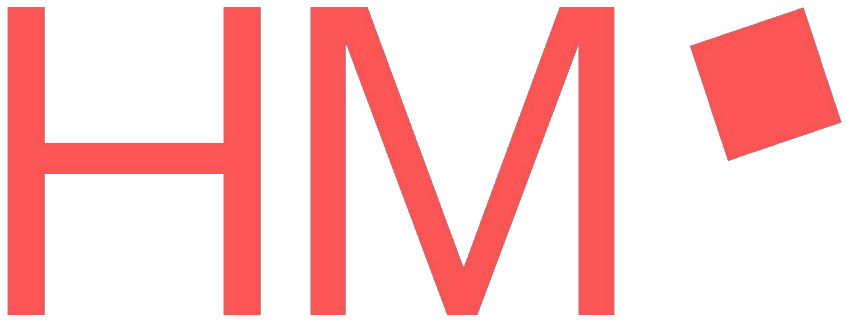
eBPF jAgent: Transaction-level Resource Demand Tracing for Java Applications Using eBPF and OpenTelemetry

Bryan Liegsalz, Andreas Brunnert
byteleaf GmbH · Munich University of Applied Sciences HM

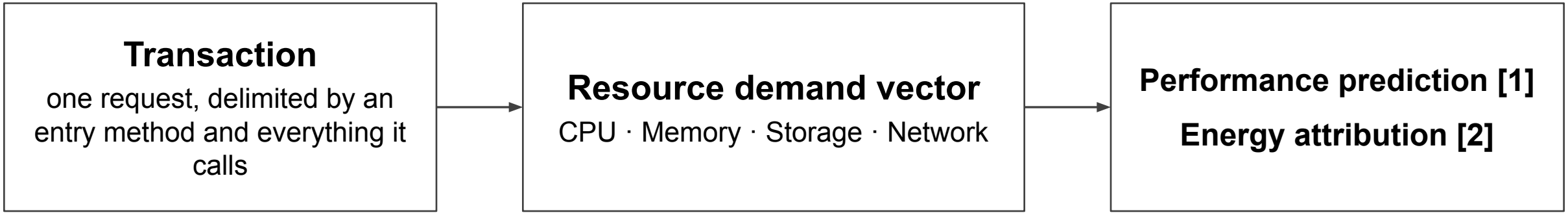
Symposium on Software Performance (SSP) 2026





Motivation

Per-transaction resource demands feed performance and energy models, but collecting them is tied to one runtime



Today

In-process instrumentation per framework or language, e.g. the OpenTelemetry Java Agent Extension (OTJAE) [2]

Must be re-implemented for every runtime it is to cover

Idea: collect in the kernel with eBPF

Runtime-independent and written once

But the kernel cannot see **where a transaction begins**

Related Work

Existing eBPF tools observe processes or wire requests, but none attributes resource demands to a transaction

	Transaction boundary	Resource demands	Collection
OTJAE [2]	Entry method (@WithSpan)	CPU, memory, storage, network	In-process, per runtime
OTel eBPF Profiler	None: periodic stack sampling	CPU only, statistical	Kernel, any runtime
OTel eBPF Instrumentation	Wire request at the socket	None: exports response time	Kernel, any runtime
eBPF jAgent	Entry method via HotSpot USDT	CPU, memory, storage, network	Kernel, no code changes

OTel = OpenTelemetry. OTJAE = OpenTelemetry Java Agent Extension. eBPF has also measured consumption per process, cgroup or container [3], but not per transaction.

What is eBPF?

eBPF runs small, verified programs inside the Linux kernel, without changing kernel code or loading kernel modules [6]

Safe and fast

`verifier + JIT compiler`

Checked before loading, then compiled to native code.

Event-driven

`kernel and user-space hooks`

A program runs whenever the hook it is attached to fires.

