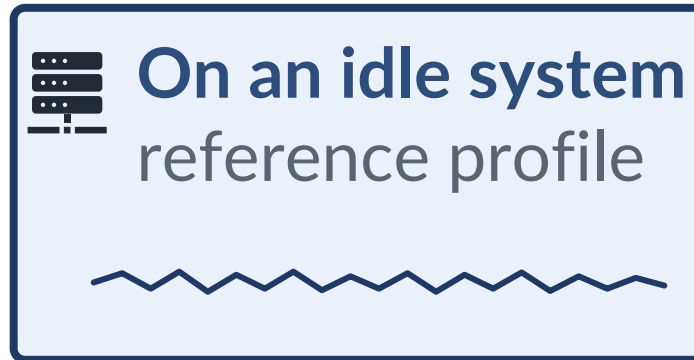
Perturber

Workload or system configuration
that stresses a specific code path

Victim

Workload that we profile under perturbation

Argus – Reference Calibration

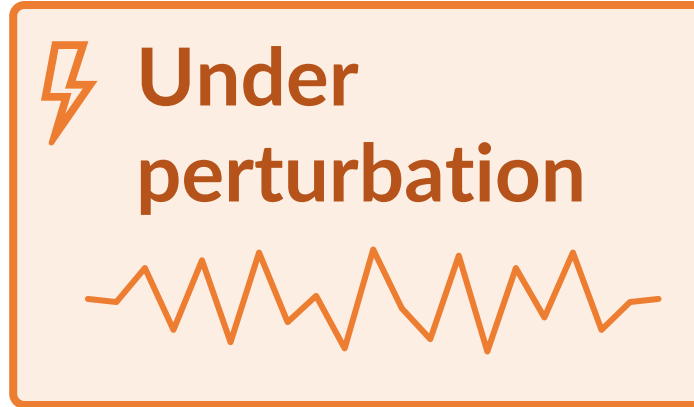


Reference **M**easurement **V**ector (**RMV**)

	Minor Page Fault	CoW	...
# Cycles	500 cycles	...	
LLC Misses	200	...	

...

Argus – Reference Calibration

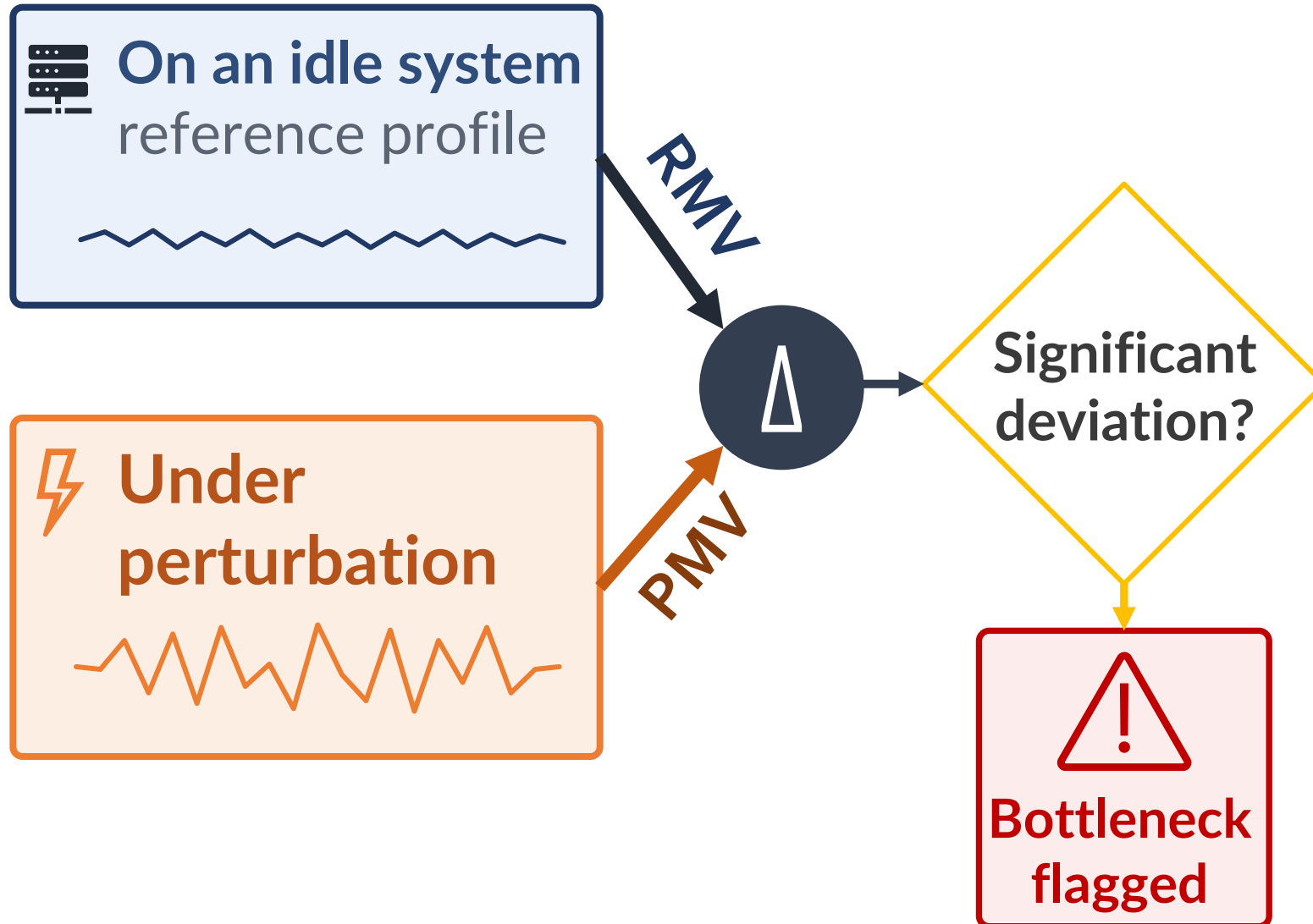


Perturbed **M**easurement **V**ector (**PMV**)

	Minor Page Fault	CoW	...
# Cycles	600 cycles	...	
LLC Misses	200	...	

...

