DDB: Source-Level Interactive Debugging for Distributed Applications

Yibo Yan Junzhou He Seo Jin Park
University of Southern California

Abstract

Interactive debugging is an effective tool for understanding program behavior at the source level, allowing developers to pause execution, navigate the call stack, and inspect runtime state. However, interactive debuggers are designed for single-process execution, and interactive debugging has been widely considered impractical for distributed systems. Call stacks stop at process boundaries, debugging state fails to survive infrastructure dynamics, and, most critically, debugger-induced execution pauses trigger catastrophic timeout cascades that destroy the intended debug flow. Consequently, developers are forced to abandon live hypothesis testing in favor of unwieldy and iterative log-and-redeploy cycles.

We present DDB, a source-level interactive debugger that extends interactive debugging capabilities to distributed applications. We show that each of these challenges admits a targeted solution. To bridge disjoint processes, Distributed Backtrace (DBT) embeds compact causality metadata in every RPC and reconstructs a unified call stack across RPC boundaries. To manage the lifecycle of a distributed session, an intent-preserving control plane automatically coordinates and propagates breakpoints across dynamic process sets. To make pausing safe, Pause-Erased Time (PET) virtualizes each process's clock, decoupling logical time from physical pauses and preventing timeout cascades. DDB integrates with an RPC framework in 10–60 lines of code. Evaluated on gRPC, ServiceWeaver, Nu, and Quicksand across up to 122 processes, DDB achieves 30 ms median cross-RPC backtrace latency, sub-5 ms time jump under repeated execution pauses, and adds 1–5% throughput overhead, comparable to attaching a single-process debugger. In a controlled user study, DDB achieves a 100% fault localization success rate (compared to 38.5% for baseline tools) with a median localization time of ∼8 minutes.

CCS Concepts: • **Computing methodologies** → **Distributed computing methodologies**; **Parallel computing methodologies**; • **Software and its engineering** → **Software notations and tools**.

SOSP '26, September 29–October 2, 2026, Prague, Czech Republic

ACM ISBN 979-8-4007-2585-2/26/09
https://doi.org/10.1145/3830418.3843853

ACM Reference Format:
Yibo Yan, Junzhou He, and Seo Jin Park. 2026. DDB: Source-Level Interactive Debugging for Distributed Applications . In *Symposium on Operating Systems Principles (SOSP '26), September 29–October 2, 2026, Prague, Czech Republic.* ACM, New York, NY, USA, 22 pages. https://doi.org/10.1145/3830418.3843853

1 Introduction

Serverless platforms, microservice architectures, and modular programming frameworks increasingly decompose what was once a single-process application into code that executes across multiple services and machines [38, 42, 48, 73, 75]. The same trend spans actor frameworks [1, 24, 61], distributed application runtimes [18], and memory-disaggregated runtimes [72, 74, 80]. The result is that application developers—who may have little expertise in distributed systems—are now routinely writing code whose execution spans dozens of processes. As these paradigms and infrastructure have emerged, the expertise barriers to adopting distributed programming have dropped substantially, as depicted in Figure 1.

For debugging single-process programs, a developer attaches a debugger (*e.g.,* GDB [19], LLDB [10]), sets breakpoints, and inspects call stacks, variables, and caller state—a tight hypothesis–inspect–refine loop for understanding *why* code misbehaves. Once an application becomes distributed, this workflow collapses. A breakpoint in one process reveals nothing about the remote callers that triggered the current execution: the chain of upstream RPCs, the arguments they passed, and their runtime state remain invisible. Attaching separate debuggers to every upstream process and locating the correct thread among potentially hundreds [25] does not scale well. Moreover, pausing any process triggers leader re-elections, transaction aborts, or cluster-wide crash recovery, because distributed systems rely on failure-detection timeouts as short as 50–100 ms [87].

For traditional distributed infrastructure, such as large-scale storage systems [29, 41] and distributed databases [54], this absence of interactive debugging has been mitigated by heavy investment in alternative diagnostic tooling. The teams that build these systems at major organizations invest heavily in structured logging [87], distributed tracing [8, 12, 13], and custom monitoring infrastructure to diagnose faults in production. Application developers typically lack deep expertise in distributed systems; without interactive debugging, they resort to iterative log-and-redeploy cycles that studies show consume days to weeks—and in extreme

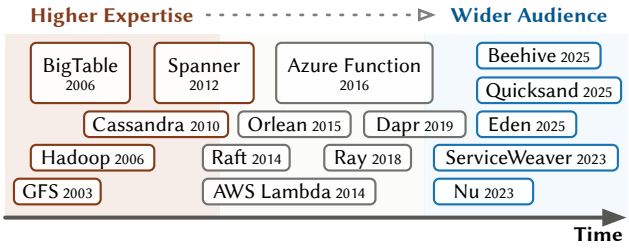

Figure 1. Distributed programming paradigms and infrastructure have become much more accessible to ordinary application developers. The adoption of distributed programming has increased while the required expertise has dropped substantially.

cases over a month—per fault [86, 89]. As quantified in our user study (§6.1), participants using baseline tools (GDB, distributed tracing) failed to localize cross-service faults in 61.5% of cases and often exceeded the 20-minute time limit.

These failures stem from a fundamental mismatch: each existing tool addresses a different aspect of distributed diagnosis, but none provides the interactive workflow that cross-RPC, source-level fault localization requires. Distributed tracing and logging reconstruct request paths and record event histories [8, 12, 13]; record-and-replay enables deterministic post-hoc analysis [15, 39, 79]. Interactive debuggers exist for parallel and HPC settings [9, 16, 20], but these assume each process can be paused independently without side effects—an assumption that breaks when applications use RPC timeouts for failure detection. None supports what application developers need: pausing a running distributed execution, navigating the cross-RPC call chain, and examining live runtime state across service boundaries.

We present DDB, a distributed interactive debugger that provides this workflow. With DDB, a developer can set a breakpoint once and it is automatically inserted across all relevant replicas, including processes that join later through scaling or restart. When the breakpoint fires, DDB defaults to a pause-the-world model (akin to GDB's default behavior), halting all attached processes to provide a consistent global view. Because all processes are concurrently paused, waiting for manual inspection is safe: DDB virtualizes each process's view of time so that application-level timeouts and timers are unaffected by the execution pause.

We observe that in development and test environments, where coordinated global pauses are feasible, cross-RPC call stacks can be reconstructed by embedding compact causality metadata in every RPC; dynamic process sets can be managed through an intent-preserving control plane; and timeout cascades can be eliminated by virtualizing each process's view of time, which requires addressing a temporal anomaly (static application state combined with advancing kernel time) that no existing clock-virtualization mechanism handles (§4.3). Because interactive debugging naturally targets staging and test environments rather than production, the overhead of these mechanisms is acceptable.

Like traditional interactive debuggers (*e.g.,* GDB [19], LLDB [10]), DDB uses the pause-the-world behavior by default, targeting semantic logic and state errors across RPC boundaries; concurrency bugs that depend on precise thread interleaving fall outside this scope, as they do for all pause-based debuggers. Under the same assumptions as traditional interactive debuggers, DDB provides the best debugging experience when debug symbols are embedded in the compiled binaries and the optimization level is tuned down.

This paper makes the following contributions:

- **Distributed Backtrace (DBT).** Cross-RPC stack reconstruction, transparent to user-level threads, that enables live inspection of caller state across RPC boundaries (§4.1).
- **Intent-Preserving Control Plane.** A debug-intent abstraction that propagates breakpoints and commands across replica scaling, node churn, and computation migrations (§4.2).
- **Pause-Erased Time (PET).** Time virtualization via *Virtual Deadline Enforcement* that decouples physical and logical time, enabling safe debugger pauses without timeout cascades (§4.3).
- **Implementation and Evaluation.** We integrate DDB with four RPC frameworks in two languages (10–60 LoC each) and evaluate at 122-process scale: 1–5% throughput overhead, ~30 ms backtrace latency, and <5 ms time jumps. A controlled user study shows DDB delivers 100% cross-service fault localization vs. an aggregated 38.5% localization rate with baseline tools (§5, §6).

2 Background and Motivation

2.1 The Developer Workflow and the Missing Capability

Consider a developer debugging a distributed key-value store where certain read requests return stale values after a write. The developer can reproduce the issue with a specific sequence of write and read operations issued from a test client. Distributed tracing identifies the request path: a client request reaches a router, which forwards it to one of several "storage-node" replicas. Structured logging on the storage node confirms that the node served the read from a cached entry and that a cache-invalidation RPC from the router did arrive, but the cached entry was not updated.

At this point the developer knows *where* the stale read occurs and *what* happened, but not *why* the invalidation failed to take effect. Understanding the root cause requires inspecting runtime state that no existing log captured. The root cause lies in cross-service runtime state: *what version identifier did the router embed in the invalidation RPC, what did the storage node's comparator evaluate, and did the invalidation target a different cache slot than intended?* In single-process development, GDB answers such questions directly: the developer sets a breakpoint, inspects

local variables, and walks the call stack. For a distributed application, no equivalent tool exists.

The developer falls back to iterative logging: instrumenting the invalidation handler, redeploying, and re-running the test, with each round revealing only the specific variables the developer chose to log. After several such cycles spanning two services, the developer discovers that the router was embedding a version field from a stale routing-table entry—a conclusion that a single interactive inspection of the cross-service call chain could have reached in minutes.

Monitoring, tracing, logging, and record-and-replay each address a distinct need, but none provides the capability this workflow demands: live, interactive inspection across service boundaries. The following subsections detail the entrypoints where interactive debugging begins (§2.2) and the specific challenges that prevent it from working across process boundaries (§2.3).

2.2 Entrypoints for Interactive Debugging

An interactive debugging session begins at an entrypoint, such as a manual breakpoint, a watchpoint, an assertion failure (`SIGABRT`), or a memory fault (`SIGSEGV`). In single-process development, these entrypoints cleanly pause execution for live inspection. In a distributed setting, catching the event is insufficient because the root cause often resides in an upstream remote caller or manifests across dynamic, auto-scaling replicas. Regardless of the specific trigger, every entrypoint shares a core requirement: once execution pauses, the developer must reconstruct the full cross-process execution context—including the chain of remote callers, their arguments, and their runtime state—to reason about the application. The next section examines the specific challenges that make this reconstruction difficult.

2.3 Why Interactive Debugging Breaks Across Process Boundaries

Distributed execution breaks interactive debugging in three ways (①–③ in Figure 2): call stacks end at process boundaries, debugging state does not survive churn, and temporary execution pauses trigger timeout-based failure detection.

Call stacks stop at the process boundary (Figure 2 ①). Consider a request that traverses services A → B → C. The developer suspects C's handler and sets a breakpoint. When the breakpoint fires, GDB reveals C's local stack but nothing about B or A—neither the arguments B passed nor the execution path that led B to issue the call. Recovering this context requires attaching a separate debugger to B, identifying the correct thread among potentially hundreds, inspecting B's stack, and repeating for A. The manual effort scales linearly with call depth. User-level threading compounds the difficulty: if B uses goroutines, the goroutine that sent the RPC

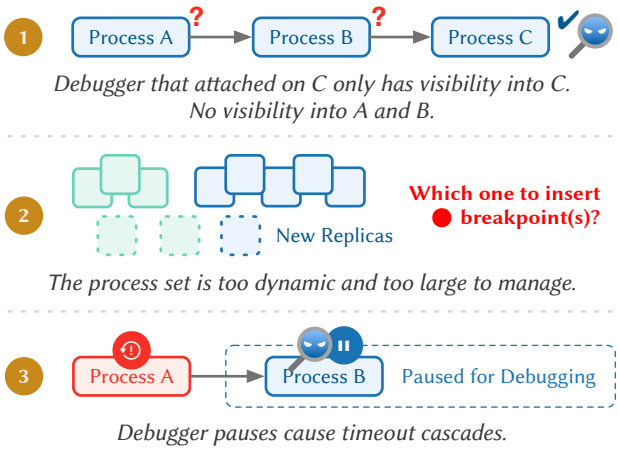

Figure 2. Three challenges of employing interactive debugging in distributed applications.

to C may have been descheduled and its OS thread reused, so GDB shows the wrong stack entirely.

Debugging operations require per-process manual effort (Figure 2 ②). Consider the 180-process deployment of the `socialnet` benchmark [37]. To debug an issue, a developer must manually identify relevant replicas and insert breakpoints on each individually, which is a cumbersome and tedious process. Furthermore, the process set changes constantly: auto-scaling adds replicas, rolling restarts replace them, and emerging distributed programming frameworks [42, 71–73] introduce runtime computation migrations. Unless breakpoints automatically follow these state changes and computation migrations, the developer silently loses debugging coverage as execution moves across processes.

