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

Bryan Liegsalz
bryan.liegsalz@byteleaf.de
byteleaf GmbH, Munich, Germany

Andreas Brunnert
brunnert@hm.edu
Munich University of Applied Sciences HM, Germany

Abstract

Collecting resource demands for individual software transactions is a prerequisite for accurate performance models and software energy attribution, yet existing solutions rely on framework-specific in-process instrumentation or cover a single resource dimension. This paper presents `eBPF jAgent`, a prototype using the extended Berkeley Packet Filter (eBPF) to measure CPU, memory, storage and network demands for Java transactions from within the Linux kernel, without modifying application code. Transaction boundaries come from HotSpot User Statically-Defined Tracing (USDT) probes; the demands are exported via OpenTelemetry. In a randomized campaign of 72 runs, passive USDT scaffolding increases the median request latency by 10.7 % and active tracing raises it 6.1-fold; a control with empty eBPF handlers attributes 93 % of that cost to the probes firing rather than to the agent's own work. Tracing also perturbs the demands themselves: an in-process agent and the `eBPF jAgent` report CPU demands 7.4-fold apart when measured separately, yet agree within 3 % on the same JVM because the probe overhead is included in both readings.

1 Introduction

Performance prediction [1] and software energy attribution [3] often rely on accurate resource demand measurements. Yet while many tools monitor host- or process-level utilization, measuring consumption per transaction still requires instrumentation tied to a framework or language, such as the OpenTelemetry Java Agent Extension (OTJAE) [3]. Such tooling must be implemented anew for every runtime it is to cover. Kernel-side collection is in principle runtime-independent and written once; what it cannot determine by itself is where a transaction begins.

We therefore present `eBPF jAgent`[1], a prototype that attaches eBPF probes to kernel and user-space events to capture per-transaction CPU, memory, storage and network demands, exporting OpenTelemetry metrics without application changes. The JVM is the hardest case for such a collector, as just-in-time compiled methods expose no file-backed symbols to probe; establishing a kernel-side transaction boundary for it, and quantifying what it costs, is the contribution of this paper. OTJAE serves as the baseline for fidelity and overhead.

2 Background and Related Work

Following [3], a *transaction* is here a logical unit of work triggered by one request, delimited in the code by a designated entry method and everything it calls; it is neither a database nor a memory transaction. The prototype treats one invocation of a name-filtered Java method as a transaction. Its resource consumption forms a vector over CPU time, memory allocations, storage I/O and network usage, which the methodology of [3] collects in all four dimensions, but from inside the JVM.

The extended Berkeley Packet Filter (eBPF)[2] runs sandboxed, verified programs inside the Linux kernel, attached to kernel functions (kprobes), user-space functions (uprobes), tracepoints or User Statically-Defined Tracing (USDT) probes, and can therefore observe applications without language-specific instrumentation. It has been used to measure resource consumption per process, cgroup or container for energy attribution [2], but not per transaction.

Two OpenTelemetry projects apply eBPF to running applications, but neither attributes resource demands to a transaction. The eBPF Profiler[3] samples stack traces periodically: the result is statistical, aggregated over a window, observes no request boundary and covers only CPU. The eBPF Instrumentation[4] reconstructs transactions by parsing the protocols crossing socket calls, so its transaction is a wire request seen from outside the process: the methods inside it are never observed, and response time is exported instead of resource demand. OTJAE, which does attribute all four dimensions per transaction, therefore serves as our baseline.

[1] `https://github.com/hm-green-it-lab/ebpf-jagent`

[2] `https://ebpf.io/what-is-ebpf/`

[3] `https://github.com/open-telemetry/opentelemetry-ebpf-profiler`

[4] `https://github.com/open-telemetry/opentelemetry-ebpf-instrumentation`

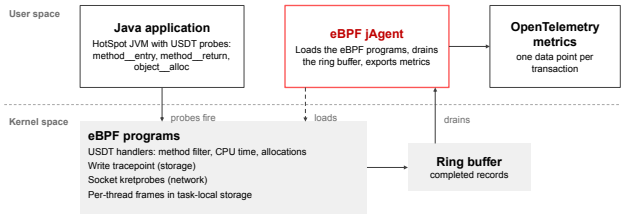

Figure 1: High-level architecture of the `eBPF jAgent`

3 Design of the eBPF jAgent

The architecture (Figure 1) pairs eBPF programs in kernel space with a user-space agent that loads them, drains completed method records from a ring buffer and exports them as OpenTelemetry metrics[5], one data point per transaction.

