B-Perf: Black-box Performance Antipattern Detection Using System-level Execution Tracing

Morteza Noferesti
Computer Science
Brock University
St.Catharines, Canada
mnoferesti@brocku.ca

Mahsa Panahandeh
School of Electrical Engineering and Computer Science
University of Ottawa
Ottawa, Canada
mpanahan@uottawa.ca

Naser Ezzati-Jivan
Computer Science
Brock University
St.Catharines, Canada
nezzatijivan@brocku.ca

ABSTRACT

Performance antipatterns capture recurring behaviours that degrade software efficiency. Black-box approaches aim to detect such issues without modifying the application. This paper presents B-Perf, a system-level black-box method that reconstructs execution, memory, and messaging behaviour from kernel-level traces. By analysing scheduling, allocation, and communication events, B-Perf derives workload-dependent behavioural trends and reports antipattern indicators grounded in resource usage and contention. To handle large trace volumes, the approach follows a pipeline of workload generation, event gathering, trace handling, and antipattern inference.

We evaluate B-Perf on three representative antipatterns—One Lane Bridge, Empty Semi Trucks, and Excessive Dynamic Allocation—and apply it to traces from real multi-threaded applications. The results show that system-level events are often sufficient to expose bottlenecks linked to resource contention and system-level interactions. A key limitation is that kernel traces provide limited visibility into fine-grained in-process behaviour. When performance issues are driven by internal logic or function-level interactions, B-Perf may capture only indirect symptoms and may not reveal the full root cause. Within this scope, B-Perf provides practical and efficient black-box detection for antipatterns driven by resource interaction and competition.

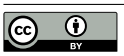
CCS CONCEPTS

• **Software and its engineering → Software performance**.

KEYWORDS

Software performance antipattern; Performance analysis; Black-box monitoring; Execution tracing; Software analysis

ACM Reference Format:
Morteza Noferesti, Mahsa Panahandeh, and Naser Ezzati-Jivan. 2026. B-Perf: Black-box Performance Antipattern Detection Using System-level Execution Tracing. In *Proceedings of the 17th ACM/SPEC International Conference on Performance Engineering (ICPE '26), May 04–08, 2026, Florence, Italy*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3777884.3797014



ICPE '26, May 04–08, 2026, Florence, Italy

ACM ISBN 979-8-4007-2325-4/2026/05.
https://doi.org/10.1145/3777884.3797014

1 INTRODUCTION

The performance of enterprise software systems is a persistent concern for both vendors and operators. Modern applications must handle large volumes of input data and serve many concurrent users, which places significant pressure on their underlying architectures. Detecting performance problems before they affect end users remains difficult in practice [25]. System logs and runtime signals are often noisy, uncorrelated, or incomplete, making root-cause identification challenging [14]. Prior studies have shown that performance is a central factor in the success of software products and services [28], yet diagnosing performance issues still requires considerable expertise and effort.

Despite its importance, performance evaluation is often deferred until late stages of development or omitted entirely due to the complexity of workloads and the overhead of measurement [12]. When problems are discovered during operation, resolving them becomes expensive and can harm the reliability and reputation of the system. In contrast, proactive approaches aim to surface performance issues early. Model based techniques such as Queueing Networks, Stochastic Petri Nets, and Queueing Petri Nets support reasoning about performance during the design phase [25], but the gap between these formalisms and real software constructs has limited their adoption in practice.

A complementary line of work focuses on recurring performance mistakes known as Software Performance Antipatterns (SPAs). SPAs document typical performance problems and their solutions [4]. These issues often arise from architectural or design choices but become visible only when concrete implementations are exercised under load. Prior work has shown that SPAs can manifest in several observable behaviours, including messaging, service execution, concurrency, and memory usage [11]. Recent studies connect antipatterns to misconfigurations and architectural decisions in microservice systems [4, 24], reinforcing their continued practical relevance.

Measurement-based detection techniques analyse runtime activity to identify these patterns. White-box approaches [24] insert instrumentation or modify program code to expose detailed behaviour, but they cannot be applied when source code is unavailable or when changing the system is undesirable. This limitation has motivated black-box techniques [13, 18] that infer performance behaviour from system calls and kernel-level events. Modern tracing frameworks, including LTTng [5] and eBPF-based observability [17], demonstrate that kernel events provide rich behavioural insight with low overhead. Such methods avoid code-level dependencies

and work with proprietary or binary-only systems, although the diversity and volume of system-level events make accurate inference challenging. Prior work on structure-aware log analysis highlights similar challenges in deriving reliable performance insights from raw system activity [14]. These challenges mirror broader concerns in trustworthy automated performance diagnosis, where reliable inference must be drawn from low-level or indirect signals [28].

This paper presents **B-Perf**, a black-box performance antipattern analysis method based on system-level execution tracing. B-Perf observes how software interacts with CPU, memory, and messaging resources and reconstructs behavioural projections from kernel events. The method derives execution, memory, and messaging behaviour summaries without access to internal program structure and reports antipattern indicators from workload-driven trends. We also examine whether these indicators remain informative on complex multi-threaded real-world traces, beyond controlled antipattern implementations.

B-Perf is designed for antipatterns that manifest as resource interaction and contention. When performance issues are driven by fine-grained in-process logic, kernel visibility is limited and B-Perf may capture only indirect symptoms. Portability across tracing backends also depends on whether comparable scheduling, synchronization, memory, and network event semantics are available.

To structure our evaluation, we study four research questions. RQ1–RQ3 examine whether execution, memory, and messaging projections derived from system-level events can distinguish antipattern implementations from matched baselines under increasing workloads. RQ4 investigates whether the same projections remain informative on complex multi-threaded web applications and publicly available traces where only system-level events are observable.

The central question of this work is whether system-level traces provide enough behavioural information to support black-box antipattern diagnosis. We treat detection as inference of behavioural indicators rather than definitive classification, since real systems can exhibit overlapping effects. Our main contributions are:

(1) A unified system level trace based analysis. B-Perf shows that kernel events can be sufficient to derive key behavioural projections without source c

ode, application-level instrumentation, or knowledge of internal architecture.

(2) A behavioural reconstruction pipeline from raw events. The method derives execution, memory, and messaging projections from raw system-level events and links these projections to the characteristic symptoms of known antipatterns.

(3) Cross pattern detection using a single black-box mechanism. The evaluation demonstrates that B-Perf can identify representative performance antipatterns, including One Lane Bridge, Excessive Dynamic Allocation, and Empty Semi Trucks, by observing how reconstructed behaviour changes under increasing workloads.

(4) An empirical study of system-level tracing for black-box performance diagnosis. Across controlled experiments, B-Perf captures behavioural differences between antipattern and baseline implementations with low tracing and analysis overhead, which supports the use of system-level traces as a basis for black-box performance diagnostics.

The remainder of the paper is structured as follows. Section 2 reviews performance antipatterns and existing black-box detection approaches. Section 3 presents the B-Perf methodology. Section 4 evaluates the approach on several antipattern scenarios. Section 5 discusses implications and limitations, and Section 7 concludes the paper.

2 RELATED WORK

This section reviews prior research on software performance antipatterns and black-box detection methods.

2.1 Software Performance Antipatterns

SPAs describe recurring performance problems arising from inefficient design or resource use [25]. Foundational work by Smith and Williams documents patterns such as One Lane Bridge (OLB), Traffic Jam, Ramp, and Blob [19, 20]. These antipatterns span multiple observable behaviour scopes, including messaging, service execution, and memory [4].