Execution pauses cause timeout cascades (Figure 2 ③). Distributed applications use timeouts extensively to detect and handle failures. We surveyed timeout thresholds in three open-source distributed systems: LogCabin [63] (a Raft implementation), RAMCloud [67] (in-memory distributed storage), and Nu [73] (a distributed programming framework). Thresholds range from 100 ms (RPC retry) to 500 ms (election timer). A developer pausing at a breakpoint for even a few seconds triggers every timeout in this range. The consequences range from harmless (extra heartbeats) to severe (aborted transactions, cluster-wide crash recovery for a healthy node). The disruption breaks the intended debug flow. For example, when an execution pause spuriously triggers crash recovery, the system transitions to a recovery state that did not exist before the pause, and the developer ends up inspecting recovery logic instead of the intended code path.

3 Overview: A DDB Debugging Session

To illustrate how DDB addresses the challenges outlined in §2, Figure 3 shows a live debugging session on a 3-node Raft cluster. In Raft, a single *leader* node coordinates the cluster

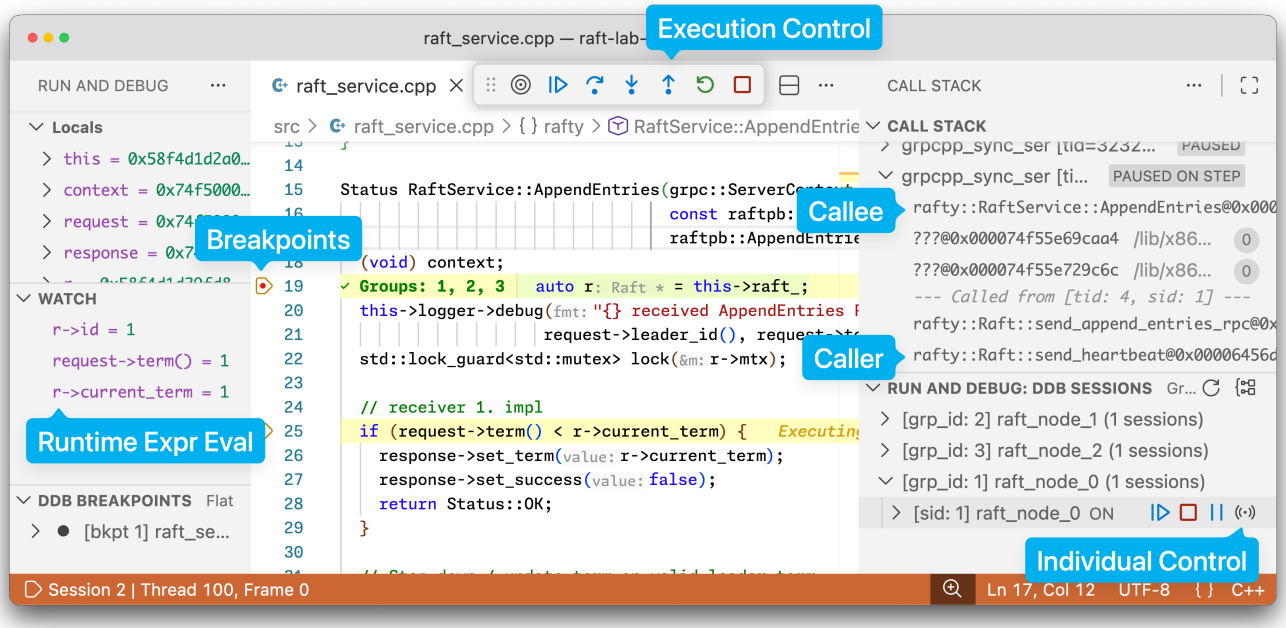

Figure 3. The DDB VS Code graphical frontend debugging a three-node Raft cluster (with one leader and two followers in the screenshot). Both followers have paused at the local `AppendEntries` RPC handler (middle). The Call Stack (top-right) displays a Distributed Backtrace extending to the leader's remote calling frame. The DDB Sessions pane (bottom-right) and debugger control pane (top) allow granular per-process execution control while the rest of the cluster remains globally paused.

by sending periodic heartbeat RPCs (`AppendEntries`) to each *follower* replica; if a follower stops receiving heartbeats, it initiates a leader election. The developer is investigating a bug where followers are improperly rejecting the leader's heartbeats. Instead of adding iterative logging statements and restarting the cluster, the developer attaches the DDB VSCode extension and resolves the issue in a single interactive session.

Setting a breakpoint across replicas. The developer sets a single breakpoint at the `AppendEntries` RPC handler. DDB's intent-preserving control plane (§4.2) automatically applies this breakpoint to all follower replicas in the cluster, without requiring the developer to identify or attach to each process individually.

Pausing the cluster safely. When the leader broadcasts its next heartbeat, both followers hit the breakpoint concurrently and DDB defaults to a pause-the-world model, halting the entire cluster. Without DDB, this pause would be fatal: Raft's election timeouts (typically 100–500 ms) would expire immediately, triggering a spurious leader election that destroys the state being debugged. DDB's Pause-Erased Time (PET) (§4.3) virtualizes each process's clock, keeping the cluster's failure detectors entirely unaware of the execution pause.

Inspecting the cross-RPC call chain. With the cluster safely paused, the developer opens the Call Stack pane. Instead of the local stack frames that a single-process debugger would show, DDB presents a Distributed Backtrace (DBT) (§4.1) that extends from the follower's handler back to the leader's `send_heartbeat` caller frame. The developer clicks on the upstream leader frame to inspect the exact arguments sent across the network. Local variables and expressions are automatically re-evaluated in the context of the selected frame.

Granular per-process control. While the cluster remains globally paused, the developer selects one follower and single-steps it forward (identifiable as "PAUSED ON STEP" in the interface), while the other follower stays suspended at the entrypoint. The developer then evaluates expressions in the Watch pane—comparing the incoming `request->term()` against the local `r->current_term`—and observes how divergent state across replicas affects execution flow, arriving at the root cause. This coordination is managed by DDB's intent-preserving control plane (§4.2).

This workflow relies on three core technical pillars—Distributed Backtrace, an intent-preserving control plane, and Pause-Erased Time (Figure 4), as described in the following section.

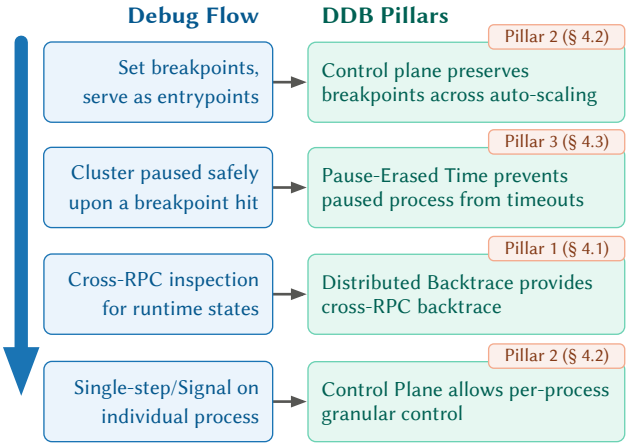

Figure 4. Overview of DDB's three pillars. Distributed Backtrace (DBT) reconstructs cross-RPC call stacks for live inspection. The intent-preserving control plane automatically propagates debug operations across dynamic process sets and enables granular per-process control. Pause-Erased Time (PET) virtualizes each process's clock to prevent debugger pauses from triggering timeout cascades.

4 DDB Debugging Model and Design

DDB is organized around three pillars: Distributed Backtrace (DBT), an intent-preserving unified control plane, and Pause-Erased Time (PET). For each pillar, we describe the developer-facing abstraction, the mechanism that realizes it, and the guarantees it provides.

4.1 Distributed Backtrace (DBT)

Abstraction. When execution pauses at any entrypoint, the developer issues `dbt` command to start a distributed backtrace. DDB returns a single, unified call stack that begins at the current frame in the paused process and extends backward across each RPC boundary to the root of the request path. For a request that traversed A → B → C, pausing in C and issuing `dbt` presents frames from C, then B, then A, in standard GDB backtrace format. The developer can navigate to any frame (including frames in remote processes) to inspect local variables, read heap-allocated objects, and modify values, much as in single-process debugging. We use this three-service request path as a running example throughout the rest of this section.

Mechanism. Realizing DBT requires solving two problems: locating the correct caller context in a remote process, and presenting a coherent unified call stack.

Caller-context capture design. When process A sends an RPC to B, the distributed backtrace must record enough context at the call site so that the caller's stack can later be reconstructed to form a coherent call stack along the entire causal chain (the exact recursive algorithm is detailed in Appendix §B.1). A strawman design records the caller's OS thread ID at the time of sending the RPC and, at reconstruction time, retrieves the thread using that ID to perform stack unwinding. This fails when the application uses user-level threads (uthreads), such as Go goroutines, CILK workers [26], Folly fibers [59], Shenango [66], and Caladan [34], which multiplex over a smaller pool of OS threads. Between the RPC send and the callee's pause, the runtime may reschedule the calling uthread off its original OS thread. Therefore, unwinding the stack on the caller thread identified by the OS thread ID produces the wrong call frames.

DDB instead captures the *thread context* (a set of register values that identify the stack) at the call site and embeds it directly in the RPC payload as compact caller-context metadata. Because the metadata carries the *thread context* rather than a thread identity, it remains valid regardless of thread scheduling decisions made by the runtime. This embedding is the only change to the RPC protocol and requires approximately 20 LoC per framework. At the callee process, the RPC handler extracts this metadata and stores it in a lightweight injected extraction frame (`DDB::Backtrace::extraction`, as shown in Figure 5), making it available on the stack for DDB to read at reconstruction time.

Cross-process stack reconstruction. When `dbt` is issued at process C, DDB locates the injected extraction frame in C's stack and reads the caller-context metadata, which identifies B by the embedded IP address and PID. Crucially, the reconstruction of B's stack does not happen inside C's address space. Instead, DDB's centralized control plane reads the metadata, contacts the debugger agent attached to process B out-of-band, and fetches the necessary stack memory segments. The agent then temporarily restores B's thread context and performs standard DWARF stack unwinding using B's debug symbols. DDB repeats this procedure for B's caller, iterating until no further caller-context metadata exists. It then concatenates all stack frames into a single stack and presents it to the developer. The entire procedure is transparent and the developer sees one contiguous call stack.

Concurrent request handling. Distributed backtrace correctly reconstructs call stacks for concurrent requests as long as the corresponding caller contexts remain valid. For example, suppose a client uses three threads to issue three concurrent requests to a server. If each sending thread remains blocked on its RPC (*i.e.,* synchronous blocking RPCs), DDB can independently perform `dbt` for any of the three request chains by placing a breakpoint in the server-side handler logic. Because each RPC carries the caller's thread context, DDB can uniquely identify and reconstruct the corresponding caller stack from each server-side handler thread. Consequently, concurrent requests remain isolated during backtrace reconstruction rather than being conflated with one another. However, distributed backtrace does not currently support event-driven concurrency, where the caller stack may be unwound or reused immediately after the request is issued. We discuss this limitation in more detail below.

Guarantees. DBT provides three guarantees. *Completeness:* the cross-RPC caller chain is reconstructed to the root or

```
0  RpcFlow::Parse
1  |   RpcMethod::RunHandler
2  |   |   DDB::Backtrace::extraction  // injected
2  |   |   |   meta = extractor();     // local var
3  |   |   |   RPCServer::user_handler
4  |   |   |   |   // User defined RPC handlers
```

Figure 5. Illustration of the frame injection. Blue numbers on the left indicate the frame number, where 0 is the outermost frame and 4 is the innermost frame. Frame #2 is the injected frame. The local variable `meta` is inside the `extraction` frame.

to the earliest reachable caller, correctly recovering each caller's stack regardless of user-level thread rescheduling. *Inspectability:* every reconstructed frame exposes locals, arguments, and heap-accessible objects for reading and modification. *Order preservation:* frames appear in reverse of request-path order (from callee to ancestor callers).

Edge cases. Under replication, DBT reconstructs the chain for the specific request that reached the paused process. If a caller has terminated before the callee pauses, DBT reports a truncated backtrace with a "caller terminated" annotation. If an upstream framework lacks DDB integration, DBT stops at that boundary and reports the truncation reason.