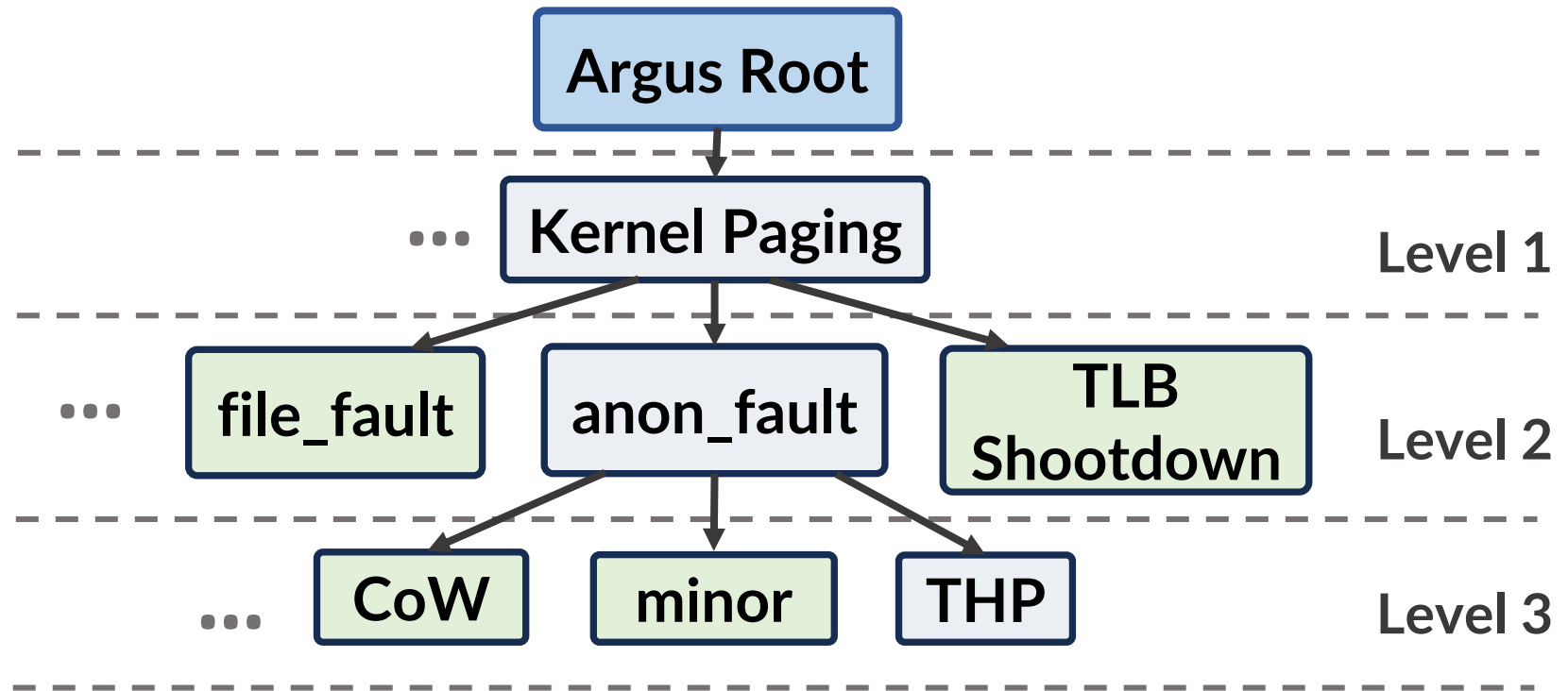
Argus – Reference Calibration



Argus – Reference Calibration

What should the profiler measure?

Argus – Tree of Kernel Code Paths

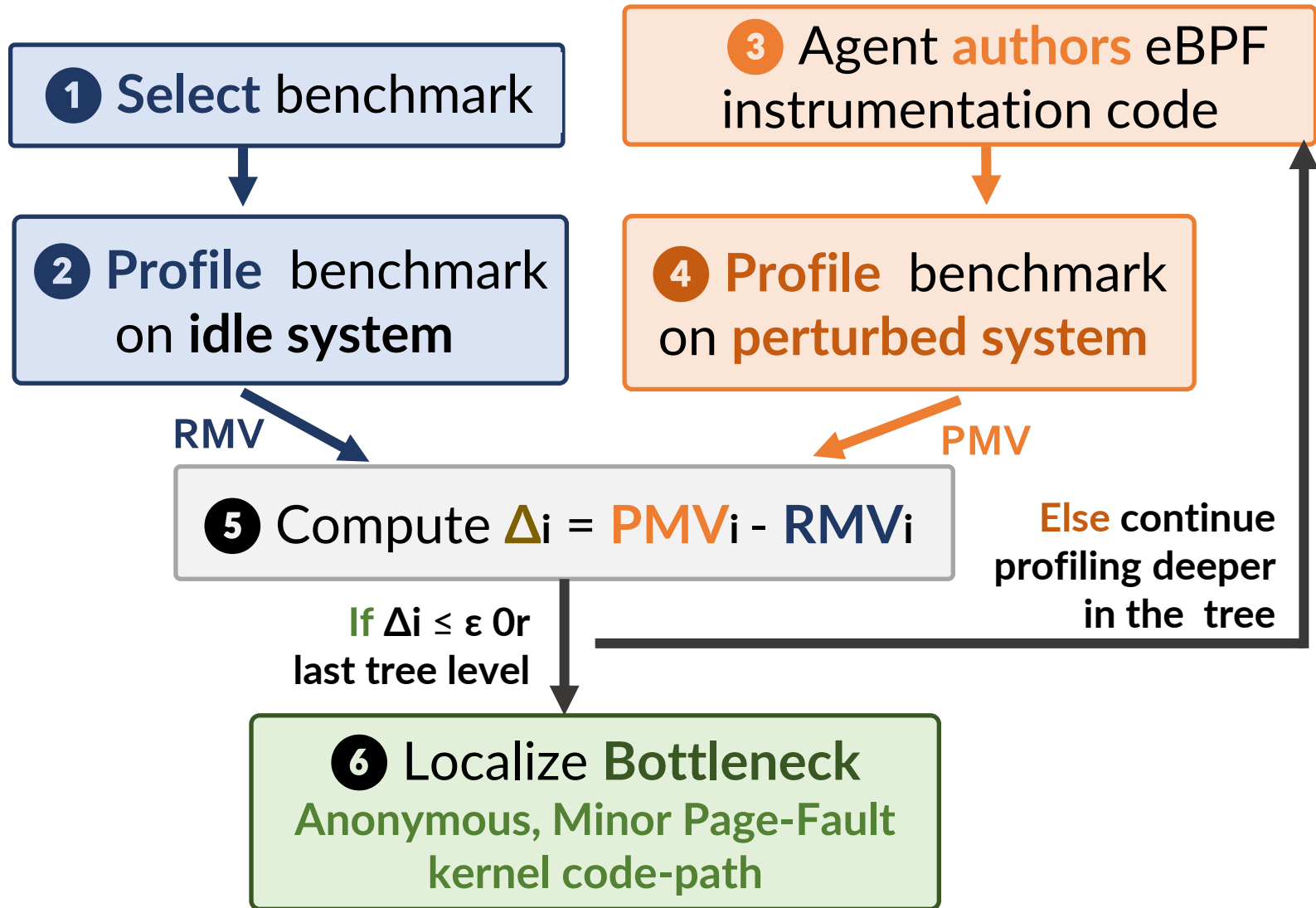


The tree provides context for **bottleneck localization** to constrain exploration to **valid** code paths.

Example - 3rd Tree level

	Minor Page Fault	Copy On Write	THP Collapse
RMV	50k	250	50
PMV	100k	250	50
Δ	$4\sigma > \epsilon$	0	0
i	0	1	2

Argus: End-to-End Flow



Case Study #1: Overheads caused by THP

THP Aggressor



fragments physical memory and then forces compaction by demanding huge pages, **stressing specific code paths** (e.g., THP allocation, compaction, etc.)

What we study



How Argus **localizes the kernel code path** through which a single aggressor slows down diverse victim workloads.