Subsequent work generalised OLB into the N-Lane Bridge (NLB) model [23], showing that system-level execution traces can expose thread contention and serialized execution. Messaging-related issues such as Blob and Empty Semi Trucks (EST) introduce overhead through centralised retrieval or many small-message transmissions [2]. Memory-related antipatterns such as Excessive Dynamic Allocation and Dormant References capture allocation churn, fragmentation, and memory leaks [15, 16].

These studies show that performance problems can manifest across multiple behavioural scopes. However, most detection approaches inspect only one scope at a time and rely on application-level information or instrumentation.

2.2 Black-Box Detection Approaches

Measurement-based SPA detection uses runtime observations to identify performance symptoms. Systems-log comparison techniques also show the difficulty of diagnosing performance problems using coarse or noisy behavioural signals [14].

White-box approaches [24] instrument or modify code to capture fine-grained internal behaviour, but they require source access and thus do not apply to black-box scenarios [13].

Profiling-based methods collect periodic samples of program state. Trubiani et al. [22] used controlled load tests and profiling data to detect patterns such as Circuitous Treasure Hunt and Extensive Processing. While useful for high-level analysis, sampling misses short blocking intervals and kernel-level interactions. Studies of synchronization bugs show that many performance problems arise from fine-grained locking and thread interactions that are difficult to capture using periodic sampling alone [3].

Userspace tracing approaches [26] instrument function boundaries in the application to compute performance metrics, but they lack visibility into kernel scheduling, synchronization, and system calls—key elements for understanding multithreaded contention.

Fault-injection frameworks such as PPInject [12] artificially inject performance degradations (e.g., Ramp, OLB) to support evaluation of detection techniques. They provide controlled test scenarios but require detailed knowledge of application internals, conflicting with black-box constraints.

Table 1: SPA detection approaches and behavioural scopes.

Name	Service execution	Messaging	Memory
Trubiani et al. [22]	√	–	–
Wert et al. [26]	√	–	–
Keck et al. [12]	√	–	√
Wert et al. [27]	–	√	–
VanDonge et al. [23]	√	–	–
B-Perf	√	√	√

Dynamic Spotter [27] automates the detection of messaging-centric antipatterns (Blob, EST, DB overhead) using heuristics and generated workloads. However, its analysis depends on application-level instrumentation and does not address execution or memory-specific behaviour.

System-level tracing offers a more general black-box view. Host system tracing has also been used to characterize workload behaviour in containerized environments without modifying applications [10]. LTTng provides low-impact, fine-grained kernel visibility [5], and a recent work on eBPF shows growing interest in in-kernel observability of request-level metrics [17].

In our previous work [23], we used kernel events and dependency graphs [8] to detect OLB and NLB by analysing thread blocking relationships. That study also analysed a Firefox trace using response-time-based N-Lane Bridge analysis. In this paper, we reuse the same Firefox trace dataset as an external case study, and we apply B-Perf's multi-scope behavioural projections and trend-based inference.

Recent surveys on AI-based performance diagnosis further emphasise the need for reliable, low-level behavioural signals when source code is unavailable [21, 28].

Many existing tracing and profiling tools provide rich low-level views, but they do not explicitly map cross-scope trends under workload to antipattern-oriented indicators. B-Perf complements these tools by adding trend-based inference over execution, memory, and messaging signals in a black-box setting.

Existing approaches often focus on one behavioural scope—execution concurrency, messaging efficiency, or memory use—and typically require some form of application-level instrumentation. As summarised in Table 1, there is no unified black-box framework capable of analysing execution, messaging, and memory behaviour using kernel-level traces alone.

B-Perf addresses this gap by reconstructing execution, memory, and messaging behaviour directly from kernel events and applying trend-based inference across workloads. This enables detection of a wider range of SPAs without source code access, code annotations, or userspace instrumentation.

3 METHODOLOGY

The design of B-Perf is driven by practical constraints that arise in real deployments. Many systems are treated as black boxes, depend on complex libraries, or cannot be instrumented safely or consistently in production. In these settings, developers often have access only to behaviour that is exposed at the operating-system level. These constraints motivate four design goals for B-Perf: (G1) operate in black-box settings, (G2) remain portable across tracing backends, (G3) reconstruct execution, memory, and messaging behaviour from kernel-level events, and (G4) support trend-based inference over ordered workloads.

First, to satisfy G1, the analysis must remain black-box. B-Perf relies only on system-level interactions with the operating system, rather than internal application logic or source code. This makes the approach applicable to closed-source systems and deployed binaries where internal instrumentation is not possible.

Second, to satisfy G2, the method should be portable across tracing backends. Production environments differ in which tracepoints and tooling are available, so B-Perf assumes an abstract event model that requires timestamps, CPU identifiers, process and thread identifiers, event types, and a small set of type-specific attributes, such as allocation sizes or payload lengths. The current implementation uses LTTng on Linux, but the methodology applies to any backend that can provide equivalent signals.

Third, to satisfy G3, the framework should cover multiple behavioural scopes. Real performance problems rarely isolate themselves to a single resource. Thread contention, dynamic allocation, and messaging inefficiencies can interact in subtle ways. B-Perf reconstructs a common state system from the raw event stream and derives execution, memory, and messaging behaviour from this shared representation. This unified view removes the need for separate scope-specific tools.

Fourth, to satisfy G4, the analysis relies on workload trends rather than fixed signatures. Many antipatterns become visible mainly as load increases. B-Perf therefore evaluates an ordered workload sequence $\mathcal{L} = \{L_1, \ldots, L_k\}$ with a consistent request mix and examines how response times, blocking intervals, fragmentation signals, and messaging patterns change across L_1 to L_k. This allows B-Perf to report antipattern-consistent indicators even when implementations differ.

With these goals in place, B-Perf instantiates a high-level transformation:

$$E \;\rightarrow\; \left(B_{\text{exec}}, B_{\text{mem}}, B_{\text{msg}}\right) \;\rightarrow\; \textit{AntipatternInferences}, \qquad (1)$$

where E is the event stream collected under one or more workloads, B_{exec}, B_{mem}, and B_{msg} are the behaviour projections produced from the reconstructed state system, and the final step infers indicators that are consistent with known performance antipatterns.

Because system-level traces can be large, B-Perf aggregates trace-derived features to the workload level for scalable analysis [7].

Figure 1 summarizes the architecture of B-Perf. The *Workload Generation* module exercises the system under a sequence of increasing workloads with a fixed request mix. The *Event Gathering* module records kernel-level events during each execution and produces an event stream E that conforms to a generic schema with timestamps, CPU identifiers, thread and process identifiers, event types, and type-specific attributes. The *Trace Handler* module reconstructs a state-based abstraction from E, including per-request critical paths and resource usage intervals. The *SPA Detection* module then derives the behavior projections $\left(B_{\text{exec}}, B_{\text{mem}}, B_{\text{msg}}\right)$ and applies trend analysis across workloads to infer execution-, memory-, and messaging-scope antipattern indicators.

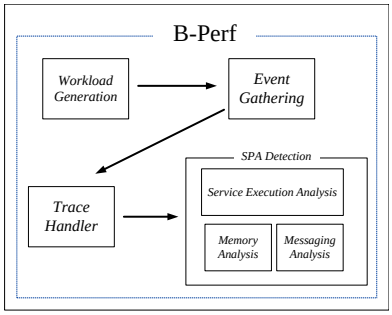


Figure 1: The architecture of B-Perf.

3.1 Trace Collection and Event Model

B-Perf operates on execution traces that record kernel-level interactions of the target software with the operating system. The framework assumes an abstract event model that can be instantiated by any tracing backend able to provide precise timestamps, CPU identifiers, thread and process identifiers, event types, and a small set of type-specific attributes. The current implementation uses LTTng on Linux to collect such traces, but the formalisation in this section is independent of a particular tracer.