Limitations. Distributed backtrace performs stack unwinding under the assumption that the relevant caller stacks remain available. It therefore requires the caller's stack frames to remain valid when the backtrace is collected and currently supports only synchronous, blocking RPCs; in particular, it is incompatible with event-driven, asynchronous, or stackless concurrency models that do not preserve a physical caller stack across an RPC boundary. When DDB encounters such an asynchronous RPC boundary, `dbt` returns a truncated call stack because the current design does not propagate caller thread context across that asynchronous RPC boundary. Accordingly, reconstructing a complete distributed call chain requires every RPC along the chain to preserve a valid ancestor caller context, as is the case for synchronous, blocking RPCs.

This is a longstanding challenge for interactive debuggers rather than a limitation specific to DDB's design: any debugger that offers conventional backtrace capability, where the developer can switch to an arbitrary ancestor frame and inspect its values, must assume that caller stacks remain valid. To capture causality across asynchronous requests, prior work either falls back to tracing and logging, giving up interactive inspection [56, 69, 82], or offers only a causal view of ancestor callers by function name, without inspectable on-stack values [78].

4.2 Intent-Preserving Unified Control Plane

Abstraction. We define a *debug-intent* as a triple: a source-code *location* (file and line number), a *scope* identifying which processes to target ("all replicas of service X" or "a specific process"), and a debugger *command* ("set breakpoint", "continue", "evaluate expression", "send signal", *etc.*). The control plane resolves each intent against the current application topology and propagates it to every matching process. This unified control plane lowers the cognitive overhead of operating individual debuggers on distributed applications. Crucially, to provide a consistent view of distributed state, the control plane enforces a *pause-the-world* policy by default: whenever any attached process pauses (due to a breakpoint hit, single-step operation, or manual interrupt), the control plane automatically broadcasts a pause command to all other processes attached to the debugging session.

For example, if a developer sets a breakpoint targeting "all replicas of service B", DDB will preserve the debug-intent and install the breakpoint on every current replica and on every future replica that joins, restarts, or receives a migrated computation. The developer never manages a process list. This abstraction reduces the operational complexity of debugging from per-process to per-intent, regardless of deployment size.

Pause-the-world Policy. Pausing the world prevents in-flight requests from advancing while the developer inspects the system, decoupling the human speed of inspection from the execution speed of the cluster. For example, distributed backtrace could pause only the processes involved in a call chain, but other processes could continue executing and make the inspected state stale or inconsistent with the rest of the system. This challenge is inherent to interactive debugging: GDB similarly stops all threads by default, while providing a non-stop mode for independently controlling individual threads. Likewise, DDB allows developers to disable pause-the-world and pause only the processes involved in a debugging action, at the cost of accounting for state changes caused by processes that remain active.

Comparison with Consistent Snapshots. Classic snapshot algorithms [28] *record* a causally consistent cut while the system keeps running, but the recorded state may not match any global state the execution actually passed through, and it is a static copy that the developer cannot probe or modify. Pause-the-world instead *freezes* the execution itself: since a paused process cannot send messages, the resulting cut is always consistent, and once the last process stops, the frozen state is the actual global state of the system at that instant. In-flight messages stay buffered in the network stack and are delivered on resume, so the developer inspects, and may modify, a real reachable state rather than a recorded copy. The cost is that the pause is not instantaneous: later-paused processes keep executing briefly during the skew window (§4.3 discusses the resulting timeouts).

Logical Groups and Scope Resolution. DDB organizes processes into *logical groups*. By default, processes running the same binary belong to the same group. In the case of microservices, the default binary-based grouping policy puts all replicas of the same service under the same group. In the

`socialnet` example with 36 services at 5 replicas, the developer sees 36 groups rather than 180 processes; expanding a group reveals its members.

When a developer specifies a source-code location, DDB performs *automatic scope resolution*: it identifies which logical groups map that source file, presents them as selectable scopes, and lets the developer choose a target. This eliminates the need to know which binary hosts a source file—a non-trivial question in distributed application deployments. Internally, DDB maintains a two-level *source-map*, which maps source-code paths to logical groups and maps logical groups to running processes.

Intent Preservation Under Dynamics. Distributed applications are dynamic: auto-scaling, rolling restarts, and failover continuously change the process set. The control plane tracks these changes and continuously enforces each intent on the correct set of processes. Specifically, it maintains a continuous invariant: an intent is active on a process if and only if the process matches the intent's logical scope and its loaded binary maps the specified source-code location. When a new process joins the system, the control plane evaluates this invariant and applies the relevant intents immediately upon debugger attachment. Conversely, when a process terminates, its debugging state is cleanly garbage-collected. The intent is a *specification* (*e.g.,* "all replicas of service X should have a breakpoint inserted at line 42") and DDB continuously enforces it.

For frameworks that support computation migration (Nu [73], Quicksand [71, 72], ServiceWeaver [42]), breakpoints follow the computation automatically: because the control plane targets logical scopes rather than physical processes, the receiving process simply evaluates the invariant and applies the breakpoint locally. Heap state requires additional handling, as a migrated caller's heap may no longer reside on the original process; DDB can be extended to support these frameworks by supplying custom logic that locates and restores heap segments whenever the execution is paused (Appendix §E).

We note that DDB preserves intents across process churn, but it does not preserve a process's runtime state across restarts. Once a process terminates, its heap and stack state is lost, and DDB cannot restore it on the restarted process. For example, if a callee handler holds the caller context of a process that has since restarted, a distributed backtrace from that handler is truncated because the original caller process no longer exists.

Guarantees. *Invariant enforcement:* the control plane maintains the invariant that an intent is active on a process if and only if the process matches the intent's scope. When an intent is created, the command completes only after every current in-scope process acknowledges installation. When a process joins, it is held at an attach-time barrier until all pending in-scope intents are installed, so a matching process never runs without its intents. *Minimal developer expression:* developers operate at source-file and logical-group granularity, abstracted from physical process churn or state migration. *Composability:* intents compose orthogonally with DDB's other primitives; a developer can specify an intent, allow the system to continuously enforce it across churn, and subsequently issue commands like `dbt` against the paused state without managing the underlying topology.

Semantic	Description	Example POSIX APIs
`get_time`	Get the current timestamp	`clock_gettime` `gettimeofday`
`sleep_until`	Sleep until the absolute deadline	`pthread_cond_timedwait` `sem_timedwait`
`sleep`	Sleep for the relative duration	`sleep` `nanosleep`

Table 1. The Pause-Erased Time (PET) layer provides three semantics; each covers a subset of POSIX time APIs.

4.3 Pause-Erased Time (PET)

Problem. Every debugger-induced pause—breakpoint hit, single-step, manual interrupt—advances the system clock while the application is stopped. From the application's perspective, time jumps by the pause duration. Distributed applications fundamentally rely on these strict physical timing invariants for failure detection: a missed heartbeat triggers a leader election [64], a late RPC triggers a retry storm, and an expired lease triggers client disconnection. A developer pausing a process for ten seconds triggers every timeout below that threshold, forcing the distributed system into recovery procedures that permanently alter the state and destroy the intended debug flow. Because of this coupling, pause-based interactive debuggers have historically been considered impractical in distributed contexts [20, 25].

Abstraction. Pause-Erased Time (PET) gives each debugged process a pause-erased view of time: a clock in which all debugger pauses are invisible. Timestamps and timer waits operate against this clock. From the application's perspective, it was never paused, thus mitigating the timeout cascades caused by debugger-induced execution pauses. PET covers three broad categories of time semantics that implement a large portion of timeout and timer logic observed in distributed systems: reading the current timestamp (*e.g.,* `get_time`), sleeping until an absolute deadline (*e.g.,* `sleep_until(deadline)`), and sleeping for a relative duration (*e.g.,* `sleep(duration)`). These semantic categories map to a wide range of underlying POSIX APIs, as demonstrated in Table 1.

Pause-offset Accumulation. DDB records a timestamp when a given debuggee process is paused and another when it is resumed. The difference is the measured pause duration. DDB accumulates these into a running cumulative pause offset. Before resuming execution, DDB publishes this offset

to a local shim layer, which sits between the user-level process and the `libc` library to intercept time API calls. This shim layer intercepts POSIX time API calls and subtracts the cumulative offset from the `libc` and kernel's return value, erasing all pauses from the process's view of time.

Virtual Deadline Enforcement. The offset mechanism handles `get_time` correctly, but a subtler problem arises for processes that are *sleeping* when the debugger pauses them. On resume, the kernel observes that the real-time deadline has passed and wakes the thread immediately—before the pause-adjusted deadline has been reached. This *premature wakeup* can produce spurious timeouts.

PET prevents this with the *Virtual Deadline Enforcement* mechanism. The shim layer tracks the pause-adjusted absolute deadline for every active timer wait. On every return from the underlying `libc` sleep primitive—whether a genuine wakeup, a signal interruption, or a spurious wakeup from a debugger resume—the shim layer reads the current PET. If the current PET is earlier than the adjusted deadline, the deadline has not been reached in pause-erased time; the shim layer re-arms the wait for the remaining duration and returns to the kernel sleep. Application code does not execute until the PET deadline is genuinely reached. The shim layer re-traps the thread whenever a resume would wake it prematurely.

Virtual Deadline Enforcement is a fundamental requirement for interactive distributed debugging. Without it, dynamically erasing pauses would successfully adjust `get_time` but fail to prevent timeout cascades triggered by in-flight sleeps. Unlike virtual clocks in discrete event simulation [35, 47] or chaos engineering [23, 44, 52, 53]—which use static offsets or state rollbacks—PET handles a fundamentally different temporal anomaly: the application state remains static while real time progresses, requiring dynamic offset growth upon every pause.

Guarantees. PET satisfies two invariants; the formal correctness proofs are detailed in Appendix §A.1.

- *Monotonicity.* PET never decreases. Two consecutive `get_time` calls return non-decreasing values regardless of intervening pauses.
- *Timer correctness.* Time-blocking operations (both absolute deadlines via `sleep_until` and relative durations via `sleep`) return only when PET reaches or exceeds their target. Any pause during the sleep, and any premature kernel wakeup from a debugger resume, causes the shim to re-arm. Application post-sleep code executes only when the intended PET duration has genuinely passed.

These invariants ensure that application logic that is correct with respect to real time remains correct with respect to PET. Timeouts fire when and only when the application intends, as if no debugger were attached. PET works for both monotonic and wall-clock time APIs (Appendix §A.3).

PET Effectiveness under Global Pause. We distinguish between three types of timeouts related to DDB-attached processes: local timeouts, cross-process timeouts, and external timeouts. Local timeouts depend entirely on local state (*e.g.,* a periodic timer fired every N seconds) without relying on states from other components or services. PET is unconditionally effective for local timeouts because it directly virtualizes the local clock to minimize execution-pause-induced time jumps. In contrast, cross-process timeouts depend on communication with other attached processes, *e.g.,* a timer triggered when a process receives no response from a peer before a deadline, heartbeats between a leader and its followers, or a cache server waiting on a database reply. For these cross-process timeouts, PET is effective only if the global pause is tightly synchronized: a pending timeout can fire spuriously only if the pause-time skew between the involved processes exceeds the timeout's remaining slack. Therefore, to properly mitigate cross-process timeouts with PET, the communication delay between DDB and attached processes must be bounded so that all attached processes are paused nearly simultaneously. We state this as a deployment assumption rather than a property DDB enforces: the control plane is expected to share a low-latency network with the attached processes, as is typical in the development and testing environments DDB targets, so the pause skew remains well below common timeout values (*e.g.,* heartbeat intervals of hundreds of milliseconds). Finally, external timeouts originate from uncaptured services outside the attached cluster; we discuss them in the limitations below.

Limitations. PET maintains virtual time purely within the boundaries of the attached cluster. However, PET cannot virtualize the physical flow of time for external, true-time observers. If the cluster interacts with external systems that rely on physical time—such as an uninstrumented cloud storage service enforcing a strict lease expiration, or a third-party API gateway—those external services will perceive the developer's pause as a genuine timeout. Thus, DDB's temporal guarantees are strictly intra-cluster: developers must either mock time-sensitive external dependencies or attach them to DDB. For example, when debugging client–server interactions, attaching the load-generating clients as well allows DDB to pause the world and mask time jumps on both sides, so developers can debug the interaction without triggering unintended timeouts between the clients and the server.

5 Implementation

We implement DDB on the `x86_64` architecture and Linux. We also add experimental support for `aarch64`.