Why we study



- 1) To validate that **Argus produces different diagnoses** for different victims under the same aggressor
- 2) To show **Argus can correctly abstain** when there's no bottleneck

Evaluation Methodology

System configurations:

- **CPU:** Intel i9-14900K (workloads pinned to P-cores)
- **DRAM:** 32 GB DDR5
- **OS:** Linux 6.17, Ubuntu 24.04 LT
- **LLM-based agent:** Gemini 2.5 Flash

Workloads

- GUPS, Stream, llama.cpp, fork_storm
- Each workload run for N repetitions, where the ground truth bottleneck is known

Aggressor

- Microbenchmark that:
 - 1) allocates 1 GB
 - 2) fragments memory in a checkerboard pattern
 - 3) randomly touches 512 MB on the fragmented region

Case Study #1: Overheads caused by THP

① GUPS and Stream do not perform many allocations, so the THP aggressor should not cause overheads

Argus always reports **no bottleneck**

② llama.cpp suffers from a high number of page faults

Argus is always partially correct, flagging **file** and **anonymous page fault** code paths

③ fork_storm stresses the Scheduler subsystem

Argus is always partially correct, finding a fine-grained bottleneck **66% of the time**

Comparison Points

We use **4 comparison points** in our evaluation

1

B-LLM-Prior: naïve agent that decides based on LLM's prior knowledge

2

B-LLM-Prior-Tree: B-LLM-Prior augmented with the Tree-based data structure

3

B-LLM-Prior-Tree-Probes:
B-LLM-Prior-Tree augmented with LLM-authored eBPF instrumentation code

4

Argus: B-LLM-Prior-Tree-Probes augmented with the **reference calibration** component

Evaluation Metrics

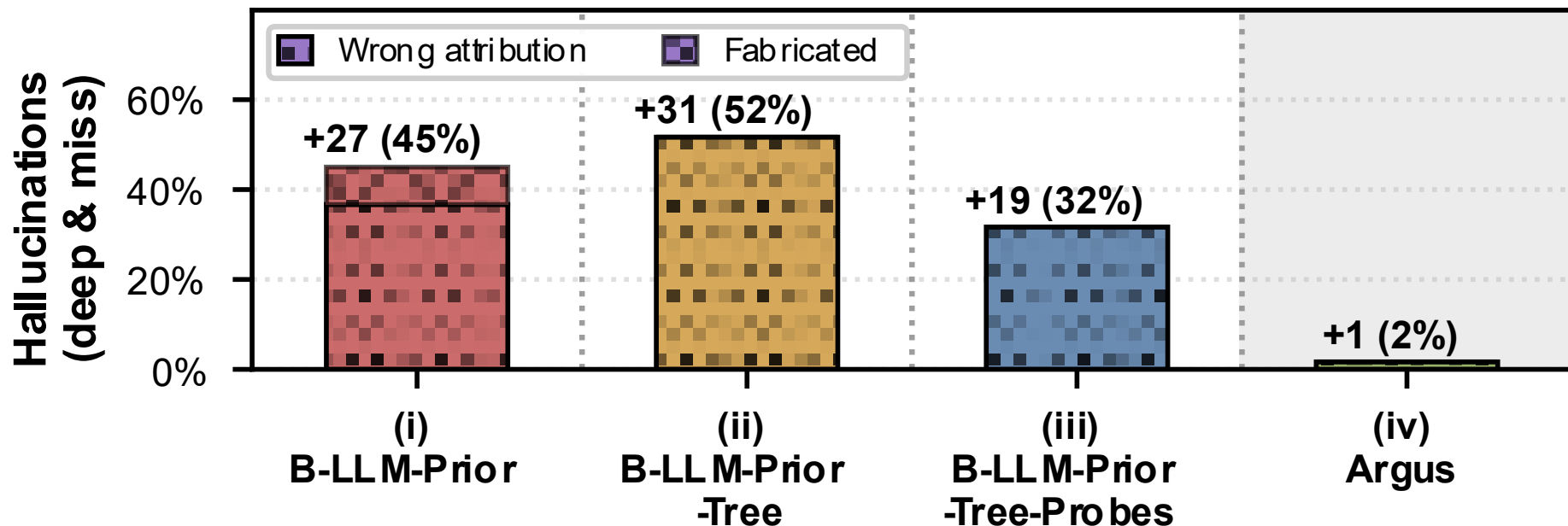
1 Hallucination Rate

2 Time-to-Diagnosis (TtD)

1 Hallucination Rate

Percentage of deep diagnoses
(path depth ≥ 2) that did not match the
ground-truth bottleneck

Measuring Accuracy



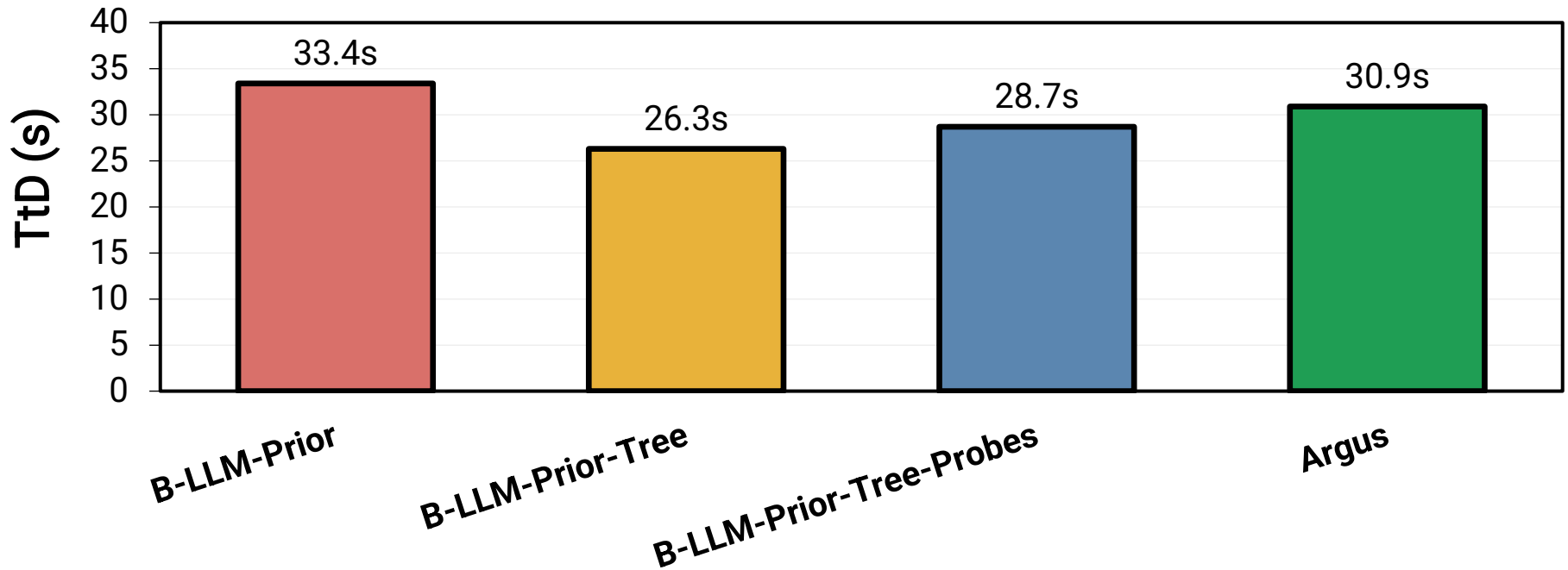
Argus hallucinates **16× less**,
compared to the best-performing baseline

Evaluation Metrics

2 Time-to-Diagnosis (TtD)

Wall-clock time from workload start to bottleneck diagnosis

Time-to-Diagnosis (TtD)



Argus incurs no measurable TtD increase over the LLM-prior baselines

Argus is Open-Source

 **Argus** Public

Watch 0 Fork 0 Star 0

main 1 Branch 0 Tags

Go to file T Add file <> Code

 **Vlad-Petru Nitu** Upload Code Architecture 2.0 workshop @ ISCA 2026 726e5ce · 4 minutes ago 1 Commit

agent_handlers	Upload Code Architecture 2.0 workshop @ ISCA 2026	4 minutes ago
benchmarks	Upload Code Architecture 2.0 workshop @ ISCA 2026	4 minutes ago
docs	Upload Code Architecture 2.0 workshop @ ISCA 2026	4 minutes ago
orchestrator	Upload Code Architecture 2.0 workshop @ ISCA 2026	4 minutes ago
paper	Upload Code Architecture 2.0 workshop @ ISCA 2026	4 minutes ago
perturbations	Upload Code Architecture 2.0 workshop @ ISCA 2026	4 minutes ago
scripts	Upload Code Architecture 2.0 workshop @ ISCA 2026	4 minutes ago
src	Upload Code Architecture 2.0 workshop @ ISCA 2026	4 minutes ago
.gitignore	Upload Code Architecture 2.0 workshop @ ISCA 2026	4 minutes ago
.gitmodules	Upload Code Architecture 2.0 workshop @ ISCA 2026	4 minutes ago
ARTIFACT.md	Upload Code Architecture 2.0 workshop @ ISCA 2026	4 minutes ago
DEMO.md	Upload Code Architecture 2.0 workshop @ ISCA 2026	4 minutes ago

About

No description, website, or topics provided.