Shares data

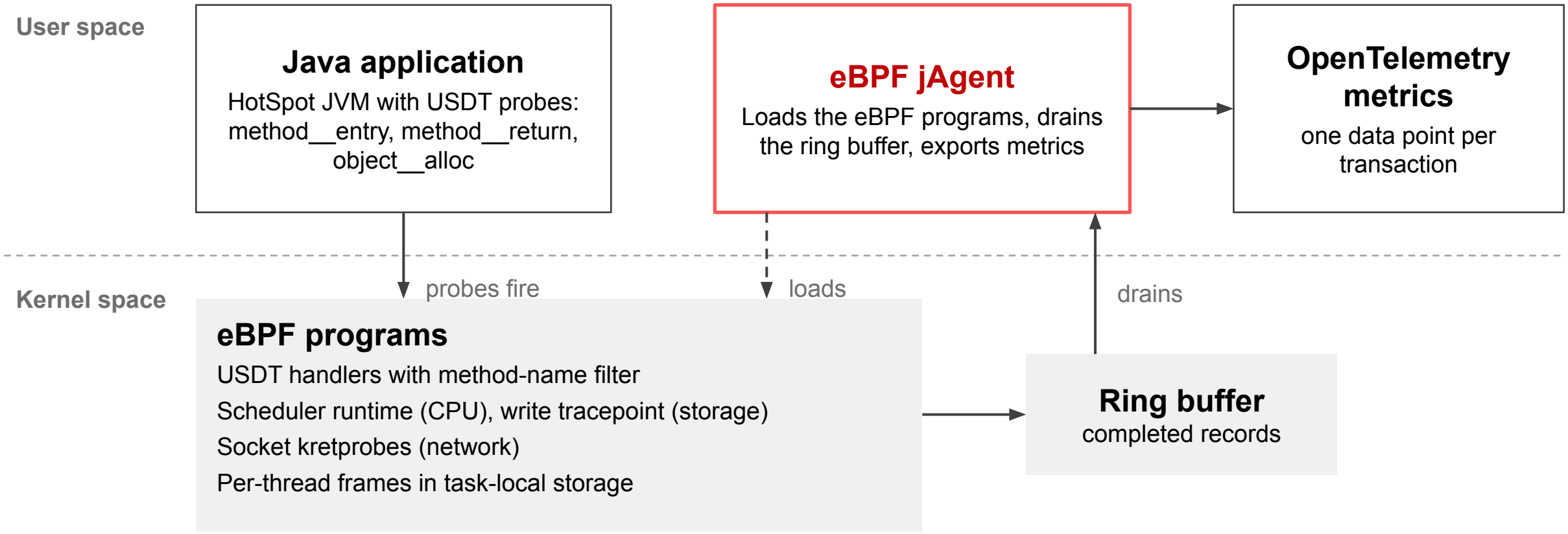
`maps, ring buffer`

Results reach user space through kernel data structures.

In the jAgent: eBPF programs on JVM and kernel hooks collect the demands; a user-space agent reads them from a ring buffer.

Architecture of the eBPF jAgent

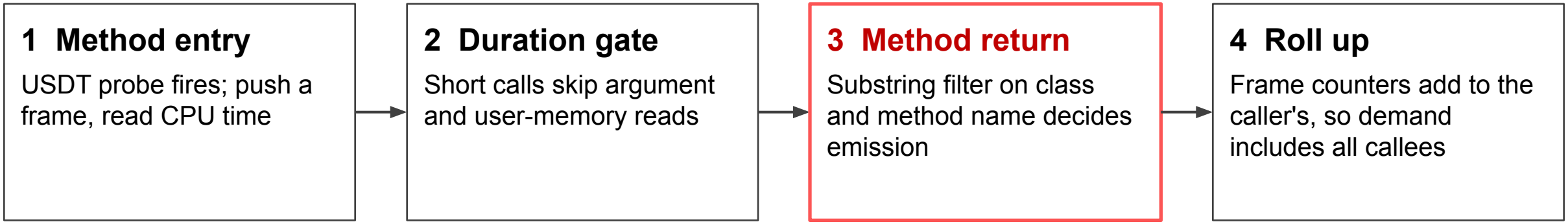
eBPF programs in the kernel collect the demands; a user-space agent exports one OpenTelemetry data point per transaction