System Architecture. Figure 6 illustrates DDB's architecture, which comprises a centralized coordinator, a graphical frontend, and per-process distributed components. The

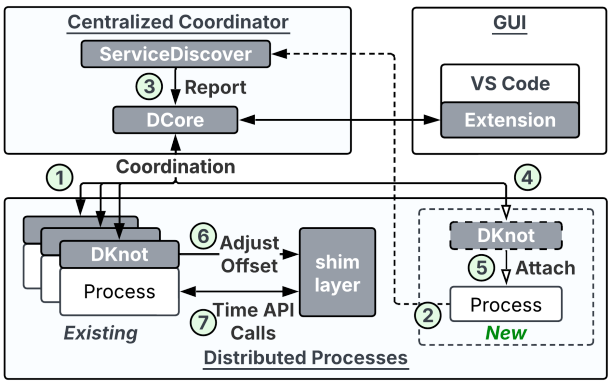

Figure 6. The design overview of DDB. Components in gray are DDB components.

	Component	LoC	Language
DDB	DCore	14,867	Rust
	DKnot	1,401	Python
	DConnector	1,954	C/C++/Go
	Adapter	≈700	TypeScript
Framework Support	gRPC	≈20	C++
	Nu	≈30	C++
	Quicksand	≈60	C++
	ServiceWeaver	≈10	Go
Ported Apps	socialnet	3,154	Go
Total		22,196	

Table 2. LoC for DDB's components, excluding `libfaketime`. Calculated by `cloc` [2], excluding blanks and comments.

VS Code Extension serves as the graphical user interface (GUI) for the developer. This frontend communicates with the *Centralized Coordinator*, which uses *ServiceDiscover* to track dynamic process topologies and report them to *DCore*. *DCore* acts as the central engine, orchestrating debugging sessions and distributing coordination messages to the relevant *Distributed Processes*. At the application layer, DDB pairs each target process with a *DKnot* agent (an extended version of GDB). *DKnot* attaches to the underlying process, relays debug-intents to it, and implements the `dbt`.

To prevent timeout cascades, *DKnot* sends offset adjustments to a locally interposed shim layer. This shim layer intercepts POSIX time API calls from the process to enforce Pause-Erased Time. The shim layer approach uses `LD_PRELOAD` to interpose on top of the `libc` interfaces and works with vDSO transparently (Appendix §A.4). Finally, a lightweight *DConnector* integration library (not pictured) embeds caller-context metadata within RPC payloads.

Interaction Between Components. The lifecycle of a debugging session follows the numbered steps in Figure 6. During normal execution, DCore maintains continuous coordination (①) with all active DKnot agents. When a new process spins up (②), ServiceDiscover detects its presence and reports (③) to DCore updating the global topology. DCore then immediately initializes a new DKnot instance (④) which performs an attach (⑤) to the new process, ensuring seamless debug-intent propagation across dynamic scaling. When a process hits a breakpoint, DCore pauses the world and the DKnot agents calculate the precise timestamp of the pause time. Upon resuming, each DKnot calculates the cumulative pause duration and publishes it as an offset to its local shim layer (⑥). The shim layer then intercepts all subsequent time API calls from the application (⑦), seamlessly applying the offset to enforce the Pause-Erased Time abstraction.

Codebase. To evaluate our realization, we added DDB support for four frameworks, gRPC [7], Nu [73], Quicksand [72], and ServiceWeaver [42]. We summarize the LoC in Table 2. To facilitate the evaluation of ServiceWeaver-based applications, we ported `socialnet` from DeathStarBench [37] to ServiceWeaver.

The modest integration effort reflects DDB's design philosophy: the communication framework layer is the ideal interposition point. Integrating DDB's caller-context embedding within the framework's payload handling path guarantees transparent cross-process stack reconstruction without heavy application-level instrumentation. This one-time, minimal integration cost (under 60 LoC per framework) instantly unlocks interactive distributed debugging for all downstream application code. We defer the discussion of more implementation-specific aspects to Appendix §C.

5.1 Limitations

Programming Language Support. Our prototype currently assumes GDB as the underlying debugger. Therefore, our prototype does not work for binaries and programming languages that GDB cannot interpret and manage.

PET. In our realization, DDB only provides PET on top of POSIX time APIs. Thus, it does not work in applications that directly use raw `rdtsc` or NIC hardware timestamp. However, three API semantics and two invariants ensure our techniques are transferable to other cases. PET's core mechanism is independent of the time source; only the interposition point differs. For example, as `rdtsc` cannot be intercepted via `LD_PRELOAD`, supporting it requires developers to route rdtsc through a PET wrapper. The wrapper can be implemented as a real rdtsc instruction offset by a global cumulative counter. We measured the worst-case time for the first wrapper call after the execution resume to be 70 ns, and successive calls have no measurable overhead (16 ns with or without wrapper).

In-flight Packets During Execution Pauses. DDB relies on the host OS's TCP receive buffers to queue in-flight packets during a pause, avoiding complex infrastructure-level checkpointing [27]. Thanks to PET, the application processes these buffered packets upon resumption exactly as if they

arrived over a congested link. Crucially, under DDB's pause-the-world behavior, clients and upstream services are also paused, halting transmission. Thus, TCP buffers trivially absorb the in-flight traffic without exhausting capacity, provided the cluster is not exposed to unmanaged, continuous external traffic.

6 Evaluation

We evaluate DDB to answer the following key questions:

1. How does DDB's integration effort compare to existing tools? (§6.1)
2. Does DDB improve diagnostic reliability and efficiency over existing tools? (§6.1)
3. Is DDB responsive enough for interactive debugging at distributed scale? (§6.2)
4. Does PET effectively prevent timeout cascades? (§6.3)
5. What is the runtime overhead of the DDB distributed control plane? (§6.4)

Setup. We evaluate DDB responsiveness on a cluster of 24 physical servers hosted on Chameleon [50]. The servers are `compute_cascadelake_r` instances located at the CHI@TACC site. Each server has an Intel Xeon Gold 6240R (24 cores, 2.4GHz), 187GB RAM, and Broadcom BCM57414 NIC. We evaluate DDB's PET effectiveness in minimizing time gaps on our local server, which has Intel Xeon Gold 5420+ (28 cores, 2.00 GHz) and 256GB RAM. We measure the DDB metadata embedding and attachment overhead on a cluster of 12 physical servers hosted on CloudLab [32]. These servers are `c6525-25g` instances (16-core AMD 7302P at 3.00GHz, 128GB RAM, Mellanox ConnectX-5 NIC).

Evaluation Scope. Because our prototype is currently tightly coupled with GDB, Java-based systems are outside the scope of our evaluation. Supporting Java-based systems would require DDB to integrate a Java-compatible debugger, such as `jdb`, as its backend. We also found relatively few open-source distributed systems that are implemented in C or C++ and use gRPC as their primary inter-component communication framework, further limiting the range of suitable evaluation targets.

Applications. Due to the evaluation scope constraint, we evaluate DDB with applications and frameworks that are readily available. To measure responsiveness and overhead across our target research frameworks (ServiceWeaver, Nu, Quicksand), we use the `socialnet` microservice benchmark as a standardized baseline. For gRPC, we evaluate a C++ Raft consensus implementation, which is derived from a course lab assignment and is representative of internal cluster communication. Finally, we isolate PET effectiveness using a synthetic application (§6.3) paired with a survey of real-world timeout thresholds.

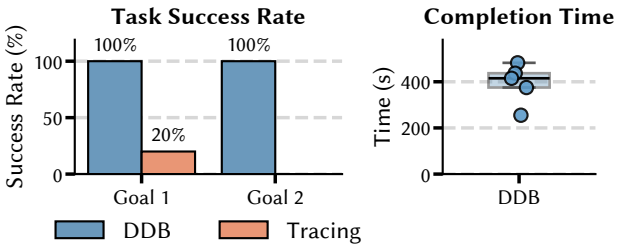

Figure 7. Left: Integration success rates for DDB vs. OpenTelemetry, where success requires viewing linear (G1) and concurrent (G2) caller–callee traces. **Right:** Completion time for successful DDB integrations (both G1 and G2). OpenTelemetry is not shown as no participant finished the full integration within the 20-minute time limit.

6.1 User Study

To evaluate DDB's ease of adoption and its impact on developer velocity, we conducted a two-part controlled user study. The study quantifies the gap between DDB and the status quo of debugging tools for distributed applications across two dimensions: the effort required to integrate the tools, and the diagnostic efficacy of DDB during real debugging tasks. Prior to the main study, we piloted our methodology by recruiting one volunteer for each study to conduct a trial-run, allowing us to refine the tasks and improve the clarity of the study materials.

Study 1: Integration Effort. The first study evaluates the operational overhead of deploying distributed debugging capabilities. We recruited 5 participants for this study, comprising graduate researchers from the ECE and CS departments with distributed systems experience. Each participant was asked to integrate DDB and OpenTelemetry into a C++ codebase using official documentation. Each tool integration task was allocated 20 minutes. A successful integration required two goals: tracing execution backward from callee to caller with local state visible (G1), and observing a concurrent fan-out pattern (G2). DDB's integration inherently satisfies both goals, whereas OpenTelemetry requires distinct, context-specific instrumentation for each. As shown in Figure 7, all 5 participants completed both goals with DDB (median 400 seconds), while only 1 completed G1 with OpenTelemetry and none completed G2.

Study 2: Diagnostic Efficacy. The second study evaluated diagnostic efficiency and reliability. This study employed a within-subjects (repeated measures) design to compare DDB against GDB and OpenTelemetry (OTel). A total of 9 participants were recruited for this study; each participant was required to solve three separate bug cases, utilizing a different tool for each case. To mitigate potential order effects, learning effects, and fatigue, the presentation sequence of the tools was partially counterbalanced using a 3×3 Latin Square design: DDB-GDB-OTel, GDB-OTel-DDB, or OTel-DDB-GDB. Across all conditions, participants had full access to structured logging (`spdlog` [58]) and a traffic-replay

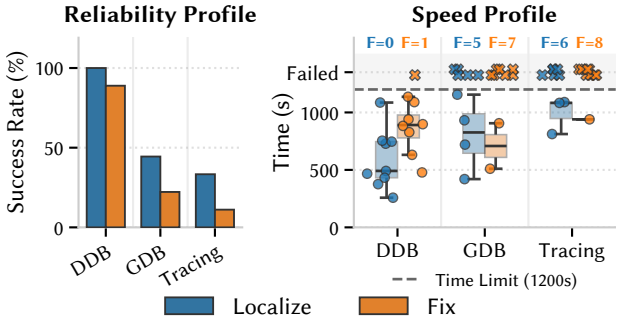

Figure 8. Aggregate reliability and diagnostic speed profiles across all trials. DDB reliably enabled participants to localize the root cause in 100% of trials, whereas baseline tools failed in the majority of cases and frequently reached the 1200-second timeout limit. Note that baseline boxplots may appear compact or low in certain cases; this reflects survivorship bias, as these distributions include only the few participants who localized the fault, while the majority timed out and are placed as failed cases.

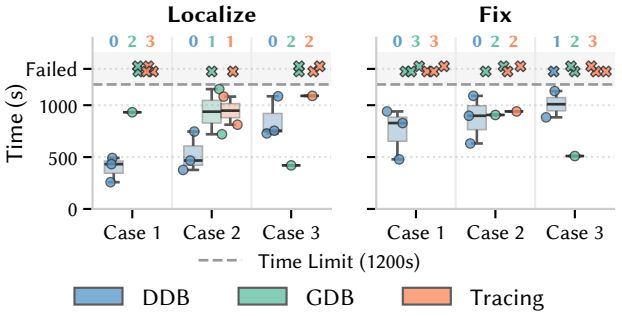

Figure 9. Case-by-case comparison of time-to-localize and time-to-fix across different bug cases. Case 1: Random Segmentation Fault; Case 2: Logic Error; Case 3: Distributed Deadlock. The GDB outlier in Case 3 corresponds to a participant who resolved the fault based on prior intuition rather than systematic tool-guided diagnosis.

utility (`grpcreplay` [79]), so the OTel baseline effectively represents the modern status quo: using a distributed tracer to identify the failing node, followed by iterative logging to find the root cause. For each task, we recorded the success rate and the total time required to localize and fix the bug, capped at a 1200-second (20-minute) time limit.

Methodology. Comparing DDB with GDB demonstrates the usability benefits of a unified control plane and evaluates the effectiveness of PET in masking unintended timeouts that would otherwise disrupt the debugging workflow. Comparing DDB with OTel evaluates DDB's debugging efficiency by contrasting interactive, source-level debugging with an iterative logging-based approach.