3.1 Transaction Detection

Detecting transaction boundaries with eBPF is complicated by just-in-time compilation: HotSpot compiles methods into anonymous, non-file-backed memory regions[6], whereas uprobes require file-backed mappings, so Java methods cannot be probed directly. The prototype therefore uses the `hotspot:method__entry` and `hotspot:method__return` USDT probes [4], stable points passing pointers to the class and method name, enabled by `-XX:+DTraceMethodProbes` and `-XX:+DTraceAllocProbes`.

Tracing every invocation would produce an impractical event volume, so a substring filter over class and method names prunes events in the kernel; it is evaluated on return and suppresses only emission, while a minimum-duration gate applied first skips the argument and user-memory reads for the short calls that dominate the event count.

Per-thread state lives in eBPF task-local storage, which the kernel frees on thread exit, so abandoned frames cannot accumulate. It holds a call-depth counter and per-frame resource counters; on return a frame's counters are added to its caller's, so a method's demand includes that of its descendants. The transaction is the filtered method, exported as a cumulative sum per class and method name; the filter must match at most one frame per stack, or nested records would contain one another.

This model assumes a thread-per-request runtime in a single process: each request is served by one platform thread that stays the same kernel task for the duration of the traced method. Requests may run concurrently, each tracked independently in its own task-local state, but work handed to an executor, to virtual threads or to another process leaves the frame that caused it and is not attributed to it.

[5] `https://opentelemetry.io/docs/specs/otel/metrics/`

[6] `https://github.com/jvm-profiling-tools/perf-map-agent/issues/26`

Configuration	Median	Δ	rel.	p
`otjae`	27.0	−0.5	−1.8 %	0.031
`none`	28.0	—	—	—
`usdt`	30.5	+3.0	+10.7 %	0.0005
`noop`	159.0	+131.5	+470 %	0.0005
`jagent`	170.5	+142.0	+507 %	0.0005
`both`	194.0	+166.5	+595 %	0.0005

Table 1: Median response time (ms) and paired difference against `none`, 12 repetitions.

3.2 Resource Demand Collection

CPU time is derived from the scheduler's `se.sum_exec_runtime` field, read on method entry and return and corrected by the time since `se.exec_start`, as the field is updated only on scheduler events. **Memory** comes from the `object__alloc` USDT probe [4], reporting object size and allocating thread; only managed heap allocations are covered, and allocation volume is a proxy for, not a measurement of, retained memory. **Storage** comes from the `sys_exit_write` tracepoint, reads excluded as they are often served from the page cache, and **network** from kretprobes on `sock_sendmsg` and `sock_recvmsg`, both switchable off as they hook kernel-wide functions.

4 Evaluation

4.1 Experiment Setup

Measurements were taken on a bare-metal server with two Intel Xeon E5-2630 v4 processors (10 cores each, 2.20 GHz base, 3.10 GHz turbo), 256 GB RAM and OpenJDK 25, running the Debian-based Proxmox Virtual Environment[7] directly with its virtual machines stopped, so nothing else on the host competes for the resources being measured. With two sockets, memory access is non-uniform (NUMA) and no pinning was applied; cross-socket effects are thus possible, but apply to all configurations alike and are paired away within a repetition.

The workload is the `Spring-REST-Service`[8] shipped with the OTJAE repository, whose `veryComplexBusinessFunction` is the measured transaction: a single code path whose duration varies only with its random input, so path diversity does not confound the comparison. A JMeter driver on a separate host sends an open workload of 50 requests per second to an HTTP GET endpoint that calls this method once, so each REST call is one transaction. Each run lasts four minutes (one ramp-up, two steady, one ramp-down) and only the steady-state window is evaluated; a replication package is online[9].

Six configurations were measured: **none**, uninstrumented; **usdt**, the DTrace probe flags set but nothing attached; **noop**, the same probes with empty eBPF handlers; **jagent**, the full prototype; **otjae**, the in-process baseline; and **both**, the two agents on one

[7] `https://www.proxmox.com/`

[8] `https://github.com/RETIT/opentelemetry-javaagent-extension/tree/v0.1.1-beta/examples/spring-rest-service`

[9] `https://github.com/hm-green-it-lab/ssp2026`

	eBPF jAgent	OTJAE	Ratio
measured separately			
CPU (ms)	118.99	15.99	7.44
Memory (B)	14,329	16,059	0.89
measured together			
CPU (ms)	131.56	127.21	1.03
Memory (B)	18,621	16,199	1.15

Table 2: Per-transaction demands, means of 12 per-run medians.