Transaction Detection

HotSpot USDT probes mark method boundaries, because JIT-compiled methods cannot be probed with uprobes

Problem: HotSpot compiles methods into anonymous, non-file-backed memory; uprobes need file-backed mappings.
Solution: hotspot:method__entry / method__return USDT probes [4], enabled by `-XX:+DTraceMethodProbes`



Execution model: thread-per-request. Each request runs on one platform thread; per-thread state lives in eBPF task-local storage, freed on thread exit.

Not attributed: work handed to executors, virtual threads or other processes.

eBPF Probe Types and Resource Demand Collection

eBPF programs attach to static or dynamic hooks in the kernel and user space; the jAgent uses three of the four to collect the resource demands

kprobe / kretprobe `sock_sendmsg / sock_recvmsg`	Kernel, dynamic: entry or return of any kernel function; no stable interface across kernel versions	**Network:** kretprobes on kernel-wide functions, so attached only on request
Tracepoint `syscalls/sys_exit_write`	Kernel, static: hooks defined in the kernel source, stable across kernel versions	**Storage:** tracepoint on writes; reads excluded, as the page cache often serves them
uprobe / uretprobe `used only beneath USDT`	User space, dynamic: any function in a file-backed binary; not usable for JIT-compiled methods	**None:** no demand is collected with uprobes directly
USDT `method__entry, method__return, object__alloc`	User space, static: markers compiled into HotSpot, enabled by -XX:+DTraceMethodProbes and -XX:+DTraceAllocProbes	**Memory:** object size per allocating thread [4]; managed heap only, allocation volume rather than retained memory
Scheduler field `se.sum_exec_runtime`	Kernel, no probe of its own: read inside the USDT handlers on method entry and return	**CPU time:** corrected by the time since se.exec_start, as it updates only on descheduling

Memory Measurement: Approaches Considered

Three ways to attribute memory to a method were evaluated; only the object__alloc USDT probe ties allocations to the executing thread

1 Heap snapshots

`jcmd + GC USDT probes`

Heap usage at method entry and return, refined by the memory reclaimed in each GC cycle.

The heap is shared by all threads, GC runs asynchronously and jcmd snapshots are costly: too indirect and imprecise.

2 System-level syscalls

`mmap, munmap, brk`

Memory footprint of the whole JVM process: Σ mmap + Σ Δbrk − Σ munmap.

No link to a Java method or thread.

3 Object allocations

`hotspot:object__alloc`

Fires on every new Java object with its size and thread; attributed to the method on top of the per-thread stack.

Managed heap only (no JNI, off-heap or Unsafe memory); allocation volume rather than retained memory.

Selected: approach 3, the best compromise between precision and overhead. It gives object-level granularity, attributes per thread, and plugs into the method tracing already used for CPU time.

eBPF Probe Types

eBPF programs attach to static or dynamic hooks in the kernel and user space; the jAgent uses three of the four

kprobe / kretprobe

`sock_sendmsg / sock_recvmsg`

Kernel, dynamic: entry or return of any kernel function; no stable interface across kernel versions

Tracepoint

`syscalls/sys_exit_write`

Kernel, static: hooks defined in the kernel source, stable across kernel versions

uprobe / uretprobe

`used only beneath USDT`

User space, dynamic: any function in a file-backed binary; not usable for JIT-compiled methods

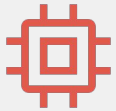
USDT

`method__entry, method__return, object__alloc`

User space, static: markers compiled into HotSpot, enabled by -XX:+DTraceMethodProbes and -XX:+DTraceAllocProbes