Case Selection Criteria. The three test cases were selected to highlight different diagnostic challenges: (1) A *segmentation fault* where an out-of-bounds memory access is nondeterministically triggered across two out of three Raft processes. (2) A *logic error* within the Raft consensus implementation, where a stale leader fails to correctly update its term after recovering from a network partition. (3) A *distributed deadlock* triggered by concurrent leader elections that paralyze cluster operations. All three cases are organic, drawn from course teaching staff implementations. All three require reasoning across multiple Raft processes (three to five nodes) and involve timing-sensitive protocol behavior, making them representative of cross-service faults. All three study cases are reproducible. In two cases (Case 1 and 2), however, the specific node on which the failure manifests is nondeterministic. DDB addresses this uncertainty through all-replica breakpoints and pause-the-world execution, allowing the failure to be captured regardless of which replica exhibits it.

Figure 8 summarizes the reliability and speed profiles across all trials. DDB enabled participants to localize the root cause in 100% of trials and fix the bug in 89% within the time limit. In contrast, GDB achieved only 44% localization and 22% fix rates; OTel achieved 33% and 11%, respectively. For successful trials, DDB also demonstrated high efficiency. Participants using DDB attained a median localization time of approximately 500 seconds across all bug cases. The case-by-case breakdown (Figure 9) confirms this consistency: in Cases 1 and 3, baseline tools timed out for most participants, while in Case 2, participants using baseline tools required substantially more time than those using DDB. Overall, these results indicate that DDB accelerates distributed debugging and makes complex diagnostic tasks easier to reason about.

Qualitative Analysis. Analysis of screen recordings of participant debugging procedures revealed two recurring patterns behind baseline failures. Participants assigned to OTel spent a disproportionate share of their session performing the instrumentation and switching between the trace visualizer and log collector, rather than performing diagnostic reasoning. This instrumentation and navigation overhead consumed substantial time and, in many cases, participants exceeded the time limit. GDB was effective for single-process faults but scaled poorly: participants managed three processes, but at five could no longer efficiently attach to and navigate the relevant threads. In timing-sensitive cases (Cases 2 and 3), pausing a process for inspection triggered RPC timeouts from its callers, which invalidated the cluster state and forced participants to restart the debugging session from scratch.

6.2 Responsiveness

An interactive debugger must respond to user commands and reconstruct state fast enough to maintain the developer's train of thought at human timescales. Table 3 presents the end-to-end latency of two basic control plane operations—executing a distributed backtrace (`dbt`) and continuing execution—across varying cluster sizes (38, 62, and 122 processes). Due to the space constraint, we present

Command	Scale (# pods)	Latency (ms)				
		Count	Mean	Median	P95	P99
dbt	38	2,348	32.5	14.2	128.5	153.9
	62	4,303	32.1	13.9	144.5	156.3
	122	14,573	26.5	13.3	110.1	160.1
continue	38	37	2.2	2	3	4.2
	62	8	2.8	2.7	3.6	3.6
	122	7	4.7	4.8	4.9	4.9

Table 3. Measured latencies for two commonly used commands, `dbt` and `continue`, under different cluster sizes. The **count** column shows the number of times the command is handled by DDB.

Call Depth	Latency (ms)				
	Mean	Median	StdDev	P95	P99
2	18.3	18.2	0.2	18.7	18.8
3	29.1	28.4	2.7	29.6	33.1
4	36.7	36.1	2.6	37.9	38.6
5	50.8	50.3	2.4	52.0	57.1
6	63.8	62.1	6.6	70.5	80.6
10	107.1	105.2	5.0	116.4	120.0

Table 4. Latencies of `dbt` command with varying call depths.

Time APIs	w/ PET (ns)	w/o PET (ns)
`chrono::system_clock`	91	30
`std::time`	86	4
`gettimeofday`	95	29
`clock_gettime(REALTIME)`	88	28
`clock_gettime(MONOTONIC)`	89	28

Table 5. Average execution time for time APIs with or without the PET shim layer interception.

the full measurement of command handling latencies in Appendix §D.

DDB exhibits stable, interactive-grade latency across all tested scales. The `continue` command consistently finishes under 5 ms because debug-intent distribution is heavily parallelized. Crucially, evaluating `dbt` latency serves as a rigorous stress test of DDB's responsiveness under heavy concurrent load. When execution pauses globally, the VS Code frontend automatically issues a burst of `dbt` commands—one for every thread across every paused process—resulting in over 14.6 k concurrent requests at the 122-pod scale. Despite this sheer volume of requests triggering a costly, iterative stack-stitching algorithm, `dbt` latency remains under 200 ms even at the tail.

Call Depth. Because distributed backtrace is an iterative algorithm, `dbt` handling latency is primarily bounded by call depth. A longer call chain entails larger latency, as DDB must cross more RPC boundaries, query more DKnot agents, and perform iterative stack restorations and frame stitching. To evaluate this impact, Table 4 isolates the end-to-end latency of a single `dbt` command as a function of RPC call depth. Reconstructing the distributed stack takes a median of 18.2 ms at a depth of two hops, and 36.1 ms at a depth of four hops. A recent study [46] reports that typical call depths observed in modern cloud applications average around 3 to 4. At these depths, the sub-50 ms `dbt` latency remains well within the target threshold for a fluid interactive experience. For long call chains, we discuss a mitigation in Appendix §B.2 that harnesses parallelism for a lower latency.

6.3 PET Effectiveness

To evaluate the effectiveness of the PET mechanism, we answer two questions: how much overhead does the interception layer introduce, and how effectively does it virtualize time to prevent unintended timeouts?

Overhead. We first measure the overhead of PET's interception by invoking common POSIX time APIs one million times. Compared to direct bare-metal execution, the intercepted time APIs introduce only nanosecond-scale overhead per call as described in Table 5. Furthermore, the routine that calculates and publishes the cumulative pause offset before a process resumes execution takes 1.265 μs on average. These negligible overheads confirm that the PET layer does not distort the normal execution performance of the debugged application.

Masking Time Jumps. To evaluate the effectiveness of PET at masking time jumps, we run a synthetic application and repeatedly retrieve timestamps using POSIX time APIs when PET is applied. The synthetic application executes a tight loop with a light workload (≈ 50 μs) in each iteration. We attach DDB to the synthetic application with PET enabled and repeatedly inject pause and resume commands while the application is running. In this experiment, we fix the interval between pause and resume at 100 ms; during real-world interactive debugging, this interval can be arbitrarily large. Because we tightly control the interval of each loop iteration to the microsecond scale, any noticeable time jumps at the millisecond scale are effectively captured. We present the timeline perceived by the synthetic application in Figure 10. The application consistently perceives ≤ 5 ms time jumps, which is clearly smaller than the 100 ms interval. Because PET accumulates and subtracts the cumulative pause offset, we observe similar sub-5 ms results regardless of the actual duration between pause and resume.

Real-World Safety Margin. To contextualize this 5 ms maximum drift, we examined default timing thresholds in three distributed systems, LogCabin [17], RAMCloud [67], and Nu [73]. As summarized in Table 6, these systems rely on timeouts to detect and handle failures, such as LogCabin's election timers and RAMCloud's fast crash recovery [65]. Exceeding these thresholds triggers non-negligible side effects, ranging from degraded performance via excessive heartbeats

Legend: ① Harm Performance ② Disrupt System States ③ Change System Behaviors

System	Case	Threshold	Aftermath	Effect Category	Severity
LogCabin [17]	Heartbeat Timeout	250 ms	Send excessive heartbeats	①	low
	Election Timeout	500 ms	Disrupt established leadership	① + ②	moderate
	Client Session Expiry	3600 s	Disconnect clients undesirably	② + ③	severe
RAMCloud [67]	RPC Timeout	100 - 200 ms *	Send excessive RPCs	①	low
	Lease Renewal Threshold	900 s	Force clients to renew the lease	②	moderate
	Transaction Timeout	$N \times 50$ ms †	Abort the transaction	① + ② + ③	severe
	Crash Recovery Trigger	250 ms	Initiate large-scale crash recovery	① + ② + ③	severe
Nu [73]	Full Shard Probing	400 ms	Probe full shards for freed memory	① + ②	moderate
	Connection Pool Timeout	10 s	Connections considered dead	① + ② + ③	severe
	Service Keepalive	10 s	Connections considered idle	② + ③	severe

* Value is chosen randomly between 100 - 200 ms. † N is the number of participants involved in the transaction.

Table 6. Timeout thresholds used in real-world systems, with their aftermath, effect category, and severity.

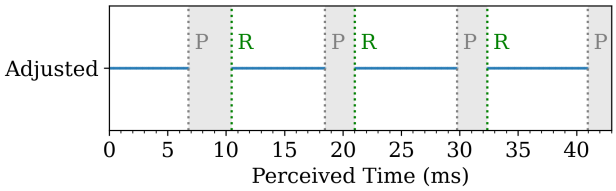

Figure 10. The application-perceived time (PET) with multiple pauses (P) and resumes (R). The gray area shows the time jumps that the application perceives between an execution pause and resume.

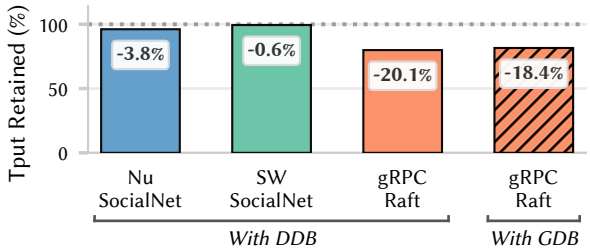

Figure 11. The performance overhead when DDB metadata is embedded, and DDB is attached across Nu's `socialnet`, ServiceWeaver (SW)'s `socialnet`, and gRPC-based Raft consensus implementation.

to disrupted system states via forced leader step-downs or a massive crash recovery. Our survey indicates that the most aggressive timeout thresholds in these systems typically range from 50 ms to 250 ms. Because PET reliably bounds the perceived time jump to under 5 ms, it provides an order-of-magnitude safety envelope that isolates the application from debugger-induced timeout cascades.

6.4 DDB Runtime Overhead on Applications

While DDB targets staging and test environments, its runtime impact must be comparable to a traditional single-process source-level debugger and incur no substantial overhead from the orchestration and management of distributed processes. Figure 11 plots the retained throughput of the `socialnet` benchmark (on both the Nu and ServiceWeaver runtimes) and a gRPC-based Raft consensus implementation with and without DDB attached.

DDB introduces minimal throughput degradation on `socialnet`. Specifically, `socialnet` running on ServiceWeaver and Nu exhibits a 0.6% and 3.8% throughput drop, respectively. To establish a baseline for binary-level debugging overhead, we also compare DDB against GDB. Attaching GDB to the gRPC Raft service yields an 18.4% throughput degradation. Attaching DDB to the same gRPC Raft service yields a 20.1% throughput degradation. This demonstrates that DDB achieves distributed debugging capabilities with overhead comparable to a traditional single-process debugger. The caller-context metadata embedding required for `dbt` adds only 52 bytes to each RPC payload, resulting in a minimal impact on network utilization.

7 Related Work

Formal methods and model checking. Formal verification and model checking [43, 62, 83] provide strong correctness guarantees for protocol specifications but operate on mathematical models, not running code. Verification-guided tests are effective [81, 85], but they require substantial expertise. Runtime verification systems [21, 55] check specifications against live executions but do not enable interactive inspection or modification of the state that triggered a violation.

Record and Replay. Capturing and reproducing non-determinism are crucial in distributed applications [51, 70, 91]. Some work [39, 40] also explores replay interfaces that resemble a traditional symbolic debugger, *e.g.*, GDB. Record-and-replay is orthogonal to DDB: it targets non-deterministic bugs by replaying a recorded execution, whereas DDB operates on live, real-time execution.

Distributed logging, tracing, and monitoring. These frameworks enhance the logging and reasoning experience

by automatically aggregating logs [3, 4, 11], establishing causality between cross-service interactions and improving observability [8, 12, 13, 22, 57, 76, 77], and enabling tracing-based aftermath-analysis to reveal erratic symptoms [36, 45, 49, 88]. Monitoring tools [6, 68] can also provide high-level metrics to indicate potential issues. While these tools are effective at identifying *which* service is behaving anomalously or reconstructing the sequence of events around a failure [31, 84], they do not provide source-level interactive debugging. DDB addresses a different need: once a developer knows *where* a problem occurs, they need to understand *why* by pausing execution, navigating the cross-service call chain, and inspecting live runtime state. Unlike tracing, DDB's metadata is not sampled and produces no persistent records; it is interrogated on demand.

HPC/Distributed debuggers. TotalView [16] and Linaro DDT [9] are interactive debuggers for HPC applications that support multi-process debugging at scale. Although they provide mechanisms for managing breakpoints across MPI processes, they are designed for HPC's homogeneous execution model: they do not reconstruct call stacks across RPC boundaries, and they offer no protection against timeout interference from debugger pauses [90]. Both are also proprietary. Ray [61] features a built-in interactive source-level debugger [14] that provides a single-point view of distributed tasks within the Ray framework. However, Ray's debugger does not offer distributed backtrace and still requires developers to manually cross-reference call stacks across process boundaries.