Formally, an event e in the abstract model is a tuple $e = \langle t, c, k, a \rangle$ where t is a timestamp, c identifies the execution context (such as a process or thread), k denotes the event kind (for example schedule, block, syscall, alloc, send, or recv), and a contains event-specific attributes such as a CPU identifier, wait channel, memory address range, socket identifier, or payload size. The following subsections instantiate this model for concrete tracepoints in LTTng and compatible Linux tracing backends.

3.1.1 Event stream abstraction. Let $E = \langle e_1, e_2, \ldots, e_n \rangle$ denote the event stream collected during one execution under a given workload. The events are ordered by time, so that $\mathrm{ts}(e_i) \leq \mathrm{ts}(e_{i+1})$ for all i. Each event e_i carries a timestamp, a CPU identifier, a process/thread identifier, an event type, and a tuple of event-specific attributes. For instance, a scheduling event records the identifiers of the outgoing and incoming threads and the reason for descheduling, a memory event records an allocation size or address, and a network event records a socket identifier, payload size, and connection state.

In B-Perf, this event abstraction is instantiated using LTTng tracepoints that expose scheduling, synchronization, timer, interrupt, memory, and TCP activity in the Linux kernel. Examples include kernel scheduling events such as `sched_switch` and `sched_wakeup`, synchronization events such as `sys_enter_futex`, timer and interrupt events such as `hrtimer_expire`, `irq_handler`, and `softirq`, memory events such as `mm_page_alloc_extfrag`, and TCP events such as `tcp_sendmsg`, `tcp_receive`, `inet_sock_set_state`, and `tcp_retransmit_skb`. B-Perf requires only that the underlying tracer exposes events that can be mapped to this abstract schema. In practice, several Linux tracing facilities (including ftrace, perf, eBPF/BPFtrace, and SystemTap) can provide equivalent scheduling, syscall, memory, and TCP-level signals, making the approach compatible with a wide range of kernel tracing backends.

3.1.2 Workloads and request delimiters. B-Perf analyses behavior across a sequence of increasing workloads $\mathcal{L} = \{L_1, \ldots, L_k\}$ that share the same request mix. For each workload level L_i, the system under study is exercised for a fixed period or a fixed number of requests, and an event stream $E^{(i)}$ is collected. The union of these streams yields the set of events that B-Perf uses to derive behavior projections and workload-level summaries. The exact workload progression and stopping conditions are part of the experimental setup and are described in Section 4. The methodology only assumes that the workload levels are ordered by increasing intensity and that each level induces comparable request types.

Within each event stream $E^{(i)}$, B-Perf identifies request intervals using configurable delimiters. A request r is defined by a start event and a corresponding end event, denoted $\mathrm{start}(r)$ and $\mathrm{end}(r)$. The concrete delimiters depend on the target system. For example, for server workloads B-Perf can use connection-accept as a request start and connection close as a request end. When requests have different lifecycles, delimiters can be based on thread lifetimes or explicit task identifiers. The only requirement is that kernel events can be associated with request intervals.

3.1.3 State system reconstruction. The raw event stream does not directly expose the stable periods that are needed for behavior analysis. B-Perf therefore reconstructs a state system from E, in the style of the state system used in Trace Compass [9]. Let $S = \{s_1, s_2, \ldots, s_m\}$ denote the set of states. Each state s_j is a maximal time interval during which the state of each thread, its CPU residency, and its blocking relationships remain unchanged. State transitions occur at event boundaries where scheduling, synchronization, timer, interrupt, or communication events modify the execution context.

For each event e_i, B-Perf updates the state system by applying event-specific rules. A `sched_switch` event determines which thread leaves and which thread enters the `RUNNING` state. For example, a kernel event of the form `sched_switch`(t, cpu_1, t_1, t_2) indicates that at time t on CPU 1 the scheduler switches from thread t_1 to thread t_2. B-Perf updates the state system so that t_1 transitions from `RUNNING` to a waiting or blocked state (depending on the recorded reason), while t_2 moves to `RUNNING`. These states remain in effect until modified by subsequent events.

In the same way, `sched_wakeup` and `sys_enter_futex` events update the waiting state of a thread and the resource or dependency that blocks it. Timer and interrupt events adjust the set of active interrupts and their impact on preemption. Memory and network events update per-thread allocation and messaging metadata but do not change thread states directly.

Over time, the sequence of such updates yields a higher level abstraction of the trace in which long intervals where a thread remains blocked or waiting become explicit and can be linked to particular resources or dependencies. The reconstructed state system groups these low-level transitions into coherent intervals, making extended waiting periods on a given resource or thread visible. Such intervals are much easier to interpret as early signs of performance problems or antipatterns.

The state system induces per-request views of execution. For each request r, B-Perf extracts the subsequence of states whose intervals intersect the request interval $[\mathrm{start}(r), \mathrm{end}(r)]$ and derives

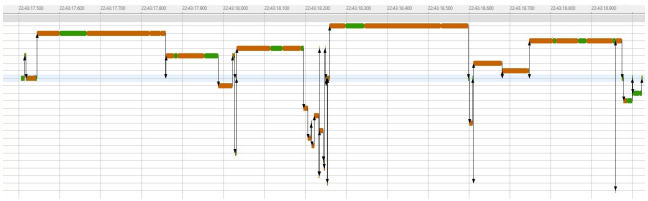


Figure 2: An example critical path showing `RUNNING`, `PREEMPTED`, and blocking relationships between threads.

a critical path and resource usage profile for that request. These per-request state sequences form the basis for the execution behavior projection B_{exec}, and they provide temporal anchors for associating memory and messaging events with specific requests.

3.2 Behavior Extraction

From the event stream and reconstructed state system, B-Perf derives three behavior projections for execution, memory, and messaging.

3.2.1 Execution behavior. Execution behavior captures how threads run, block, and compete for CPU resources while serving requests. Figure 2 shows an example critical path reconstructed by B-Perf. Green segments denote `RUNNING` intervals, orange segments denote `PREEMPTED` intervals, and arrows indicate where a thread is blocked by another.

B-Perf computes these paths using scheduling, synchronization, timer, and interrupt tracepoints, following the principles of the Trace Compass critical-path algorithm [1]. Events such as `sched_switch` identify which thread runs on each CPU, `sched_wakeup` and `sys_enter_futex` indicate synchronization waits, and `irq_handler`, `softirq`, and `hrtimer_expire` attribute preemption to interrupt activity. Together, these signals reveal the causal structure behind execution-scope antipatterns such as One Lane Bridge.

Intervals in which the handling thread is in the `RUNNING` state are treated as processing time. When the thread is not running, the state system identifies the blocking cause directly from the triggering event type. This mapping lets B-Perf attribute each segment of a request's critical path to a concrete source of delay.

For each request r, B-Perf computes

$$b_{\text{exec}}(r) = \big(R_r,\ T_r^{\text{run}},\ T_r^{\text{block}},\ \text{cpuResidency}_r,\ \text{contention}_r\big),$$

where R_r is response time, T_r^{run} is processing time, T_r^{block} is blocking time, cpuResidency_r describes how CPU time is distributed across cores, and contention_r measures blocking caused by other threads or shared resources. The execution behavior projection for a workload is

$$B_{\text{exec}} = \{\, b_{\text{exec}}(r) \mid r \text{ is a request} \,\}.$$

These per-request profiles form the basis for identifying execution-scope antipatterns such as serial bottlenecks and One Lane Bridge [19]. Combined with the memory and messaging projections, this execution view fulfils G3 by providing a consistent basis for cross-scope analysis using only kernel-level information.