Resource Demand Collection

Each resource dimension comes from its own kernel or JVM event, read on method entry and return



CPU time

`se.sum_exec_runtime`

Scheduler field, corrected by the time since se.exec_start, as it updates only on descheduling



Memory

`object__alloc (USDT)`

Object size per allocating thread [4]; managed heap only, allocation volume rather than retained memory



Storage

`sys_exit_write`

Tracepoint on writes; reads excluded, as the page cache often serves them



Network

`sock_sendmsg / sock_recvmsg`

kretprobes on kernel-wide functions, so attached only on request

Experiment Setup

Six configurations, 12 repetitions each, on a dedicated bare-metal server under a fixed open workload

Testbed

- 2× Intel Xeon E5-2630 v4 (10 cores each), 256 GB RAM, OpenJDK 25; Proxmox VE host with all VMs stopped

Workload

- Spring-REST-Service from the OTJAE repository; transaction = `veryComplexBusinessFunction`
- JMeter on a separate host, open workload of 50 transactions/s

Protocol

- 4-minute runs, only the 2-minute steady state evaluated; 72 runs ordered by Latin square
- Paired exact Wilcoxon signed-rank tests, Bonferroni threshold $p < 0.0125$

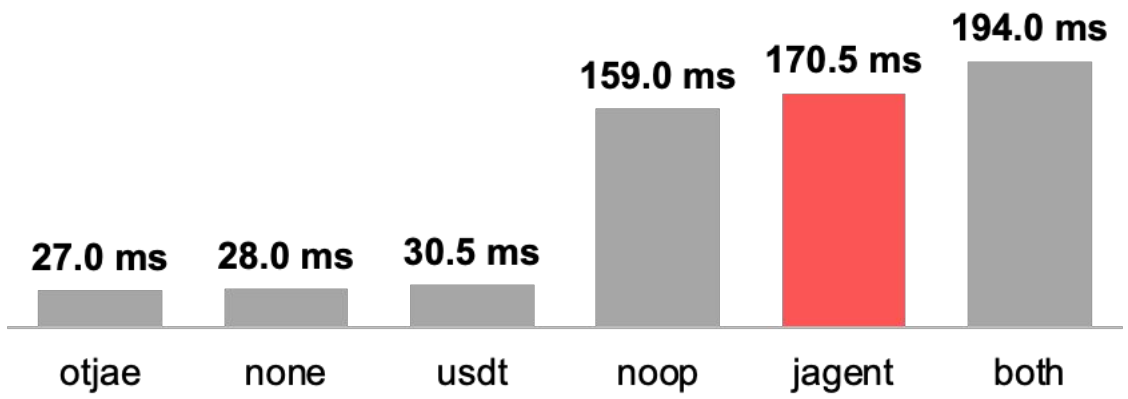
Configuration	What runs
`none`	Uninstrumented baseline
`usdt`	DTrace probe flags set, nothing attached
`noop`	Same probes, empty eBPF handlers
`jagent`	Full eBPF jAgent
`otjae`	In-process OTJAE baseline
`both`	Both agents on one JVM (control)

Comparison: CPU and memory of the eBPF jAgent vs. OTJAE, the language-specific in-process Java agent