Classic distributed debugging. Early research in distributed debugging focused heavily on algorithms for consistent global snapshots [28], distributed halting [60], and determining causal ordering of asynchronous message passing [33]. However, modern cloud-native architectures have shifted the problem space. Today's systems are characterized by deep RPC chains [77], dynamically scaling replica sets [36], and strict timeout-driven availability requirements [30]. Where classic systems focused on theoretically consistent state spaces for asynchronous events, DDB addresses practical developer workflow bottlenecks by providing an intent-preserving control plane that manages debug state across ephemeral infrastructure.

8 Discussion

Debugger backend. As of this writing, DDB has been thoroughly tested with GDB, and we have implemented experimental support for LLDB. Adding a new backend primarily requires DDB to parse and interpret the backend's commands, responses, and execution states through its native communication interface (*e.g.,* GDB's Machine Interface). Because DDB relies on a small set of custom debugger commands to implement distributed backtraces and pause-erased time, the target debugger must also provide sufficient extensibility, as GDB does through Python scripting.

Architecture assumptions. DDB's distributed backtrace relies on architecture-specific register layouts and calling conventions to reconstruct call stacks. The other functionality of DDB is not tied to a specific architecture, provided that the debugger backend supports that architecture. We have implemented and validated our stack-restoration approach on `aarch64` as well, with further implementation details in Appendix §C.

OS assumptions. The current PET implementation assumes conventional kernel-level semantics for POSIX-compliant time APIs, so its correctness may depend on the target kernel: during prototyping, we found that several time APIs on macOS (Darwin) differ from their Linux counterparts despite sharing the same signatures.

When DDB helps and when it does not. DDB is particularly useful in early development and prototyping, where the instrumentation is itself the bottleneck, as our user study shows. It complements logs and traces: once those surface a clue, the developer can inspect the suspected code paths from a chosen entrypoint, not by wading through large volumes of data. Its closer counterpart is the iterative log-and-redeploy loop developers use to understand why a reproducible failure occurs. Reproducing distributed failures is genuinely hard, and DDB does not try to solve it; for non-reproducible production incidents in hyperscaler-operated systems, observability tools and record-replay remain the right tools.

9 Conclusion

Interactive debugging enables developers to pause execution, inspect runtime state, and navigate call stacks at the source level, yet it has been widely treated as impractical for distributed systems. We present DDB, a distributed interactive debugger that restores this tight hypothesis-testing loop through three user-space mechanisms—Distributed Backtrace, an intent-preserving control plane, and Pause-Erased Time—without requiring kernel modifications.

In a controlled user study, DDB achieves 100% fault localization, compared to 38.5% for baseline tools. It delivers 30 ms median backtrace latency and 1–5% throughput overhead, and requires only 10–60 lines of per-framework integration.

Acknowledgments

We thank our shepherd and reviewers for their invaluable feedback. We thank the members of the USC Networked Systems Lab (NSL) for their insightful discussions. We appreciate the participants of the user study for their time. Results presented in this paper were partially obtained using the Chameleon and CloudLab testbeds supported by the National Science Foundation. This work was funded in part by a National Research Foundation of Korea (NRF) grant (RS-2024-00354947).

References

[1] Akka: Build and Run Apps That React to Change. https://akka.io. Accessed: 2025-02-25.

[2] AlDanial/Cloc. https://github.com/AlDanial/cloc. Accessed: 2025-05-06.

[3] Azure Monitor Logs - Azure Monitor. https://learn.microsoft.com/en-us/azure/azure-monitor/logs/data-platform-logs. Accessed: 2024-10-20.

[4] Cloud Logging on Google Cloud. https://cloud.google.com/logging. Accessed: 2024-10-20.

[5] Ephemeral Containers. https://kubernetes.io/docs/concepts/workloads/pods/ephemeral-containers/. Accessed: 2024-12-10.

[6] Grafana: The Open and Composable Observability Platform. https://grafana.com/. Accessed: 2025-02-27.

[7] gRPC. https://grpc.io/. Accessed: 2024-10-20.

[8] Jaeger: Open Source, Distributed Tracing Platform. https://www.jaegertracing.io/. Accessed: 2024-11-14.

[9] Linaro DDT. https://www.linaroforge.com/linaro-ddt/. Accessed: 2025-05-14.

[10] LLDB. https://lldb.llvm.org/. Accessed: 2025-09-15.

[11] LogDevice: A Distributed Data Store for Logs. https://engineering.fb.com/2017/08/31/core-infra/logdevice-a-distributed-data-store-for-logs/. Accessed: 2024-10-20.

[12] OpenTelemetry. https://opentelemetry.io/. Accessed: 2024-10-20.

[13] OpenZipkin: A Distributed Tracing System. https://zipkin.io/. Accessed: 2024-11-14.

[14] Ray Debugger. https://docs.ray.io/en/latest/ray-observability/user-guides/debug-apps/ray-debugging.html. Accessed: 2024-11-04.

[15] RR: Lightweight Recording & Deterministic Debugging. https://rr-project.org/. Accessed: 2025-09-15.

[16] TotalView: Debugger for HPC Computing. https://totalview.io/. Accessed: 2025-05-14.

[17] Logcabin/Logcabin. https://github.com/logcabin/logcabin. Accessed: 2024-11-22.

[18] Dapr: Distributed Application Runtime. https://dapr.io. CNCF Graduated Project, 2024.

[19] GDB: The GNU Project Debugger. https://www.sourceware.org/gdb/

[20] Rohan Achar, Pritha Dawn, and Cristina V. Lopes. GoTcha: An Interactive Debugger for GoT-based Distributed Systems. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2019)*. Association for Computing Machinery, New York, NY, USA, 94–110. doi:10.1145/3359591.3359733

[21] Kalev Alpernas, Aurojit Panda, Leonid Ryzhyk, and Mooly Sagiv. Cloud-Scale Runtime Verification of Serverless Applications. In *Proceedings of the ACM Symposium on Cloud Computing*. ACM, Seattle WA USA, 92–107. doi:10.1145/3472883.3486977

[22] Sachin Ashok, Vipul Harsh, Brighten Godfrey, Radhika Mittal, Srinivasan Parthasarathy, and Larisa Shwartz. TraceWeaver: Distributed Request Tracing for Microservices Without Application Modification. In *Proceedings of the ACM SIGCOMM 2024 Conference*. ACM, Sydney NSW Australia, 828–842. doi:10.1145/3651890.3672254

[23] Earl T. Barr and Mark Marron. Tardis: Affordable Time-Travel Debugging in Managed Runtimes. *ACM SIGPLAN Notices* 49, 10 (Oct. 2014), 67–82. doi:10.1145/2714064.2660209

[24] Philip A Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. Orleans: Distributed Virtual Actors for Programmability and Scalability.

[25] Ivan Beschastnikh, Patty Wang, Yuriy Brun, and Michael D. Ernst. Debugging Distributed Systems. *Commun. ACM* 59, 8 (July 2016), 32–37. doi:10.1145/2909480

[26] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. *SIGPLAN Not.* 30, 8 (Aug. 1995), 207–216. doi:10.1145/209937.209958

[27] Anton Burtsev, Prashanth Radhakrishnan, Mike Hibler, and Jay Lepreau. Transparent Checkpoints of Closed Distributed Systems in Emulab. In *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys '09)*. Association for Computing Machinery, New York, NY, USA, 173–186. doi:10.1145/1519065.1519084

[28] K. Mani Chandy and Leslie Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. Comput. Syst.* 3, 1 (Feb. 1985), 63–75. doi:10.1145/214451.214456

[29] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 1–22.

[30] Jeffrey Dean and Luiz André Barroso. The Tail at Scale. *Commun. ACM* 56, 2 (Feb. 2013), 74–80. doi:10.1145/2408776.2408794

[31] Pradeep Dogga, Karthik Narasimhan, Anirudh Sivaraman, Shiv Kumar Saini, George Varghese, and Ravi Netravali. Revelio: ML-generated Debugging Queries for Distributed Systems. doi:10.48550/arXiv.2106.14347 arXiv:2106.14347 [cs]

[32] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The Design and Operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 1–14. https://www.flux.utah.edu/paper/duplyakin-atc19

[33] J. Fowler and W. Zwaenepoel. Causal Distributed Breakpoints. In *Proceedings.,10th International Conference on Distributed Computing Systems*. 134–141. doi:10.1109/ICDCS.1990.89277

[34] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating Interference at Microsecond Timescales. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, USA, 281–297. doi:10.5555/3488766.3488782

[35] Richard M. Fujimoto. Parallel Discrete Event Simulation. *Commun. ACM* 33, 10 (Oct. 1990), 30–53. doi:10.1145/84537.84545

[36] Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. Sage: Practical and Scalable ML-driven Performance Debugging in Microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 135–151. doi:10.1145/3445814.3446700

[37] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, Providence RI USA, 3–18. doi:10.1145/3297858.3304013

[38] Dennis Gannon, Roger Barga, and Neel Sundaresan. Cloud-Native Applications. *IEEE Cloud Computing* 4, 5 (Sept. 2017), 16–21. doi:10.1109/MCC.2017.4250939

[39] Dennis Geels, Gautam Altekar, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Friday: Global Comprehension for Distributed Replay. In *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation (NSDI'07)*. USENIX Association, USA, 21.

[40] Dennis Geels, Gautam Altekar, Scott Shenker, and Ion Stoica. Replay Debugging for Distributed Applications. In *2006 USENIX Annual Technical Conference (USENIX ATC 06)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/2006-usenix-annual-technical-conference/replay-debugging-distributed-applications

[41] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*. 29–43.

[42] Sanjay Ghemawat, Robert Grandl, Srdjan Petrovic, Michael Whittaker, Parveen Patel, Ivan Posva, and Amin Vahdat. Towards Modern Development of Cloud Applications. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems (HotOS '23)*. Association for Computing Machinery, New York, NY, USA, 110–117. doi:10.1145/3593856.3595909

[43] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. IronFleet: Proving Practical Distributed Systems Correct. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 1–17. doi:10.1145/2815400.2815428

[44] Wolfgang Hommel. Wolfcw/Libfaketime. https://github.com/wolfcw/libfaketime.

[45] Lexiang Huang and Timothy Zhu. Tprof: Performance Profiling via Structural Aggregation and Automated Analysis of Distributed Systems Traces. In *Proceedings of the ACM Symposium on Cloud Computing*. ACM, Seattle WA USA, 76–91. doi:10.1145/3472883.3486994

[46] Darby Huye, Yuri Shkuro, and Raja R. Sambasivan. Lifting the Veil on Meta's Microservice Architecture: Analyses of Topology and Request Workflows. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 419–432. https://www.usenix.org/conference/atc23/presentation/huye

[47] David Jefferson. Virtual Time. *ACM Transactions on Programming Languages and Systems (TOPLAS)* (1985). https://dl.acm.org/doi/10.1145/3916.3988

[48] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. Cloud Programming Simplified: A Berkeley View on Serverless Computing. http://arxiv.org/abs/1902.03383. doi:10.48550/arXiv.1902.03383 arXiv:1902.03383

[49] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, Vinod Venkataraman, Kaushik Veeraraghavan, and Yee Jiun Song. Canopy: An End-to-End Performance Tracing And Analysis System. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. ACM, Shanghai China, 34–50. doi:10.1145/3132747.3132749

[50] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbach, Alex Rocha, and Joe Stubbs. Lessons Learned from the Chameleon Testbed. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association.

[51] Shreyas Kharbanda and Pedro Fonseca. Always-On Recording Framework for Serverless Computations: Opportunities and Challenges. In *Proceedings of the 1st Workshop on SErverless Systems, Applications and MEthodologies*. ACM, Rome Italy, 41–49. doi:10.1145/3592533.3592810

[52] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging Operating Systems with Time-Traveling Virtual Machines. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC '05)*. USENIX Association, USA, 1. https://www.usenix.org/conference/2005-usenix-annual-technical-conference/debugging-operating-systems-time-traveling

[53] K Kingsbury. Jepsen-Io/Jepsen. https://github.com/jepsen-io/jepsen Accessed: 2025-09-15.

[54] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS operating systems review* 44, 2 (2010), 35–40.

[55] Xuezheng Liu, Zhenyu Guo, Xi Wang, Feibo Chen, Xiaochen Lian, Jian Tang, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. D3S: Debugging Deployed Distributed Systems. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI'08)*. USENIX Association, USA, 423–437.