3.2.2 Memory behavior. Memory behavior captures how dynamic allocation and fragmentation evolve while the system processes requests. B-Perf identifies memory-related events such as allocation and free calls, along with fragmentation signals exposed by kernel tracepoints. In the current implementation, fragmentation is observed through the `mm_page_alloc_extfrag` event, which reports the requested allocation order and the fallback order chosen by the page allocator.

For each request r, B-Perf associates allocation and free events with the request interval using the state system and thread identifiers. It then extracts features such as allocation count, total allocated bytes, and the distribution of allocation sizes. In parallel, B-Perf records global fragmentation indicators by tracking the frequency of `mm_page_alloc_extfrag` and comparing `alloc_order` with `fallback_order`. Persistent use of lower fallback orders indicates sustained fragmentation pressure.

The memory behavior projection for a workload is

$$B_{\text{mem}} = \big\{b_{\text{mem}}(r) \mid r \text{ is a request}\big\} \cup \big\{\text{fragStats}(t) \mid t \text{ in the trace}\big\},$$

where $b_{\text{mem}}(r)$ summarizes allocation intensity and size patterns for request r, and $\text{fragStats}(t)$ captures fragmentation activity over time. These features support detection of memory-scope antipatterns such as excessive dynamic allocation and allocator fragmentation.

3.2.3 Messaging behavior. Messaging behavior reflects how the system sends and receives data and how communication interacts with CPU scheduling. B-Perf reconstructs messaging activity from TCP-level tracepoints and communication-related system calls, including `tcp_sendmsg`, `tcp_receive`, `inet_sock_set_state`, and `tcp_retransmit_skb`. Each event includes a socket identifier, direction, and payload size. Using the state system and connection identifiers, B-Perf links these events to requests or to long-lived sessions when requests are not separable.

For each request or connection, B-Perf computes features such as message count, total bytes transferred, payload-size distribution, and the ratio of messages below a small-payload threshold. Network-induced preemptions are captured by counting timer and interrupt events co-occurring with network activity (`irq_handler`, `softirq`). Retransmissions (`tcp_retransmit_skb`) are recorded as indicators of congestion or packet loss.

The messaging behavior projection is

$$B_{\text{msg}} = \big\{b_{\text{msg}}(c) \mid c \text{ is a request or connection}\big\},$$

where $b_{\text{msg}}(c)$ summarizes message rate, payload sizes, small-message ratio, retransmissions, and related preemption signals. These features characterize communication patterns associated with messaging-scope antipatterns such as small-message overhead and interrupt-driven delays.

3.3 Antipattern Inference from Behavior Trends

Given the execution, memory, and messaging projections for each workload, B-Perf infers antipattern indicators by analysing how these behaviors evolve across the ordered workload sequence $\mathcal{L} = \{L_1, \ldots, L_k\}$. For each workload level L_i, B-Perf aggregates features from $B_{\text{exec}}^{(i)}$, $B_{\text{mem}}^{(i)}$, and $B_{\text{msg}}^{(i)}$ into a workload summary

$$\mathcal{B}(L_i) = \big(\overline{R}_i,\ \overline{T}_i^{\text{run}},\ \overline{T}_i^{\text{block}},\ \overline{\text{alloc}_i},\ \overline{\text{frag}_i},\ \overline{\text{msg}_i},\ \overline{\text{payload}_i}\big),$$

where the components capture response time, processing and blocking breakdowns, allocation and fragmentation intensity, and messaging rate and payload size. Additional derived quantities, such as blocking shares or the proportion of small messages, are incorporated when needed. Algorithm 1 summarizes how B-Perf combines these behavior summaries across workloads to infer antipattern indicators.

Algorithm 1: Antipattern Inference from Behavior Trends

Input: Ordered workloads $\mathcal{L}$ and summaries $\{\mathcal{B}(L_i)\}$
Output: Antipattern indicators $\mathcal{A}$
$\mathcal{A} \leftarrow \emptyset$;
Execution-scope inference
  Extract $\{\overline{R}_i\}$ and blocking decomposition;
  if *IsSuperLinearGrowth* **then**
    $RS^{\star} \leftarrow$ DominantBlockingResources;
    if $RS^{\star} \neq \emptyset$ **then**
      $\mathcal{A} \leftarrow \mathcal{A} \cup \{\text{Execution-serialization}(RS^{\star})\}$;
Memory-scope inference
  Extract $\{\overline{\text{alloc}_i}\}$ and $\{\overline{\text{frag}_i}\}$;
  if *HasRisingAllocationChurn* **then**
    $\mathcal{A} \leftarrow \mathcal{A} \cup \{\text{Allocation-churn}\}$;
  if *HasPersistentFragmentation* **then**
    $\mathcal{A} \leftarrow \mathcal{A} \cup \{\text{Fragmentation-pressure}\}$;
Messaging-scope inference
  Extract $\{\overline{\text{msg}_i}\}$ and $\{\overline{\text{payload}_i}\}$;
  if *MoreMessagesSmallerPayloads* **then**
    $\mathcal{A} \leftarrow \mathcal{A} \cup \{\text{Small-message-overhead}\}$;
  if *RisingNetworkPreemption* **then**
    $\mathcal{A} \leftarrow \mathcal{A} \cup \{\text{Messaging-preemption}\}$;
return $\mathcal{A}$;

As shown in Algorithm 1, execution-scope inference combines IsSuperLinearGrowth with DominantBlockingResources. The first condition checks whether response times increase faster than workload levels using a linear regression test on the pairs $\{(L_i, \overline{R}_i)\}$. B-Perf then identifies the earliest workload level L_y after which response time increments between consecutive workload levels remain consistently large relative to the fitted linear trend. When this condition holds, the range $\{L_y, \ldots, L_k\}$ is treated as nonlinear, and B-Perf attributes the growth to the dominant blocking resources identified by DominantBlockingResources.

Memory-scope inference examines whether allocation activity increases sharply (HasRisingAllocationChurn) and whether fragmentation events such as `mm_page_alloc_extfrag` persist across workloads (HasPersistentFragmentation). These indicators map to antipatterns involving excessive dynamic allocation and fragmentation pressure.

Messaging-scope inference evaluates whether higher workloads trigger more messages with smaller payloads (MoreMessagesSmallerPayloads) and whether this coincides with increased interrupt- or timer-related preemptions (RisingNetworkPreemption). Together, these conditions capture messaging inefficiency and communication-induced delays.

Across all scopes, B-Perf interprets the trends as behavioral indicators rather than precise classifications. The inference relies on how processing, blocking, allocation, fragmentation, and messaging evolve with load, and Section 4 shows how these indicators align with documented antipattern instances. As a whole, Algorithm 1 integrates these conditions into a unified procedure that reports the behavioral indicators inferred across the workload sequence.

4 EVALUATION

This section evaluates how effectively B-Perf detects representative performance antipatterns from system-level execution traces. We follow the three behavioural scopes of the methodology and examine how execution, memory, and messaging change as workload increases.

B-Perf is assessed on three controlled case studies and two existing multi-threaded applications. In the controlled setting, each antipattern is implemented together with a baseline that performs the same logical work without the problematic behaviour. This paired design makes it possible to compare behaviour under load and to isolate the effect of each antipattern, while the real-world traces provide an external validation of the methodology using traces that were not designed for this study.

We structure the evaluation around the following research questions.

RQ1 (Execution behaviour). To what extent can B-Perf detect execution-scope antipatterns, such as One Lane Bridge, by analysing critical paths, blocking intervals, CPU residency, and response-time trends under increasing load?

RQ2 (Memory behaviour). To what extent can B-Perf reveal memory-related antipatterns, such as Excessive Dynamic Allocation, through system-level allocation frequencies, allocation/free oscillation, and fragmentation indicators?

RQ3 (Messaging behaviour). To what extent can B-Perf identify messaging inefficiencies, such as Empty Semi Trucks, by analysing message frequency, payload-size distributions, and the impact of message bursts on execution behaviour?