JVM at once. The campaign comprises 12 repetitions of all six, their order rotated across repetitions (a Latin square) so that every configuration occupies every position exactly twice. Comparisons are paired and tested with a two-sided exact Wilcoxon signed-rank test over the 12 per-run medians; its minimum, $p = 0.0005$, means slower in every repetition, and Bonferroni correction for four comparisons (`both` being a control) gives a threshold of 0.0125.

4.2 Overhead

Table 1 shows that the USDT scaffolding alone raises the median response time by 10.7 %, active tracing 6.1-fold. The `noop` control separates the contributions: of the 142.0 ms the `eBPF jAgent` adds, 131.5 ms (93 %) is already incurred with empty handlers, leaving 10.5 ms for the agent's own bookkeeping. Since `-XX:+DTraceMethodProbes` instruments every Java method invocation, every invocation traps into the kernel; a run with `object__alloc` unattached showed no change, so that floor is the method probes almost entirely. OTJAE, by contrast, is not distinguishable from the baseline at this load: six of the twelve differences are exactly zero, so the exact test runs on six pairs and yields $p = 0.031$, above the threshold.

4.3 Comparison with OTJAE

Accuracy is assessed on CPU and memory, storage and network being omitted as their semantics differ. Both tools are scoped to `veryComplexBusinessFunction`, OTJAE by `@WithSpan`, the jAgent by its filter, so the windows coincide.

Table 2 contrasts two ways of making this comparison. Measured separately, the tools disagree on CPU by a factor of 7.4; on the same JVM they agree to 3 %, because both read the same kernel field: the jAgent computes `se.sum_exec_runtime` plus the current slice, and OTJAE calls `ThreadMXBean.getCurrentThreadCpuTime()`, which the kernel answers from it. The probes fire on the measured thread, so their cost sits inside both readings and cancels out; agreement within a run therefore says nothing about accuracy. Attaching the jAgent alongside OTJAE indeed raises OTJAE's reading by 111.2 ms, close to the 103.0 ms by which the two differ in separate runs.

Memory is perturbed in the opposite direction: OTJAE's reading barely moves when the two run together (+0.9 %) whereas the jAgent's rises by 30 %, its allocation probe counting the span objects OTJAE allocates. The residual 11 % gap reflects the sources: allocation events versus the JVM's allocation counter.

5 Discussion and Limitations

Purpose and acceptable use. The collection targets offline resource demand characterization: replaying a workload pre-production to parameterize a performance or energy model once, not continuous production monitoring. There a 6.1-fold latency penalty costs measurement turnaround rather than production service levels. For production monitoring the approach is not usable, and tuning would not make it so: with 93 % of the cost incurred before any agent code runs, the levers lie outside the agent. Filtering before the probe fires would help most, but `-XX:+DTraceMethodProbes` is all-or-nothing; a marker firing once at request entry would remove the floor, at the price of the runtime-specific component the present design avoids.

Limitations. Some are structural. The probe overhead inflates the measurement itself and cannot be subtracted inside the handler, most of it being trap and uprobe dispatch during which no eBPF program runs; it can only be estimated from a firing count calibrated against `noop`. So is the execution model of Section 3.1: virtual threads, executor hand-offs and distributed contexts break a correlation keyed to a kernel task and would need propagated context instead. Native and off-heap memory, by contrast, could be covered by further probes, and eBPF's privileges are a deployment rather than a design constraint. All results come from one workload and endpoint, so the absolute figures do not generalize, though the firing rate driving them is a property of the runtime.

6 Conclusion

Transaction-level resource tracking for JVM applications with eBPF is feasible, and its cost lies in the method probes rather than in the agent. The transaction definition used here is runtime-independent; its detection is not: other runtimes would need suitable USDT probes, file-backed function boundaries or another reliable marker, and a function boundary need not coincide with a transaction boundary. Evaluating the approach outside the JVM, and quantifying the residual probe overhead, remain future work.

References

[1] S. Spinner et al. "Evaluating approaches to resource demand estimation". In: *Performance Evaluation* 92 (2015).

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

[3] A. Brunnert and F. Gutzy. "Extending the OpenTelemetry Java Auto-Instrumentation Agent to Publish Green Software Metrics". In: *15th Symposium on Software Performance*. 2024.

[4] Oracle, Inc. *DTrace Probes in HotSpot VM*. 2025.