- Readme
- Activity
- Custom properties
- 0 stars
- 0 watching
- 0 forks

Report repository

Releases

No releases published

[Create a new release](#)

Packages

No packages published

[Publish your first package](#)

Contributors

<https://github.com/CMU-SAFARI/Argus>

Conclusion

Prior profiling methodologies require **tedious, manual effort** to

- 1) **author** instrumentation code, and
- 2) **interpret** the profiling data to **reason about bottlenecks**

We propose Argus, an autonomous profiling methodology that guides an LLM agent to close the **authoring** and **interpretation gaps** via

- **reference calibration** to confirm real bottlenecks
- **tree-guided traversal** over a tree of kernel code paths

Argus localizes OS bottlenecks accurately at low cost

- **16× higher localization accuracy** than the best LLM baseline, at comparable time-to-diagnosis (~31 s)

We hope Argus inspires future work on autonomous, agentic profiling

Argus

**Agentic, Reference-Calibrated,
Tree-Guided, System-Software-Level
Bottleneck Localization**

Vlad-Petru Nitu

Harsh Songara

Konstantinos Sgouras

Spiros Galanopoulos

Konstantinos Kanellopoulos

Onur Mutlu

Argus

Backup Slides

Vlad-Petru Nitu

Harsh Songara

Konstantinos Sgouras

Spiros Galanopoulos

Konstantinos Kanellopoulos

Onur Mutlu

SAFARI
SAFARI Research Group

ETH zürich

Executive Summary

Problem: Existing OS-level profilers

- 1) report profiling data without reasoning about bottlenecks (e.g., perf),
- 2) identify bottlenecks only at coarse granularity (e.g., VTune),
- 3) distort the measurement of microsecond-scale operations (e.g., ftrace), or
- 4) demand kernel expertise to write instrumentation code (e.g., eBPF-based profilers).

Motivation: Localizing bottlenecks in OS routines is a **tedious, manual process** limited by two gaps:

- 1) **authoring gap:** writing complex instrumentation code requires kernel expertise
- 2) **interpretation gap:** reasoning over profiling data to reason about bottlenecks requires manual effort

Opportunity:

LLM-based agents can autonomously close both gaps

Challenges of LLM-based Agents:

- 1) cannot localize bottlenecks without a reference measurement, and
- 2) lacks prior kernel knowledge, which leads to oscillating between hypotheses and hallucinating function names

Goal: Develop an **autonomous** OS-level profiling methodology which closes the authoring and interpretation gaps

Argus: guides an LLM-based agent with

- 1) **reference calibration**, which provides a baseline measurement to confirm bottlenecks, and
- 2) **tree-based data structure**, which creates a structured view of the kernel code to autonomously reason over potential OS-level bottlenecks

Key Results: Argus yields **16× higher localization accuracy** than the best LLM baseline, at **comparable time-to-diagnosis** (~31 seconds)