RQ4 (Generalizability). To what extent does B-Perf provide useful behavioural insights when applied to a complex, multi-threaded real-world application using only system-level traces?

These questions test whether B-Perf can distinguish antipattern implementations from healthy baselines across the three behavioural scopes using the abstract event model and behaviour projections defined in Section 3. For each RQ, we instantiate the event model with LTTng traces, derive execution, memory, and messaging projections, and apply the trend-based inference predicates of Algorithm 1.

4.1 Experimental Setup

The evaluation uses test programs that exercise execution, memory, and messaging behaviour under increasing workloads. All experiments were executed on a Windows 10 host equipped with an Intel® Core™ i3 CPU @ 3.50 GHz and 8 GB of RAM. Traces were

recorded inside a guest system running Ubuntu 22.04 with 4096 MB of RAM and two virtual CPUs, deployed through Oracle VM VirtualBox 7.0.4. All test cases were implemented in C++ and compiled with GCC 10.3.0. LTTng 2.12.7 was used for event collection, and Trace Compass 8.2.0 was used for offline trace analysis.

The tracing session was configured in non-live mode so that events were written to disk with minimal interference to scheduling. The experiment enabled the kernel-level tracepoints required by B-Perf, including scheduling, synchronisation, memory, and TCP tracepoints. These include `sched_switch`, `sched_wakeup`, `sys_enter_futex`, `mm_page_alloc_extfrag`, `tcp_sendmsg`, and `irq_handler`. This configuration matches the abstract event model described in Section 3 and provides the data needed to compute the execution, memory, and messaging projections.

Test programs were implemented in `C++` and compiled with `gcc` using identical optimisation flags for the antipattern and baseline versions. Each program was executed under a sequence of increasing workloads that invoke the same operation while progressively raising the number of concurrent requests. The same workload levels were applied to both versions of each program, and each configuration was repeated to confirm that the observed trends were stable.

After collection, traces were processed by the B-Perf Trace Handler, which reconstructed execution intervals, resource states, and critical paths. The behaviour extraction algorithms then built the execution, memory, and messaging projections and aggregated them by workload level. The following subsections present the resulting behaviour trends and answer RQ1–RQ4.

4.2 RQ1: Detecting Execution-Scope Antipatterns (One Lane Bridge)

This part of the evaluation examines whether B-Perf can identify execution-scope antipatterns by observing how CPU usage, blocking behaviour, and critical paths evolve as the workload increases. The One Lane Bridge (OLB) scenario models a classical serialisation bottleneck: many threads compete for a shared section that allows only one active thread at a time. The baseline version replaces this design with multiple independent sections so that threads can progress concurrently.

Figure 3 shows the resource analysis for the two versions. In the baseline case, work is distributed across the available CPUs, and both cores exhibit sustained activity as the workload grows. In contrast, the OLB version concentrates almost all useful work on a single CPU, while the second core remains mostly idle except for brief scheduling noise. This behaviour is consistent with a shared critical section that serialises access regardless of the available parallelism.

The critical paths in Figure 4 make this serialisation explicit. In the OLB version, the critical path is formed by long intervals where a single thread is in the `RUNNING` state, while competing threads remain blocked. The baseline version shows frequent `PREEMPTED` segments and shorter waiting periods, indicating that multiple threads run in parallel and are preempted by the scheduler rather than blocked by a single lock.

Scheduling signals in Figure 5 further support this interpretation. As the workload increases, the OLB version exhibits repeated alternation between `WAIT` and `WAKE` events on the same shared resource, forming a queue of threads behind the critical section. The baseline version shows more dispersed wait and wake events across threads, and the duration of waits remains shorter.

These three views, CPU residency, critical paths, and wait/wake patterns, align with the behaviour of the execution inference algorithm in Section 3.3. Response times grow faster than the workload, and blocking time is dominated by a single serialising resource. In terms of the execution behaviour projection B_{exec}, this corresponds to a shift from processing time toward blocking time on one resource as the workload rises. B-Perf therefore raises an execution-serialisation indicator for the OLB version and no such indicator for the baseline.

Answer to RQ1. B-Perf detects execution-scope antipatterns in the OLB scenario. The reconstructed critical paths, blocking intervals, and CPU residency patterns are sufficient to distinguish serialised execution from a parallel baseline using system-level traces.

4.3 RQ2: Detecting Memory-Scope Antipatterns (Excessive Dynamic Allocation)

We now study whether B-Perf can detect memory-related antipatterns by observing allocation intensity, allocation–free cycles, and signs of fragmentation under increasing workloads. The Excessive Dynamic Allocation (EDA) scenario implements frequent creation and destruction of objects using `malloc` and `free`. The baseline version allocates memory once and reuses it for the duration of the run.

Figure 6 reports the memory usage over time. The EDA version shows a highly unstable pattern: memory usage repeatedly expands and contracts as the program allocates and frees objects in rapid succession. The baseline version maintains a stable footprint, because its memory is allocated once and reused. This difference appears even at low workloads and becomes more pronounced as request volume increases.

The underlying events are shown in Figure 7, which plots the number of `malloc` and `free` calls captured through LTTng. The EDA version produces a large and increasing number of allocation and deallocation events, whereas the baseline version produces only a small and nearly constant number. The detailed view in Figure 11 confirms that many distinct pointer addresses are allocated and freed, which increases fragmentation pressure on the allocator.

In terms of the behaviour projections, these traces correspond to a growing allocation frequency $\overline{\text{alloc}}_i$ and more variable allocation sizes as the workload rises. The presence of frequent allocation and free events on the same addresses and the instability of the memory footprint match the conditions of HasRisingAllocationChurn in the inference algorithm. In our experiments, B-Perf flagged allocation-churn indicators for the EDA version across higher workloads, while the baseline remained stable and did not trigger any memory-scope indicator.

Answer to RQ2. B-Perf identifies memory-scope antipattern behaviour in the EDA scenario. Allocation churn and oscillating memory usage are clearly visible in the system-level traces and absent

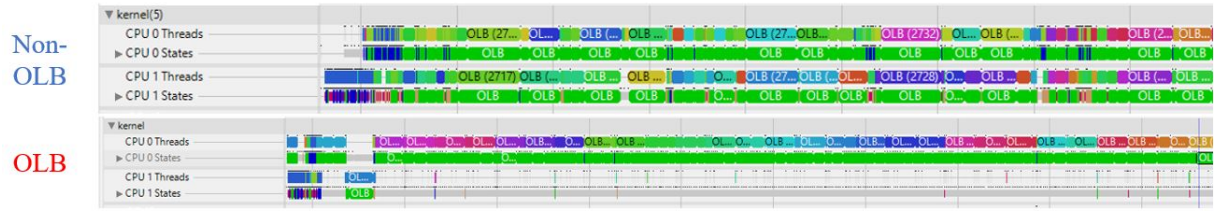


Figure 3: The *Resource Analysis* over OLB and Non-OLB versions of the application.

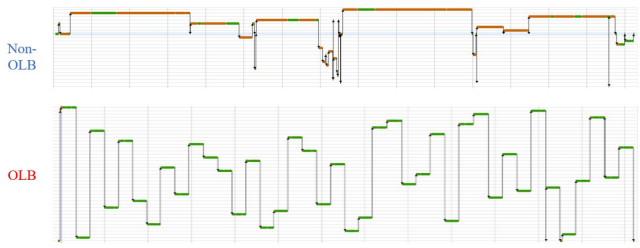


Figure 4: The critical path of the OLB and Non-OLB versions of the application.