[56] Shiqing Ma, Juan Zhai, Yonghwi Kwon, Kyu Hyung Lee, Xiangyu Zhang, Gabriela Ciocarlie, Ashish Gehani, Vinod Yegneswaran, Dongyan Xu, and Somesh Jha. Kernel-Supported Cost-Effective Audit Logging for Causality Tracking. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 241–254. https://www.usenix.org/conference/atc18/presentation/ma-shiqing

[57] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 378–393. doi:10.1145/2815400.2815415

[58] Gabi Melman. Gabime/Spdlog. https://github.com/gabime/spdlog

[59] Inc. Meta. Folly Fiber. https://github.com/facebook/folly/blob/main/folly/fibers/README.md Accessed: 2026-03-10.

[60] B.P. Miller and J.-D. Choi. Breakpoints and Halting in Distributed Programs. In *[1988] Proceedings. The 8th International Conference on Distributed*. 316–323. doi:10.1109/DCS.1988.12532

[61] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A Distributed Framework for Emerging AI Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 561–577. https://www.usenix.org/conference/osdi18/presentation/moritz

[62] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How Amazon Web Services Uses Formal Methods. *Commun. ACM* 58, 4 (March 2015), 66–73. doi:10.1145/2699417

[63] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, Philadelphia, PA, 305–319. https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro

[64] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'14)*. USENIX Association, USA, 305–320.

[65] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast Crash Recovery in RAMCloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, Cascais Portugal, 29–41. doi:10.1145/2043556.2043560

[66] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-Sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 361–378. doi:10.5555/3323234.3323265

[67] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang. The RAMCloud Storage System. *ACM Transactions on Computer Systems* 33, 3 (Sept. 2015), 1–55. doi:10.1145/2806887

[68] Björn Rabenstein and Julius Volz. Prometheus: A Next-Generation Monitoring System. (2015). https://www.usenix.org/conference/srecon15europe/program/presentation/rabenstein

[69] Lenin Ravindranath, Jitendra Padhye, Sharad Agarwal, Ratul Mahajan, Ian Obermiller, and Shahin Shayandeh. AppInsight: Mobile App Performance Monitoring in the Wild. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. 107–120. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/ravindranath

[70] Michiel Ronsse and Koen De Bosschere. RecPlay: A Fully Integrated Practical Record/Replay System. *ACM Trans. Comput. Syst.* 17, 2 (May 1999), 133–152. doi:10.1145/312203.312214

[71] Zhenyuan Ruan, Shihang Li, Kaiyan Fan, Marcos K. Aguilera, Adam Belay, Seo Jin Park, and Malte Schwarzkopf. Unleashing True Utility Computing with Quicksand. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*. ACM, Providence RI USA, 196–205. doi:10.1145/3593856.3595893

[72] Zhenyuan Ruan, Shihang Li, Kaiyan Fan, Seo Jin Park, Marcos K. Aguilera, Adam Belay, and Malte Schwarzkopf. Quicksand: Harnessing Stranded Datacenter Resources with Granular Computing. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. 147–165. https://www.usenix.org/conference/nsdi25/presentation/ruan

[73] Zhenyuan Ruan, Seo Jin Park, Marcos K. Aguilera, Adam Belay, and Malte Schwarzkopf. Nu: Achieving Microsecond-Scale Resource Fungibility with Logical Processes. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 1409–1427. https://www.usenix.org/conference/nsdi23/presentation/ruan

[74] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K Aguilera, and Adam Belay. AIFM: High-Performance, Application-Integrated Far Memory. In *NSDI*.

[75] Alireza Sahraei, Soteris Demetriou, Amirali Sobhgol, Haoran Zhang, Abhigna Nagaraja, Neeraj Pathak, Girish Joshi, Carla Souza, Bo Huang, Wyatt Cook, Andrii Golovei, Pradeep Venkat, Andrew Mcfague, Dimitrios Skarlatos, Vipul Patel, Ravinder Thind, Ernesto Gonzalez, Yun Jin, and Chunqiang Tang. XFaaS: Hyperscale and Low Cost Serverless Functions at Meta. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 231–246. doi:10.1145/3600006.3613155

[76] Junxian Shen, Han Zhang, Yang Xiang, Xingang Shi, Xinrui Li, Yunxi Shen, Zijian Zhang, Yongxiang Wu, Xia Yin, Jilong Wang, Mingwei Xu, Yahui Li, Jiping Yin, Jianchang Song, Zhuofeng Li, and Runjie Nie. Network-Centric Distributed Tracing with DeepFlow: Troubleshooting Your Microservices in Zero Code. In *Proceedings of the ACM SIGCOMM 2023 Conference*. ACM, New York NY USA, 420–437. doi:10.1145/3603269.3604823

[77] Benjamin H Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. ([n. d.]).

[78] Terry Stanley, Tyler Close, and Mark S. Miller. *Causeway: a message-oriented distributed debugger*. Technical Report. HP Labs. http://www.hpl.hp.com/techreports/2009/HPL-2009-78.html HP Labs tech report HPL-2009-78.

[79] Vearne. Vearne/Grpcreplay. https://github.com/vearne/grpcreplay

[80] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D. Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Semeru: A {Memory-Disaggregated} Managed Runtime. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 261–280. https://www.usenix.org/conference/osdi20/presentation/wang

[81] Dong Wang, Wensheng Dou, Yu Gao, Chenao Wu, Jun Wei, and Tao Huang. Model Checking Guided Testing for Distributed Systems. In *Proceedings of the Eighteenth European Conference on Computer Systems*. ACM, Rome Italy, 127–143. doi:10.1145/3552326.3587442

[82] Lingmei Weng, Peng Huang, Jason Nieh, and Junfeng Yang. Argus: Debugging Performance Issues in Modern Desktop Applications with Annotated Causal Tracing. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 193–207. https://www.usenix.org/conference/atc21/presentation/weng

[83] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. Verdi: A Framework for Implementing and Formally Verifying Distributed Systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, Portland OR USA, 357–368. doi:10.1145/2737924.2737958

[84] Thorsten Wittkopp, Philipp Wiesner, and Odej Kao. LogRCA: Log-Based Root Cause Analysis for Distributed Services. doi:10.48550/arXiv.2405.13599 arXiv:2405.13599 [cs]

[85] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: Transparent Model Checking of Unmodified Distributed Systems. 213–228.

[86] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 249–265. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/yuan

[87] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. Characterizing Logging Practices in Open-Source Software. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Zurich, Switzerland, 102–112. https://dl.acm.org/doi/10.5555/2337223.2337236

[88] Lei Zhang, Zhiqiang Xie, Vaastav Anand, Ymir Vigfusson, and Jonathan Mace. The Benefit of Hindsight: Tracing Edge-Cases in Distributed Systems. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 321–339. https://www.usenix.org/conference/nsdi23/presentation/zhang-lei

[89] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study. *IEEE Transactions on Software Engineering* 47, 2 (Feb. 2021), 243–260. doi:10.1109/TSE.2018.2887384

[90] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study. *IEEE Trans. Softw. Eng.* 47, 2 (Feb. 2021), 243–260. doi:10.1109/TSE.2018.2887384

[91] Gefei Zuo, Jiacheng Ma, Andrew Quinn, Pramod Bhatotia, Pedro Fonseca, and Baris Kasikci. Execution Reconstruction: Harnessing Failure Reoccurrences for Failure Reproduction. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. ACM, Virtual Canada, 1155–1170. doi:10.1145/3453483.3454101

Appendix

Following the practice at SOSP'26, we declare that the appendix and included materials were not peer reviewed by the SOSP committee.

A PET Mechanism

A.1 Formal Description and Invariants Correctness

We formalize the Pause-Erased Time (PET) abstraction and prove two invariants that guarantee PET induces no incorrect timing behavior. Let $t \in \mathbb{R}_{\geq 0}$ denote the continuous absolute physical time. Suppose the debugger suspends the target process multiple times during a debugging session. We represent these debugger-induced pauses as a sequence of disjoint, half-open physical time intervals $G_k = [p_k, r_k)$. Here, p_k and r_k are the exact physical times on the t continuum when the process pauses and resumes, respectively. During each pause interval G_k, the process executes no user code and makes no time API calls. To measure the duration of these pauses from within the process, we rely on the operating system's timekeeping. Let $T(t)$ denote the value of the kernel's monotonic clock at any given physical time t. The shim layer measures the duration of each pause G_k by recording two internal timestamps from this kernel clock T. Specifically, it records $t_{\text{stop}}^{(k)}$ immediately after all threads are stopped, and $t_{\text{resume}}^{(k)}$ immediately before execution resumes. Because these internal timestamp reads inevitably occur sequentially within the physical pause interval $[p_k, r_k)$, their recorded values are strictly bounded by the true kernel clock values at the interval's boundaries: $T(p_k) \leq t_{\text{stop}}^{(k)} < t_{\text{resume}}^{(k)} \leq T(r_k)$. The shim layer defines the measured pause duration Δ_k for the interval G_k as the difference between these two internal timestamps.

$$\Delta_k \;=\; t_{\text{resume}}^{(k)} - t_{\text{stop}}^{(k)} \;\in\; [0,\; T(r_k) - T(p_k)].$$

As the process resumes at physical time r_k, the shim layer adds Δ_k to a global cumulative offset $O(t)$. Formally, we define $O(t) = \sum_{k:\, r_k \leq t} \Delta_k$. When the application subsequently calls a time API, the shim layer intercepts the call and returns the virtualized time $T_v(t) = T(t) - O(t)$. This mechanism implements three time-related semantics: (i) `get_time` directly returns $T_v(t)$. (ii) `sleep_until`(D_v) safely blocks the thread and returns at the earliest physical time $t^\star$ such that $T_v(t^\star) \geq D_v$. To achieve this, the shim layer repeatedly arms the kernel's blocking system call with an absolute physical deadline $D_{\text{real}} = D_v + O(t_{\text{now}})$. Whenever the kernel wakes the thread prematurely—either due to the offset O increasing from an intervening pause or a spurious signal—the shim layer intercepts the wakeup, recalculates D_{real}, and re-arms the sleep until the virtual condition holds. (iii) `sleep`(D) is syntactic sugar for sleeping until a relative duration elapses, implemented as `sleep_until`$(T_v(t_{\text{now}}) + D)$.

Lemma 1 (Cumulative offset property)**.** *For any physical times $t_1 < t_2$, the change in the cumulative offset is exactly the sum of measured pause durations that concluded within that interval. Specifically, $O(t_2) - O(t_1) = \sum_{k:\, t_1 < r_k \leq t_2} \Delta_k$.*

Proof. By definition, $O(t)$ is a right-continuous step function that only increases by Δ_k precisely at $t = r_k$. The function otherwise remains constant during all running intervals and pauses. Therefore, the difference $O(t_2) - O(t_1)$ perfectly isolates and sums the increments of those steps where $r_k \in (t_1, t_2]$. □

Invariant 1: Monotonicity. The Pause-Erased Time never decreases. Let two `get_time` calls occur at physical times $t_1 < t_2$, returning $v_1 = T_v(t_1)$ and $v_2 = T_v(t_2)$. Then $v_2 \geq v_1$.

Proof. Because the application is entirely suspended during any pause interval G_k, the observation times t_1 and t_2 must strictly fall within running intervals. Consequently, any pause interval G_k that intersects the window $(t_1, t_2]$ must be fully contained within it, implying $t_1 < p_k < r_k \leq t_2$. We can partition the timeframe $(t_1, t_2]$ into a collection of maximal running intervals $\mathcal{R}$ and a collection of paused intervals $\mathcal{P} = \{G_k \mid G_k \subseteq (t_1, t_2]\}$. The total elapsed real time measured by the kernel clock decomposes over these intervals.

$$\begin{aligned} T(t_2) - T(t_1) = &\sum_{I \in \mathcal{R}} \big[T(\max I) - T(\min I)\big] \\ &+ \sum_{G_k \in \mathcal{P}} \big[T(r_k) - T(p_k)\big] \end{aligned}$$

We expand the difference in virtual time using Lemma 1.

$$\begin{aligned} v_2 - v_1 &= \big(T(t_2) - T(t_1)\big) - \big(O(t_2) - O(t_1)\big) \\ &= \sum_{I \in \mathcal{R}} \big[T(\max I) - T(\min I)\big] \\ &\quad + \sum_{G_k \in \mathcal{P}} \big[T(r_k) - T(p_k) - \Delta_k\big] \end{aligned}$$

The kernel's monotonic clock strictly non-decreases, making the first sum over the running intervals naturally non-negative. For the second sum, earlier definitions establish that the measured pause Δ_k is bounded by the true interval duration, meaning $\Delta_k \leq T(r_k) - T(p_k)$. Because both sums are non-negative, the virtual time successfully preserves monotonicity such that $v_2 - v_1 \geq 0$. □

Invariant 2: Timer Correctness. Time-blocking operations unblock strictly when their virtual time constraints are satisfied. Specifically, a call to `sleep_until`(D_v) returns at the earliest physical time $t^\star$ where $T_v(t^\star) \geq D_v$. Consequently, a relative `sleep`(D) issued at physical time t_0 returns at an unblocking instant $t^\star$ that guarantees $T_v(t^\star) - T_v(t_0) \geq D$.