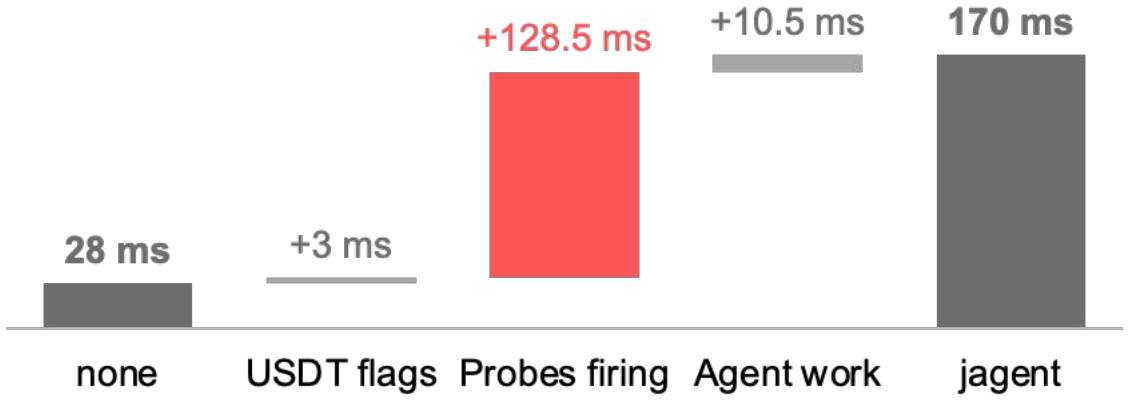
Results: Overhead and Where It Comes From

Active tracing raises the median response time 6.1-fold; 93% of the added latency already occurs with empty eBPF handlers

Median response time per configuration



Median response time, none to jagent (paired differences)



+10.7%	usdt vs. none: probe flags set, nothing attached
6.1×	jagent vs. none: 28 → 170.5 ms
p = 0.031	otjae vs. none: no measurable cost (above the corrected threshold of 0.0125)

12 repetitions per configuration at 50 transactions/s. both = both agents on one JVM (control, not deployable).

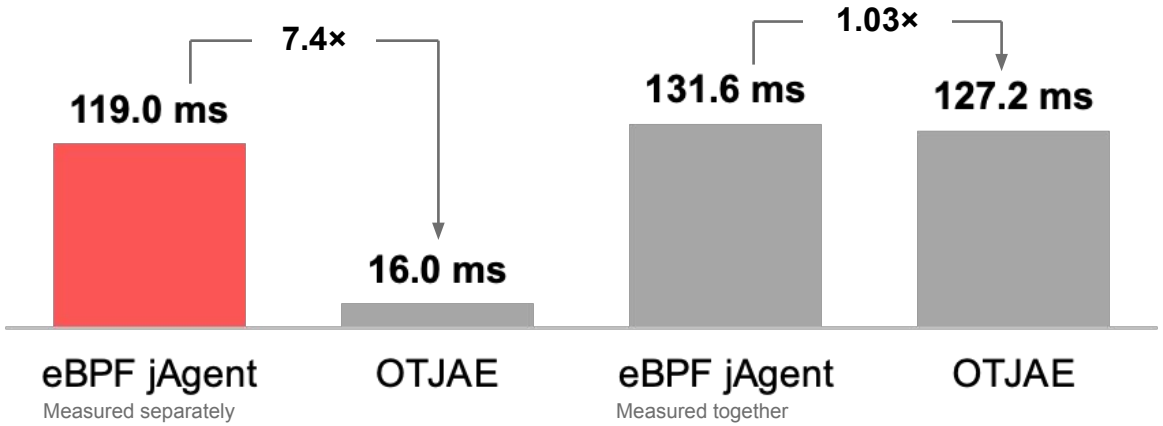
93%	of the 142.0 ms that jagent adds (131.5 ms) already occurs with empty handlers
10.5 ms	left for the agent's own bookkeeping
Cause	`-XX:+DTraceMethodProbes` traps into the kernel on every Java method call

Paired median differences against none; their sum (170.0 ms) differs from the jagent median (170.5 ms) by 0.5 ms.