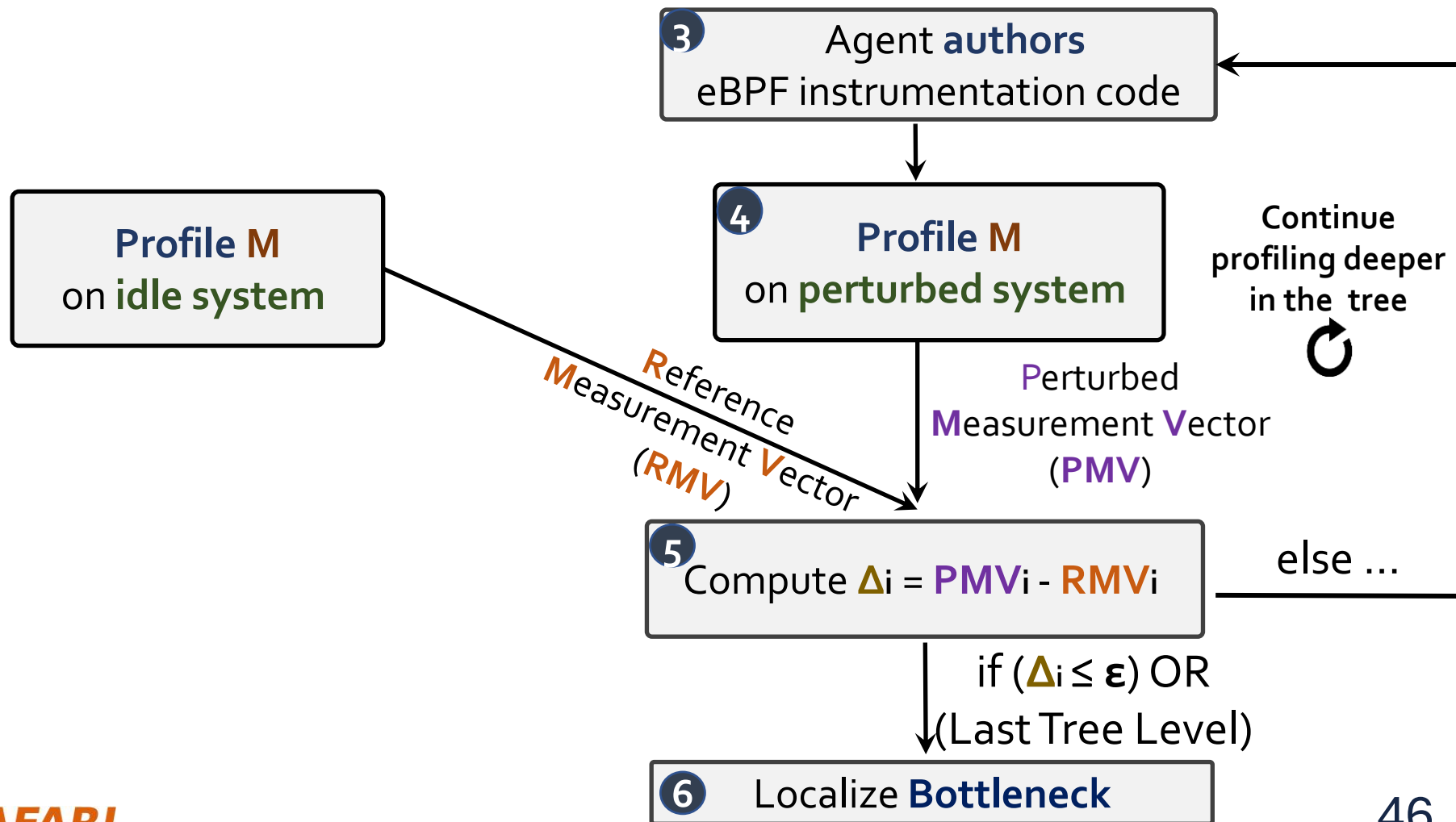
Problem

OS Routines consume up to 30% of datacenter CPU cycles

The problem is getting worse as

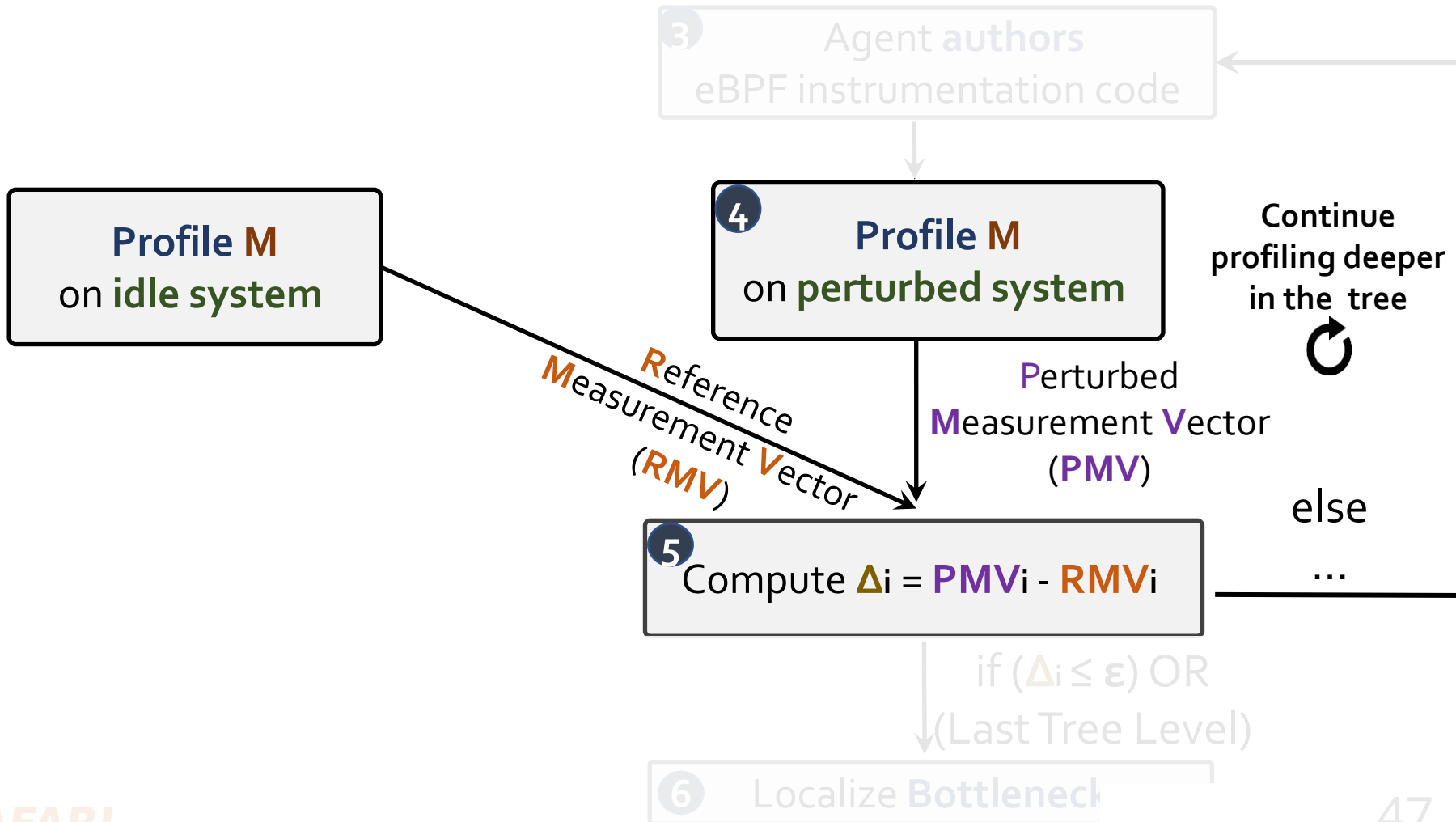
- 1) **The CPU becomes the orchestrator**, as apps increasingly offload code to accelerators
- 2) **CPU-bound workloads** drive frequent kernel events, such as **page faults** and **context switches**

Argus – Reference Calibration (I)



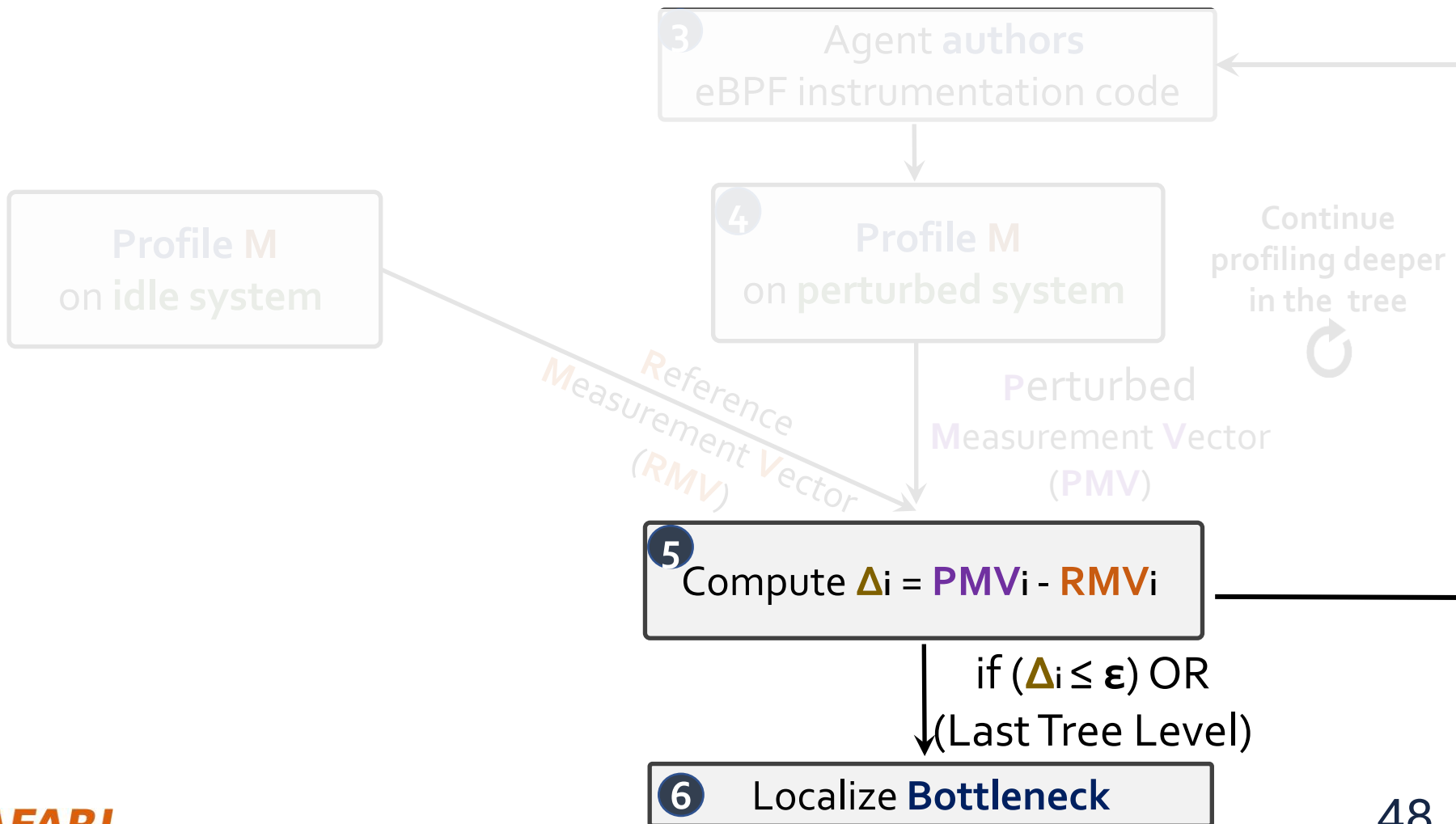
Argus – Reference Calibration (II)