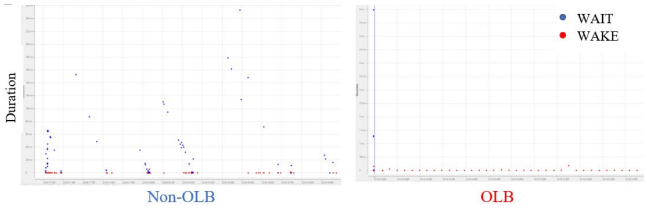


Figure 5: Number of `WAIT` and `WAKE` signals over time.

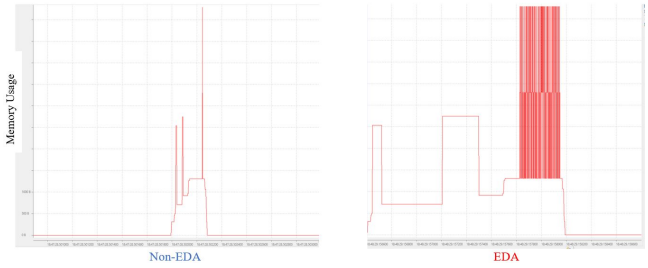


Figure 6: Memory usage over execution time for EDA and baseline versions.

from the baseline, showing that the memory behaviour projections highlight excessive dynamic allocation.

4.4 RQ3: Detecting Messaging-Scope Antipatterns (Empty Semi Trucks)

We now evaluate whether B-Perf can identify messaging-related antipatterns through kernel-level network activity. The Empty Semi Trucks (EST) antipattern occurs when an application sends many very small messages instead of aggregating data into larger payloads. This behaviour increases protocol overhead, network interrupts, and preemption. The experiment uses a simple FTP-like client that transfers a 1 GB file to a server. In the EST version, the client flushes after every 100 B of data, while the baseline version sends buffered segments.

Figure 7: Number of `malloc` and `free` events over time.

Figure 8 presents the resource analysis for the two versions. The EST case generates many more network-related interrupts and TCP events than the baseline. Each small flush forces the kernel to handle a new packet, schedule send operations, and trigger device interrupts. The baseline version exhibits a smoother profile with fewer network events, consistent with larger aggregated transfers.

The critical paths in Figure 9 show the impact on execution. The EST version contains a dense pattern of preemption intervals caused by network interrupts, and the application is frequently pushed off the CPU to service short send operations. In contrast, the baseline version maintains longer `RUNNING` intervals with fewer interrupts.

Figure 10 illustrates the network events themselves. In the EST version, the payload size of `tcp_sendmsg` events is small and nearly constant, matching the 100 B flush policy. The baseline version sends larger data chunks per message. Together, these observations correspond to a messaging sequence $\{\overline{\text{msg}}_i\}$ that rises sharply with workload and a payload sequence $\{\overline{\text{payload}}_i\}$ that remains small for the EST version. In the terminology of Algorithm 1, the predicates MoreMessagesSmallerPayloads and RisingNetworkPreemption are satisfied, and B-Perf triggers both small-message-overhead and messaging-preemption indicators in the EST case.

Answer to RQ3. B-Perf detects messaging inefficiency in the EST scenario. High message frequency, very small payloads, and increased preemption are visible in the system-level traces and absent from the buffered baseline, showing that messaging-scope antipatterns can be diagnosed without protocol-specific instrumentation.

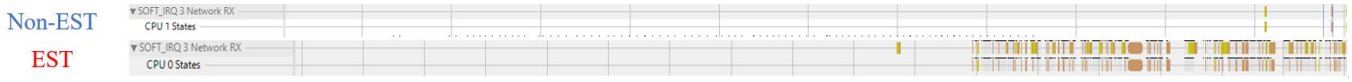


Figure 8: The *Resource Analysis* of EST and Non-EST executions.

Figure 9: The critical path of EST and Non-EST executions.

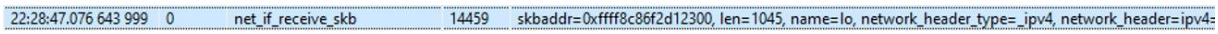


Figure 10: Network events in EST execution, showing the small size of the messages.

4.5 RQ4: External Validation on a Real-World Multi-Threaded Application

Finally, we applied B-Perf to existing traces from two multi-threaded applications. The first, Mozilla Firefox, is a widely used open-source browser with a complex rendering pipeline, substantial I/O, scripting, layout, and network interaction. We analysed a document-processing workload that is known to exhibit long response times in certain configurations [6]. The trace was collected with LTTng at system level during repeated document-rendering runs under increasing concurrency.

We considered a scenario in which one document instance is rendered in a single window and then repeated in two and three parallel windows. The average render time increased from roughly 30 s for one window to about 2.2 min for two windows and 5.7 min for three windows, a clearly super-linear trend also confirmed by statistical tests in the original analysis of this trace. When we feed the same trace into B-Perf, the execution behaviour projection reports a strong increase in blocking time on a small set of resources, while CPU residency becomes skewed toward waiting states as concurrency grows. The trend-based inference of Section 3.3 flags an execution-serialisation indicator that matches the qualitative behaviour of a One Lane Bridge on an active resource. Messaging and memory projections remain comparatively stable: message rates and payload sizes do not exhibit the high-frequency small-message pattern of Empty Semi Trucks, and allocation and fragmentation signals do not show the oscillation characteristic of Excessive Dynamic Allocation.

We also applied B-Perf to several multi-threaded, open-source web applications deployed on standard web stacks. Each system was exercised with workloads consisting of concurrent login or request-handling operations. Across all applications, the behaviour was consistent: at low concurrency levels the critical paths of request-handling threads were dominated by `RUNNING` intervals with short waiting phases, indicating balanced CPU usage. As concurrency increased to a few dozen parallel requests, the critical paths shifted toward long waiting intervals punctuated by very short bursts of processing. The execution projection showed that the majority of delay was due to threads waiting for CPU access, and the inference step consistently reported execution-serialisation indicators aligned with a One Lane Bridge on CPU. In all applications, the memory and messaging projections remained stable and did not trigger any memory- or messaging-scope indicators.

Answer to RQ4. Applying B-Perf to existing real-world applications shows that the same trace-driven methodology can highlight execution bottlenecks in complex traces and distinguish them from memory and messaging effects. Within the limits of these two case studies, this supports the generalizability of B-Perf beyond synthetic benchmarks, while broader evaluation on larger industrial systems remains future work.

4.6 Overhead and Efficiency of B-Perf

Because B-Perf operates at the system level and does not modify the target application, its runtime impact is dominated by the tracing subsystem and offline analysis cost. Across all experiments, LTTng produced traces ranging from a few hundred megabytes to several gigabytes, and the tracer remained stable without dropped events or visible disturbance of the application.

To quantify tracing overhead more explicitly, we conducted additional microbenchmarks using a standard CPU- and I/O-intensive workload generator. For a CPU-bound benchmark computing prime numbers up to a fixed limit, enabling the LTTng tracepoints used by B-Perf increased the total execution time by less than 0.01% compared to a run without tracing. For an I/O-bound benchmark performing random reads and writes on a large data file, enabling the same tracepoints reduced read and write throughput by about 7–8%. This represents a worst case with many I/O-related events and remains acceptable for offline performance diagnosis, especially when tracing can be restricted to specific phases or components.