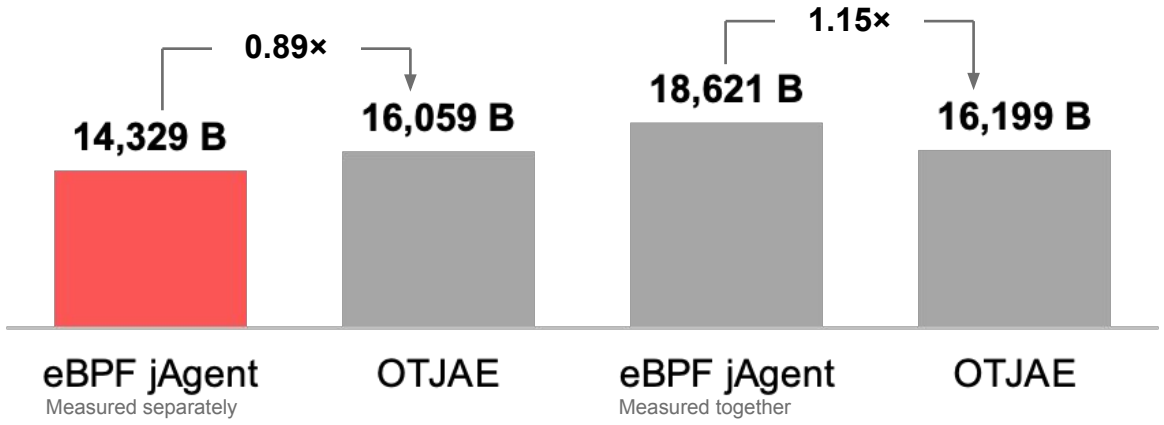
Results: Comparison with OTJAE and Probe Perturbation

Measured separately, the agents differ 7.4-fold on CPU and 11% on memory; on one JVM each reading also absorbs the other agent's work

CPU demand per transaction (mean of 12 per-run medians)



Memory allocated per transaction (mean of 12 per-run medians)



Both read the same kernel field

eBPF jAgent	`se.sum_exec_runtime` plus the current slice
OTJAE	`getCurrentThreadCpuTime()`, which the kernel answers from the same field
Scope	identical: both measure veryComplexBusinessFunction

Two different sources for allocated bytes

eBPF jAgent	`object_alloc` USDT probe: object size and thread
OTJAE	the JVM's allocation counter
Together	jAgent +30%: its probe also counts OTJAE's span objects; OTJAE +0.9%

Paired exact Wilcoxon signed-rank tests, p = 0.0005 for both comparisons.

Conclusions, Limitations and Future Work

eBPF tracing works for the JVM, but transaction detection limits it to pre-production



Conclusions

- **Feasible for the JVM:** per-transaction CPU, memory, storage and network without code changes
- **Detection drives the cost:** 93% of the added latency occurs with empty handlers



Limitations

- **Overhead:** transaction detection raises the median response time 6.1-fold
- **Concurrency:** virtual threads and executor hand-offs break the thread correlation
- **Scope:** one workload, so absolute figures do not generalize



Future Work

- **Statically compiled languages:** functions can be probed directly with uprobes
- **Probe cost in the readings:** quantify how much of the measured CPU time the probes themselves add

Q&A

Questions?

References

[1] Simon Spinner, Giuliano Casale, Fabian Brosig, Samuel Kounev. 2015. Evaluating approaches to resource demand estimation. Performance Evaluation 92, 51–71.

[2] Andreas Brunnert, Ferdinand Gutzy. 2024. Extending the OpenTelemetry Java Auto-Instrumentation Agent to Publish Green Software Metrics. In 15th Symposium on Software Performance (SSP) 2024, Linz, Austria.

[3] Marcelo Amaral et al. 2023. Kepler: A Framework to Calculate the Energy Consumption of Containerized Applications. In IEEE 16th International Conference on Cloud Computing (CLOUD 2023).

[4] Oracle. DTrace Probes in HotSpot VM. Java SE Troubleshooting Guide.

[5] eBPF jAgent: github.com/hm-green-it-lab/ebpf-jagent · Replication package: github.com/hm-green-it-lab/ssp2026

[6] eBPF.io. What is eBPF? An Introduction and Deep Dive into eBPF. ebpf.io/what-is-ebpf