Compares workload profile under perturbation (**PMV**) against profile on an idle system (**RMV**)



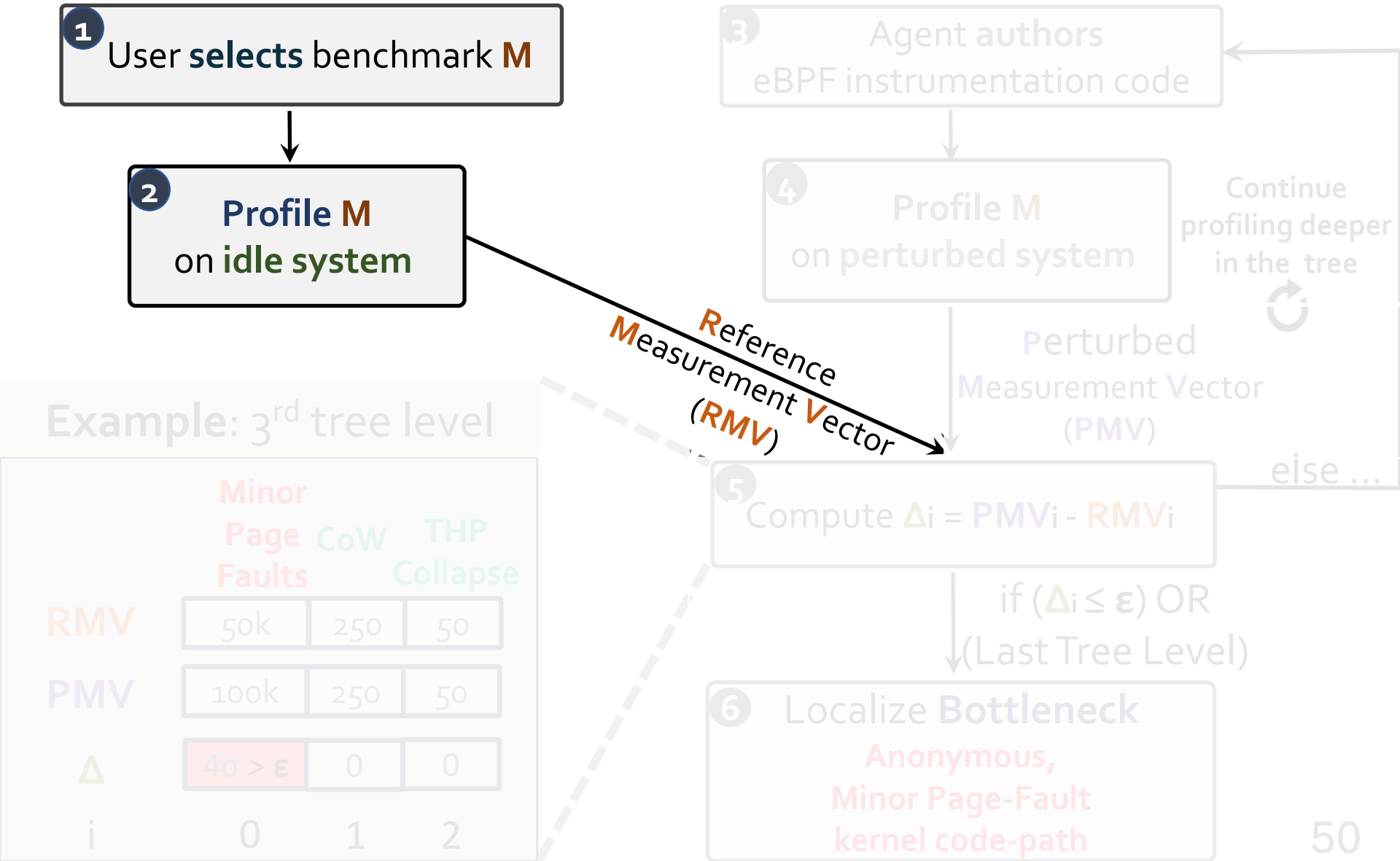
Argus – Reference Calibration (III)