The analysis pipeline processes each trace in a single pass. Event filtering, state reconstruction, and metric extraction visit each event once, so their cost grows linearly with trace size. Critical-path construction, the most computationally intensive step, remained tractable for all workloads and did not exhaust main memory. In our prototype, reading and reconstructing traces with millions of events completes in a few seconds. On the microbenchmarks used for overhead measurements, CPU-intensive traces of a few megabytes took around 2–3 s to read and 5–6 s to construct the critical path, while I/O-intensive traces of a few dozen megabytes required roughly 8–17 s. These costs are incurred offline and do not affect the timing of the system under test. The experiments therefore

indicate that B-Perf can handle realistic trace sizes without imposing additional overhead on the monitored program or complex tuning of the tracing configuration.

ust/uid/0/eda-user	18:48:29.158 135 045	u_0	0	lttng_ust_libc:malloc	size=4000, ptr=0x563899a52100, context.packet_seq_num=0, context.cpu_id=0
ust/uid/0/eda-user	18:48:29.158 135 185	u_0	0	lttng_ust_libc:free	ptr=0x563899a52100, context.packet_seq_num=0, context.cpu_id=0
ust/uid/0/eda-user	18:48:29.158 135 360	u_0	0	lttng_ust_libc:malloc	size=4000, ptr=0x563899a52100, context.packet_seq_num=0, context.cpu_id=0
ust/uid/0/eda-user	18:48:29.158 135 499	u_0	0	lttng_ust_libc:free	ptr=0x563899a52100, context.packet_seq_num=0, context.cpu_id=0
ust/uid/0/eda-user	18:48:29.158 135 676	u_0	0	lttng_ust_libc:malloc	size=4000, ptr=0x563899a52100, context.packet_seq_num=0, context.cpu_id=0
ust/uid/0/eda-user	18:48:29.158 135 812	u_0	0	lttng_ust_libc:free	ptr=0x563899a52100, context.packet_seq_num=0, context.cpu_id=0

Figure 11: The memory-related events show how memory is allocated and freed in the EDA version.

5 DISCUSSION

The evaluation across RQ1–RQ4 addresses the central question posed in Section 1: system-level execution traces provide sufficient behavioural information for diagnosing performance antipatterns in a black-box setting. B-Perf can distinguish antipattern implementations from their baselines in all three behavioural scopes using system-level traces alone and, in the external case studies, provides consistent execution-scope indicators on complex real-world traces. In the execution case, serialised critical paths, asymmetric CPU residency, and longer blocking intervals appeared only in the One Lane Bridge implementation. In the memory case, allocation churn, unstable memory usage, and fragmentation signals were specific to the Excessive Dynamic Allocation version. In the messaging case, dense interrupt activity, very small payload sizes, and frequent preemption characterised the Empty Semi Trucks scenario but not the buffered baseline. These results support the main claim that event-derived behaviour signals are sufficient to expose characteristic performance antipatterns without source code or intrusive instrumentation.

A central strength of B-Perf is its behavioural focus. Instead of matching static signatures or assuming a particular software architecture, B-Perf reasons about how behaviour evolves under increasing workload and how latency, allocation pressure, and messaging activity shift across workloads. This trend-based view makes the method less sensitive to implementation details and programming language, as long as the operating system exposes a suitable set of tracepoints. By working at the level of execution, memory, and messaging projections, the approach can be applied to diverse systems, from microbenchmarks to larger services, without redesigning the analysis for each target.

The state system reconstruction provides a common substrate for different classes of antipatterns. Execution issues appear as blocking and serialised CPU usage, memory issues appear as allocation churn and fragmentation signals, and messaging issues appear as interrupt storms and small fragmented payloads. Despite their different causes, these effects can be expressed in a uniform way through the projections of Section 3 and the inference predicates of Section 3.3. This shared representation is a key reason why B-Perf can cover multiple antipattern scopes within the same framework and can be extended to additional antipatterns that have clear behavioural signatures.

The approach has limitations. Its accuracy depends on the richness and granularity of the tracepoints available on the underlying platform. LTTng provides detailed kernel events on Linux, but systems with fewer or more coarse-grained signals may limit the reconstruction of state systems or reduce the ability to attribute delays to specific causes. The test programs in the evaluation isolate each antipattern with clean baselines, while real applications often mix multiple behaviours, introduce background noise, or involve layered service interactions. In such cases, the behavioural indicators may become harder to interpret, and some projections may require smoothing or additional filtering to separate dominant patterns from incidental effects. RQ4 shows that B-Perf generalises to multi-threaded web applications and a complex Firefox trace, but scaling to larger, highly concurrent industrial systems will require further validation.

A second limitation is the scope of kernel-level visibility. Kernel traces capture an application's interactions with CPUs, memory, and the network, giving strong insight into resource contention, scheduling delays, and fragmentation. They do not reveal the internal logic of the software itself. When performance problems arise from fine-grained function interactions, inefficient algorithms, tight in-process loops, or application-level lock contention, the kernel may show only indirect symptoms. B-Perf is therefore best suited for antipatterns driven by resource interaction and competition, and less effective when the root cause is purely internal to the application. Complementing system-level tracing with targeted userspace instrumentation is one way to address this limitation.

A third limitation is that B-Perf currently operates in a post-mortem setting. Traces are collected offline and processed after execution ends. Integrating B-Perf into continuous performance monitoring would require online filtering, incremental state reconstruction, and lightweight reporting. This opens opportunities for low-overhead live diagnosis but also raises challenges around storage, throughput, and alert precision when traces are long running and high volume.

The results show that system-level tracing enables black-box performance diagnosis grounded in observable behaviour rather than manual instrumentation or application-specific models. B-Perf provides a structured way to interpret execution, memory, and messaging trends across workloads, and this can be directly useful in practice. The method has been integrated into Trace Compass, a widely used open-source trace analysis environment. Developers often face many detailed views, such as CPU timelines, critical paths, and resource-specific charts, without a clear starting point for investigation. B-Perf offers an initial guidance layer that highlights the dominant behavioural signals and points analysts toward the resources and threads most likely linked to performance degradation. It does not replace deeper inspection, but it can serve as a practical foundation for more comprehensive observability workflows that combine trace-driven analysis with higher-level context such as service topology, configuration data, or developer annotations.

6 THREATS TO VALIDITY

Internal validity. The experiments use controlled programs where serialisation, allocation churn, or small-message behaviour is deliberately isolated. Real systems often mix several effects, which can make the root cause less clear. To reduce this risk, each antipattern was paired with a matching baseline, and workloads were repeated to confirm stable trends. LTTng has low overhead in the

configurations used here, and we did not observe dropped events or noticeable timing distortions.

External validity. The study focuses on Linux and relies on tracepoints available in LTTng and the upstream kernel. Other platforms may expose fewer or coarser-grained events. The microbenchmarks isolate one behavioural dimension at a time, which does not reflect the complexity of large multi-threaded applications with I/O, rendering, or service-level interactions. Although RQ4 shows that B-Perf can analyse complex multi-threaded traces, including a Firefox rendering workload and several web applications, broader validation on diverse, heterogeneous systems is needed.

Construct validity. B-Perf infers indicators from scheduling activity, allocation events, fragmentation signals, network payloads, and interrupts. These signals correspond to the antipatterns studied, but similar trends can arise from configuration changes or background interference. Examining trends across workloads and cross-checking execution, memory, and messaging projections mitigates this risk, but ambiguous cases remain possible when multiple effects overlap or when trace coverage is partial.

Conclusion validity. The conclusions rely on observed behavioural trends across workloads rather than strict statistical testing. The goal is to show that system-level signals allow antipattern implementations to be distinguished from their baselines. Additional experiments, parameter sweeps, and longer traces would further strengthen confidence in the robustness of the indicators, especially for memory- and messaging-intensive software running on busy systems, and would support more formal statistical analysis of the inferred trends.

7 CONCLUSION AND FUTURE WORK

This paper introduced B-Perf, a black-box method for diagnosing software performance antipatterns from system-level execution traces. The approach reconstructs execution, memory, and messaging behaviour from kernel-level events, which allows analysis without source code access or intrusive instrumentation. By tracking how behaviour changes with increasing workload, B-Perf identifies characteristic signals of execution serialisation, allocation churn, and messaging overhead and summarises them as behavioural indicators.