Significant deviation → **bottleneck**

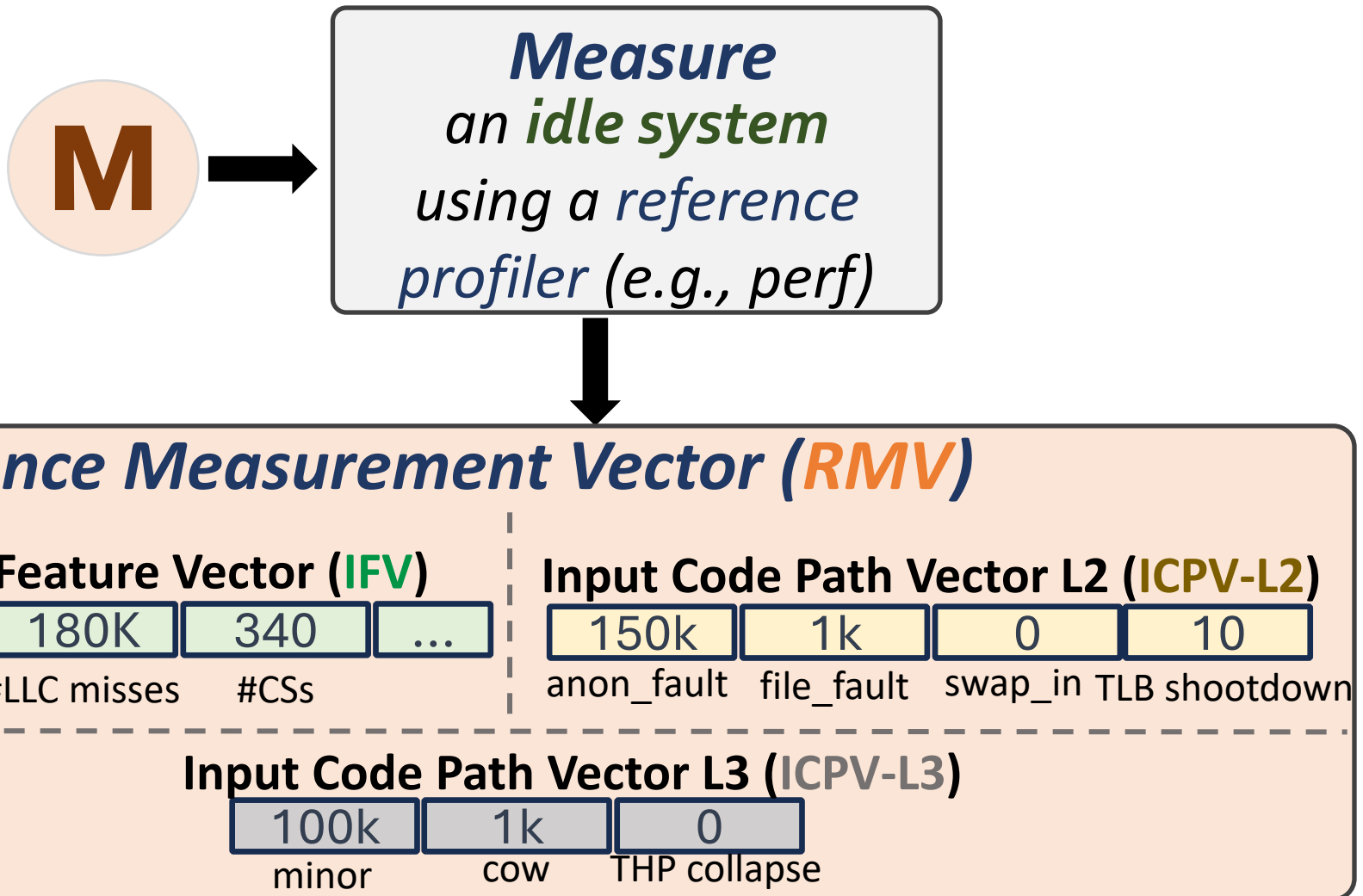


Example

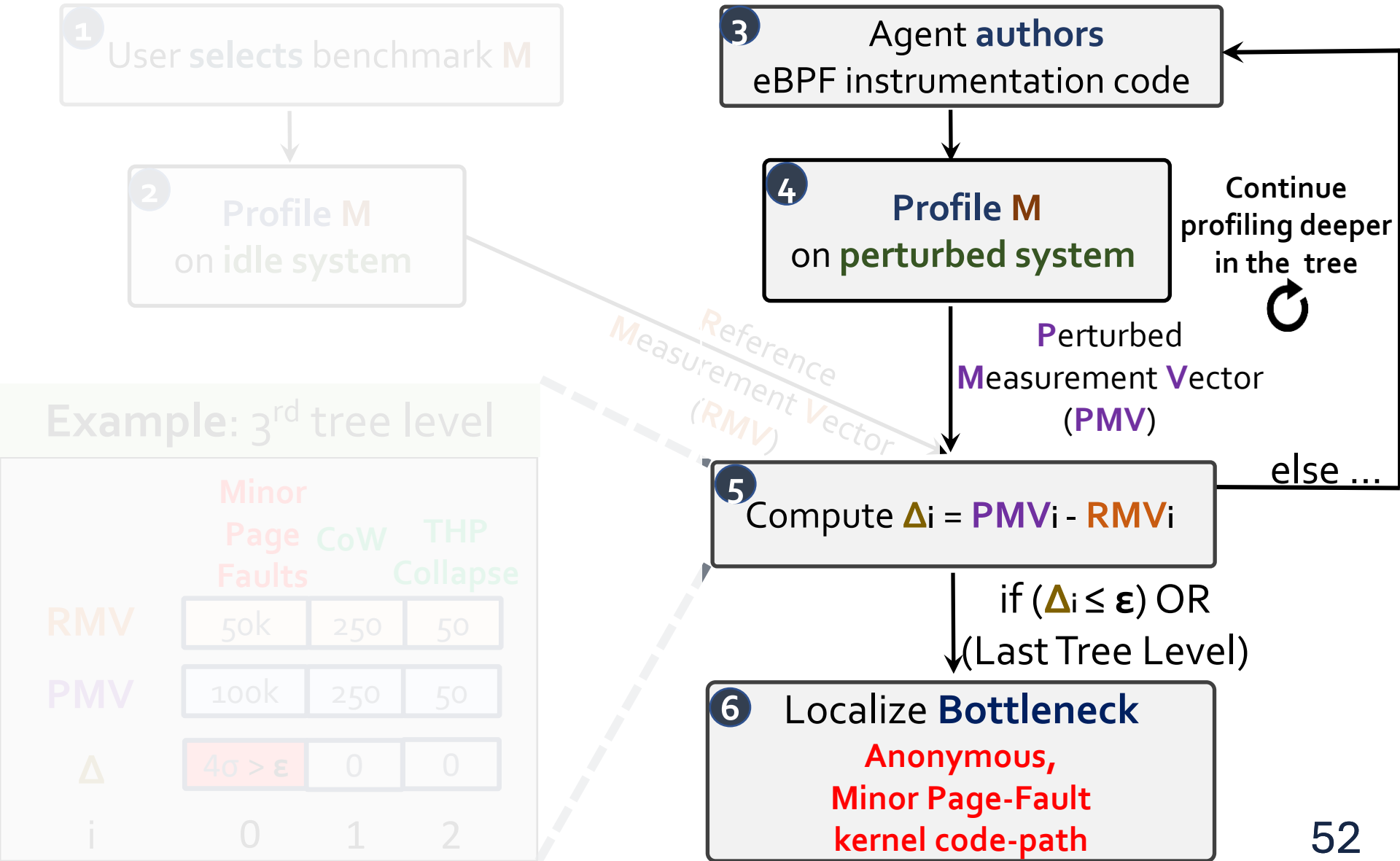
1. Reference Profiling



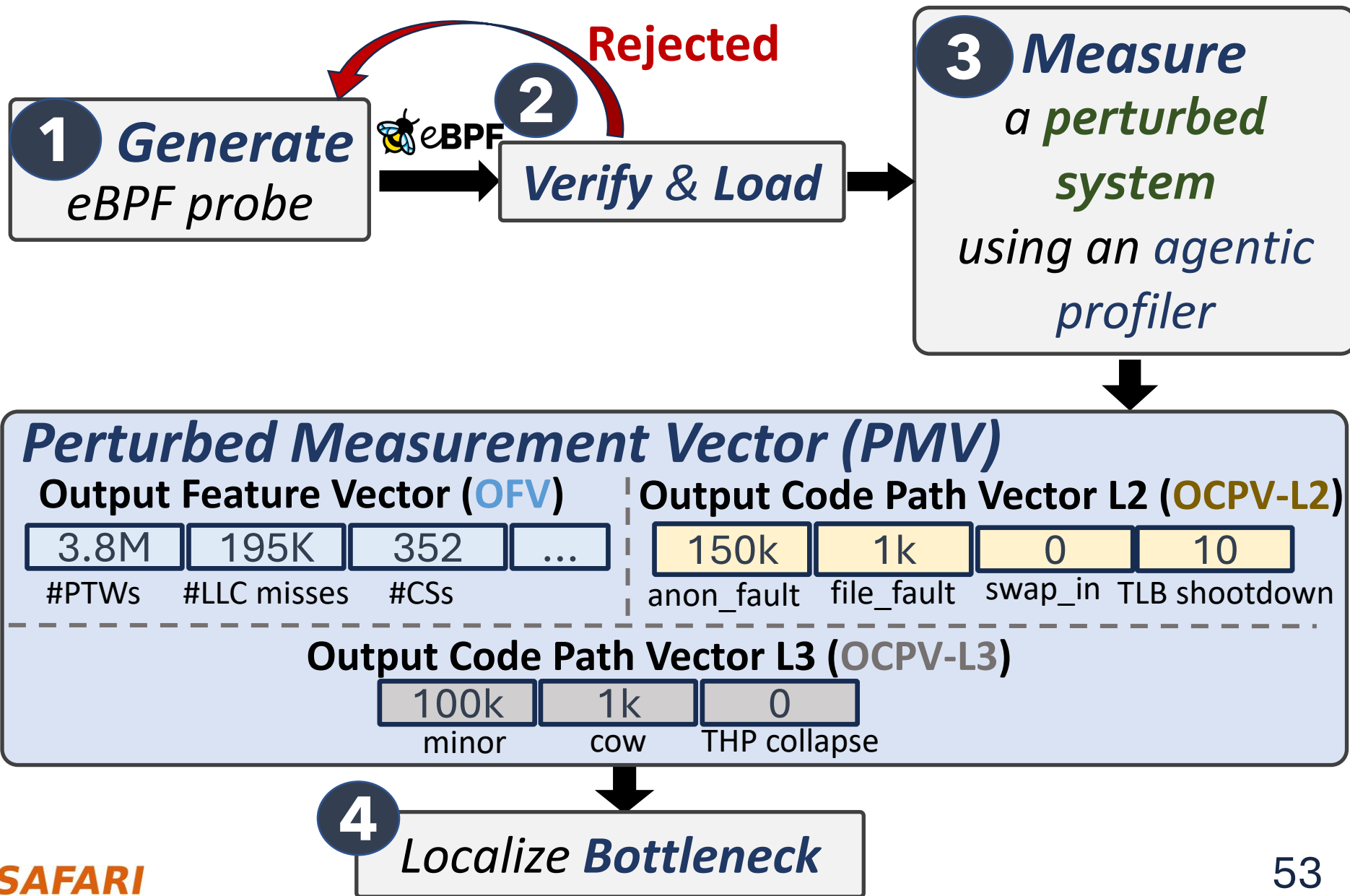
1. Reference Profiling



2. Agentic Profiling



2. Agentic Profiling



4. Bottleneck Detection (L1)

PTWs # LLC
 misses # Context
 Switches

Input Feature Vector (IFV)

1.2M	180K	340	...
------	------	-----	-----

Output Feature Vector (OFV)

3.8M	195K	352	...
------	------	-----	-----

Δ per metric

52 σ	1.9 σ	0.8 σ	...
-------------	--------------	--------------	-----



Kernel Paging is a high-level bottleneck

4. Bottleneck Detection (L2)

anon_fault file_fault swap_in TLB
shootdown

Input Code Path Vector L2 (ICPV-L2)

100k	1k	0	10
------	----	---	----

Code Path Vector L2 (CPV-L2)

150k	1k	0	10
------	----	---	----

Δ per metric

5 σ	0 σ	0 σ	0 σ
------------	------------	------------	------------



Anonymous page-fault path is a middle-level bottleneck

4. Bottleneck Detection (L3)

Minor CoW THP
Collapse

Input Code Path Vector L3 (ICPV-L3)

50k	250	50
-----	-----	----

Code Path Vector L3 (CPV-L3)

100k	250	50
------	-----	----

Δ per metric

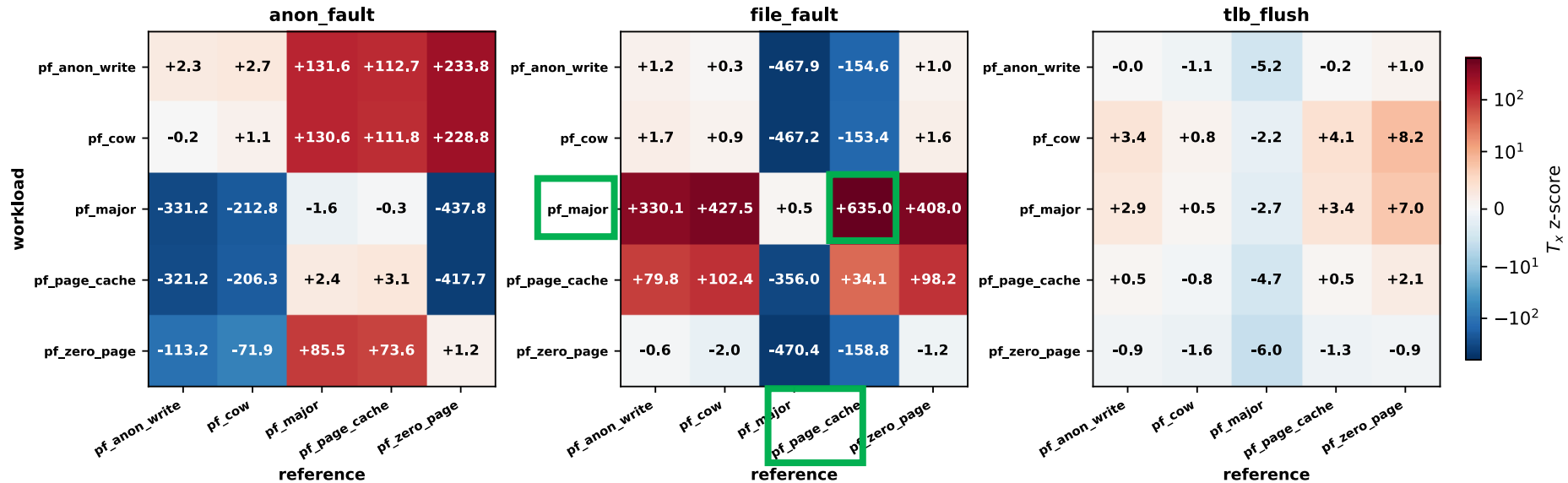
4σ	0σ	0σ
-----------	-----------	-----------



Final *Diagnosis* ↓

*Anonymous, Minor page-fault path is
the fine-grained bottleneck*

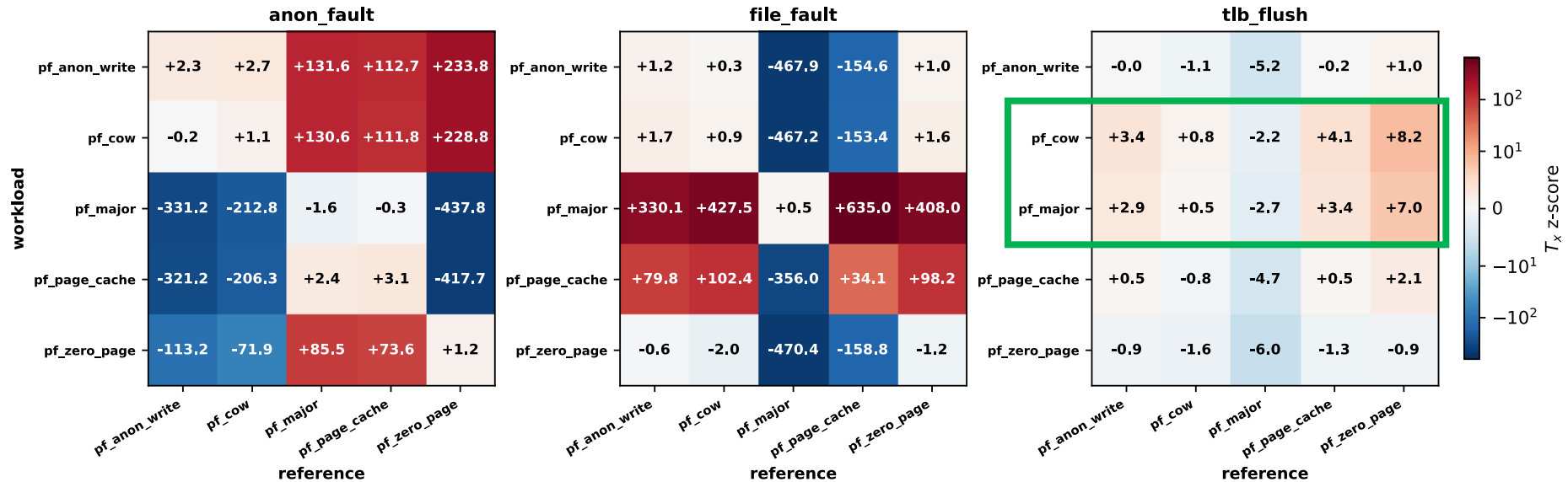
Case Study #2: Page-fault disambiguation



Key Takeaway #2

Major page faults stress the `file_fault` kernel code path substantially more than `page-cache` faults, separating disk-I/O-bound from cache-resident file faults

Case Study #2: Page-fault disambiguation



Key Takeaway #3

Significantly lower deviations in the **tlb_flush** kernel code path,
Since TLB invalidations are secondary events that fire only on
CoW and **major-fault remappings**