The evaluation covered three representative antipattern classes and included traces from real multi-threaded applications. Across all cases, B-Perf surfaced the underlying issues by analysing reconstructed behavioural patterns, which shows that system-level traces contain sufficient information to diagnose diverse performance problems. The method operated with low tracing overhead and processed large trace volumes, an important requirement for realistic workloads and long-running experiments.

There are several directions for extending this work. Applying B-Perf to larger multi-component and industrial systems would test its robustness under more complex interactions and mixed workloads. Supporting online or near real-time analysis could enable integration into continuous performance monitoring and DevOps pipelines, provided that incremental state reconstruction and lightweight reporting remain practical. Evaluating the approach on other platforms and tracing frameworks and refining the behavioural indicators for scenarios with overlapping effects will further assess its portability and strengthen its role in continuous performance engineering.

REFERENCES

[1] [n. d.]. Trace Compass. https://eclipse.dev/tracecompass/. Accessed: 2025-02-01.

[2] Marwen Abbes, Foutse Khomh, Yann-Gael Gueheneuc, and Giuliano Antoniol. 2011. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In *2011 15Th european conference on software maintenance and reengineering*. IEEE, 181–190.

[3] Mohammad Mejbah Ul Alam, Tongping Liu, Guangming Zeng, and Abdullah Muzahid. 2017. Syncperf: Categorizing, detecting, and diagnosing synchronization performance bugs. In *Proceedings of the Twelfth European Conference on Computer Systems*. 298–313.

[4] Alberto Avritzer, Andrea Janes, Catia Trubiani, Helena Rodrigues, Yuanfang Cai, Daniel Sadoc Menasché, and Álvaro José Abreu de Oliveira. 2025. Architecture and Performance Anti-patterns Correlation in Microservice Architectures. In *2025 IEEE 22nd International Conference on Software Architecture (ICSA)*. IEEE, 60–71.

[5] Mathieu Desnoyers and Michel R Dagenais. 2006. The lttng tracer: A low impact performance and behavior monitor for gnu/linux. In *OLS (Ottawa Linux Symposium)*, Vol. 2006. Citeseer, 209–224.

[6] Clemens Eisserer. 2019. *Firefox takes 5min to render poster PDF*. https://bugzilla.mozilla.org/show_bug.cgi?id=1508765

[7] Naser Ezzati-Jivan and Michel R Dagenais. 2015. Cube data model for multilevel statistics computation of live execution traces. *Concurrency and Computation: Practice and Experience* 27, 5 (2015), 1069–1091.

[8] Naser Ezzati-Jivan, Quentin Fournier, Michel R. Dagenais, and Abdelwahab Hamou-Lhadj. 2020. DepGraph: Localizing Performance Bottlenecks in Multi-Core Applications Using Waiting Dependency Graphs and Software Tracing. In *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 149–159. https://doi.org/10.1109/SCAM51674.2020.00022

[9] Francis Giraldeau, Julien Desfossez, David Goulet, Michel Dagenais, and Mathieu Desnoyers. 2011. Recovering system metrics from kernel trace. In *Linux Symposium*, Vol. 109. 26.

[10] Madeline Janecek, Naser Ezzati-Jivan, and Seyed Vahid Azhari. 2021. Container Workload Characterization Through Host System Tracing. In *2021 IEEE International Conference on Cloud Engineering (IC2E)*. 9–19. https://doi.org/10.1109/IC2E52221.2021.00015

[11] Madeline Janecek, Naser Ezzati-Jivan, and Abdelwahab Hamou-Lhadj. 2022. Performance anomaly detection through sequence alignment of system-level traces. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*. 264–274.

[12] Philipp Keck, André Van Hoorn, Dušan Okanović, Teerat Pitakrat, and Thomas F Düllmann. 2016. Antipattern-based problem injection for assessing performance and reliability evaluation techniques. In *2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 64–70.

[13] Octavio Loyola-González. 2019. Black-Box vs. White-Box: Understanding Their Advantages and Weaknesses From a Practical Point of View. *IEEE Access* 7 (2019), 154096–154113. https://doi.org/10.1109/ACCESS.2019.2949286

[14] Karthik Nagaraj, Charles Killian, and Jennifer Neville. 2012. Structured comparative analysis of systems logs to diagnose performance problems. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. 353–366.

[15] Manjula Peiris and James H Hill. 2016. Automatically detecting" excessive dynamic memory allocations" software performance anti-pattern. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*. 237–248.

[16] Derek Rayside and Lucy Mendel. 2007. Object ownership profiling: a technique for finding and fixing memory leaks. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*. 194–203.

[17] Mohammadreza Rezvani, Ali Jahanshahi, and Daniel Wong. 2024. Characterizing in-kernel observability of latency-sensitive request-level metrics with ebpf. In *2024 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 24–35.

[18] Junghwan Rhee, Hui Zhang, Nipun Arora, Guofei Jiang, and Kenji Yoshihira. 2014. Software system performance debugging with kernel events feature guidance. In *2014 IEEE Network Operations and Management Symposium (NOMS)*. IEEE, 1–5.

[19] Connie Smith and Lloyd Williams. 2001. Software Performance AntiPatterns; Common Performance Problems and their Solutions. Citeseer, 797–806.

[20] Connie U Smith and Lloyd G Williams. 2003. More new software performance antipatterns: Even more ways to shoot yourself in the foot. In *Computer Measurement Group Conference*. Citeseer, 717–725.

[21] Elder Vicente de Paulo Sobrinho, Andrea De Lucia, and Marcelo de Almeida Maia. 2021. A Systematic Literature Review on Bad Smells–5 W's: Which, When,

What, Who, Where. *IEEE Transactions on Software Engineering* 47, 1 (2021), 17–66. https://doi.org/10.1109/TSE.2018.2880977

[22] Catia Trubiani, Alexander Bran, André van Hoorn, Alberto Avritzer, and Holger Knoche. 2018. Exploiting load testing and profiling for Performance Antipattern Detection. *Information and Software Technology* 95 (2018), 329–345. https://doi.org/10.1016/j.infsof.2017.11.016

[23] Riley VanDonge and Naser Ezzati-Jivan. 2022. N-Lane Bridge Performance Antipattern Analysis Using System-Level Execution Tracing. In *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 83–93.

[24] Miguel Velez, Pooyan Jamshidi, Florian Sattler, Norbert Siegmund, Sven Apel, and Christian Kästner. 2020. ConfigCrusher: towards white-box performance analysis for configurable systems. *Automated Software Engineering* 27 (12 2020). https://doi.org/10.1007/s10515-020-00273-8

[25] Alexander Wert. 2018. *Performance problem diagnostics by systematic experimentation*. Vol. 20. KIT Scientific Publishing.

[26] Alexander Wert, Jens Happe, and Lucia Happe. 2013. Supporting swift reaction: Automatically uncovering performance problems by systematic experiments. In *2013 35th International Conference on Software Engineering (ICSE)*. 552–561. https://doi.org/10.1109/ICSE.2013.6606601

[27] Alexander Wert, Marius Oehler, Christoph Heger, and Roozbeh Farahbod. 2014. Automatic detection of performance anti-patterns in inter-component communications. In *Proceedings of the 10th international ACM Sigsoft conference on Quality of software architectures*. 3–12.

[28] Ruyue Xin, Jingye Wang, Peng Chen, and Zhiming Zhao. 2025. Trustworthy AI-based performance diagnosis systems for cloud applications: A review. *Comput. Surveys* 57, 5 (2025), 1–37.