Proof. When an application blocks using `sleep_until`(D_v), the shim layer arms the underlying kernel sleep primitive with the absolute physical deadline $D_{\text{real}} = D_v + O(t_{\text{now}})$.

The shim layer subsequently intercepts every wakeup. It unconditionally returns control to the application only when the virtual time satisfies $T_v(t) \geq D_v$. We must guarantee that the armed physical deadline D_{real} mathematically expires no later than the true physical time when the virtual condition is satisfied. Suppose at some subsequent physical time t_{target}, the virtual time reaches the deadline such that $T_v(t_{\text{target}}) \geq D_v$. By definition, this implies $T(t_{\text{target}}) - O(t_{\text{target}}) \geq D_v$, which rearranges algebraically to $T(t_{\text{target}}) \geq D_v + O(t_{\text{target}})$. Because the cumulative offset never decreases, we have $O(t_{\text{target}}) \geq O(t_{\text{now}})$. Therefore, we are guaranteed that $T(t_{\text{target}}) \geq D_v + O(t_{\text{now}}) = D_{\text{real}}$. This inequality is essential: it proves that the initially armed kernel deadline D_{real} will inevitably expire at or strictly prior to any physical time where the virtual deadline is satisfied. If no intervening pauses occur, the offset remains unchanged and the kernel wakes the thread exactly when the valid condition $T_v(t) \geq D_v$ is met. However, if an intervening pause occurs, the offset increases, causing D_{real} to expire prematurely while the actual virtual time $T_v(t)$ remains strictly less than D_v. The shim layer automatically traps this premature wakeup, recalculates a pushed-back D_{real} using the updated offset, and re-arms the sleep. By repeatedly filtering out these premature wakeups through persistent recalculation, the shim layer guarantees that the application ultimately unblocks precisely at the earliest observable physical instant satisfying the virtual deadline.

For a relative `sleep`(D), the implementation uniformly maps the request to an absolute wait using `sleep_until`(D_v), where the deadline is initialized to $D_v = T_v(t_0) + D$. By the preceding logic, the unblocking instant $t^\star$ guarantees $T_v(t^\star) \geq D_v$. Substituting D_v, we conclude that $T_v(t^\star) - T_v(t_0) \geq D$. Thus, relative sleep durations are preserved across arbitrary debugger pauses. □

Guarantees and Scope. The proof of these two invariants substantiates the mathematical soundness of the pause-erased time semantics. Because arbitrarily complex timeout and rate-limiting schemes ultimately decompose into these elemental time-reads and blocking sleeps, the shim layer averts any artificial failure cascades induced by debugger pauses. System decisions evaluated against PET, such as lease expiration and election deadlines, remain observationally equivalent to an unpaused execution state. This temporal equivalence holds completely up to a minimal under-measurement slack, formally defined in §A.2 and empirically verified in §6.3. The PET abstraction specifically governs local time reads and localized timeout evaluations, making no attempts to artificially synchronize or warp clocks across divergent system processes.

A.2 PET Under-measurement Slack

The Pause-Erased Time mechanism cannot entirely eliminate the physical duration of execution pauses due to unavoidable implementation latencies across the suspension and resumption pathways. Recall from the definitions above that for any debugger-induced pause $G_k = [p_k, r_k)$, the true physical time elapsed during the suspension is exactly the kernel clock difference $\Delta_k^{\text{true}} = T(r_k) - T(p_k)$. To ensure the system never over-estimates the pause duration—which would dangerously cause virtual time to flow backwards—the measurement is taken strictly from within the interval boundaries. Specifically, the internal timestamps $t_{\text{stop}}^{(k)}$ and $t_{\text{resume}}^{(k)}$ are captured exclusively after all debuggee threads are safely paused and exclusively before they are allowed to resume user-space execution.

Because coordinating thread suspension across a process involves asynchronous signals and OS scheduler delays, there exists a strictly non-negative latency at both boundaries. The measured pause duration $\Delta_k = t_{\text{resume}}^{(k)} - t_{\text{stop}}^{(k)}$ is therefore strictly less than or equal to the true pause duration.

Remark 1 (Under-measurement slack)**.** *Let η_k denote the under-measurement slack for a pause interval G_k, defined mathematically as the unmeasured residual duration:*

$$\eta_k \stackrel{\text{def}}{=} \big(T(r_k) - T(p_k)\big) - \Delta_k \geq 0.$$

Let $T_v^{ideal}(t)$ formally represent a hypothetically perfect virtual time that completely erases the true physical duration of every pause without any latency:

$$T_v^{ideal}(t) \stackrel{\text{def}}{=} T(t) - \sum_{k:\, r_k \leq t} \big(T(r_k) - T(p_k)\big).$$

For any physical time t, the actual virtual time $T_v(t)$ governed by PET algebraically relates to the theoretically ideal virtual time as follows:

$$T_v(t) = T_v^{ideal}(t) + \sum_{k:\, r_k \leq t} \eta_k.$$

This mathematically demonstrates that the actual virtual time T_v perpetually remains (weakly) ahead of the ideal debug time by exactly the global cumulative slack $\sum \eta_k$. Furthermore, if the OS-level pause and resume measurement latencies are empirically bounded by maximum delays σ_{pause} and σ_{resume}, the per-pause slack is strictly bounded by $\eta_k \leq \sigma_{pause} + \sigma_{resume}$.

A.3 Wall Clock as the Clock-Source

For APIs that use the `REALTIME` (wall clock) as the clock-source, Invariant 1 (Monotonicity) inherently does not apply, as NTP synchronization can abruptly step the physical wall clock backwards or forwards. Crucially, this is a fundamental property of the OS wall clock, and handling such organic time jumps is strictly an application-level responsibility. PET's formal correctness therefore remains intact: Invariant 2 (Timer Correctness) holds. The shim layer guarantees that the virtualized wall clock yields an observationally equivalent unpaused execution, preserving the natural slews and steps of the physical wall clock while omitting the debugger pauses.

We reuse the established mathematical notation from the preceding proofs: continuous absolute physical time t, the measured cumulative pause offset $O(t) = \sum_{r_k \le t} \Delta_k$, and the debugger pause intervals $G_k = [p_k, r_k)$. Let $R(t)$ denote the value of the kernel's wall clock at any given physical time t. We define the virtualized wall clock value returned to the application as:

$$R_v(t) \stackrel{\text{def}}{=} R(t) - O(t).$$

Here, $R_v(t)$ acts as the true wall clock $R(t)$ observed on a pristine timeline where all recorded debugger pauses are seamlessly subtracted. For any two physical time reads $t_1 < t_2$ (occurring outside of an execution pause), we observe:

$$\begin{aligned} R_v(t_2) - R_v(t_1) &= \big[R(t_2) - R(t_1)\big] - \big[O(t_2) - O(t_1)\big] \\ &= \big[R(t_2) - R(t_1)\big] - \sum_{t_1 < r_k \le t_2} \Delta_k. \end{aligned}$$

This proves that R_v preserves the exact slews and discrete jumps of the underlying physical wall clock $R(t)$, omitting only the measured duration of any debugger-induced pause intervals.

REALTIME API Semantics. The shim layer maps the wall clock time APIs as follows: (i) `get_time_realtime` directly returns $R_v(t)$. (ii) `sleep_until_realtime`(D_v) safely blocks the thread by arming a `REALTIME` absolute physical deadline $D^{\text{RT}}_{\text{real}} = D_v + O(t_{\text{now}})$. Whenever the kernel wakes the thread prematurely—due to the offset O increasing from a pause, a spurious signal, or an NTP-induced backward clock step—the shim layer recalculates $D^{\text{RT}}_{\text{real}}$ and re-arms the sleep until the virtual condition $R_v(t) \ge D_v$ properly holds. (iii) `sleep_realtime`(D) reduces to `sleep_until_realtime`$(R_v(t_{\text{now}}) + D)$.

Invariant 2: Timer Correctness for REALTIME. Time-blocking operations dependent on the wall clock unblock strictly when their virtual wall time constraints are satisfied. Specifically, a call to `sleep_until_realtime`(D_v) returns at an unblocking physical instant $t^\star$ where $R_v(t^\star) \ge D_v$. Consequently, a relative `sleep_realtime`(D) issued at physical time t_0 returns at an instant $t^\star$ guaranteeing $R_v(t^\star) - R_v(t_0) \ge D$.

Proof. By construction, the shim layer yields control back to the application only when it verifies $R_v(t^\star) \ge D_v$, which occurs strictly when the underlying kernel reports $R(t^\star) \ge D^{\text{RT}}_{\text{real}}$ using the most recently calculated deadline $D^{\text{RT}}_{\text{real}} = D_v + O(t^\star)$. If the physical wall clock $R(t)$ steps backwards due to NTP synchronization, the virtual wall clock $R_v(t)$ accurately reflects this backward jump. Because the shim intercepts every kernel wakeup and re-evaluates the inequality $R_v(t) \ge D_v$, backward steps of $R(t)$ prolong the suspension exactly as they would in an un-debugged execution. The re-arming loop masks both debugger pauses (which abruptly increase $O(t)$) and spurious interrupts, guaranteeing the earliest legitimate release time. Similarly, because `sleep_realtime`(D) expands to an absolute deadline $D_v = R_v(t_0) + D$, the release at $t^\star$ enforces $R_v(t^\star) \ge R_v(t_0) + D$, preserving relative wall clock durations despite temporal discontinuities. □

Algorithm 1: Distributed stack stitching.

```
input  : thread_id
output : stitched call stack across processes
DistributedBacktrace(thread_id)
while true do
    stacks ← FetchStack(thread_id);
    meta ← ScanMetadata(stack);
    if meta is invalid then
        return StitchAllStacks(stacks);
    else
        caller ← GetCaller(meta["caller_id"]);
        RestoreStack(caller, meta["caller_context"]);
    thread_id ← caller.thread_id;
```

A.4 vDSO Handling

Linux's vDSO (virtual Dynamic Shared Object) is a kernel-provided shared library mapped directly into a process's user-space memory, exposing critical kernel services without the overhead of a system call context switch. Most modern `libc` implementations optimize time-reading APIs (*e.g.,* `gettimeofday` and `clock_gettime`) by internally routing them through the vDSO.

Our implementation supports these vDSO optimizations without undermining their performance benefits. The shim layer interposes itself strictly between the user application and the underlying `libc`. When an application invokes a time API, the symbol is intercepted by the shim. The shim then queries the actual `libc` function—which uses the vDSO—to retrieve the true physical time, and applies the offset adjustments. Consequently, our design inherits the zero-syscall performance of vDSO while enforcing the Pause-Erased Time abstraction.

We acknowledge a structural limitation: if an application bypasses `libc` to invoke function pointers directly from the vDSO memory mapping (or uses raw `syscall` assembly), our `LD_PRELOAD` shim cannot intercept the observation. Such raw-level resolution is rare in application code, but it does break the shim's enforcement scope. However, because PET's semantics are defined by the invariants above rather than by the interception mechanism, the same logic can be moved into deeper interception frameworks (*e.g.,* eBPF or hypervisor trapping) if required.

B Distributed Backtrace

B.1 Distributed Stack Stitching

As described in Algorithm 1, DDB reconstructs the global call graph by iteratively tracing the causal chain backward across network boundaries. Specifically, DDB: (1) fetches

the local call stack of the targeted thread and appends it to the stitched array; (2) unwinds the stack to locate the injected RPC frame and parses the embedded causal metadata; (3) terminates the local trace if no valid metadata is found, concluding that the root of the call chain has been reached; (4) utilizes the extracted metadata to identify the upstream caller process (which may reside on a remote node) and restores the caller's exact register-level execution context; (5) transitions the backtrace focus to the restored upstream thread and recursively repeats the entire trace process until the origin is fully resolved.

B.2 Stack Stitching Latency at Scale

Because the distributed backtrace algorithm recursively sweeps the call stack backwards across RPC boundaries, its time complexity is inherently linear with respect to the depth of the distributed call chain. Consequently, resolving full traces for deeply nested call chains incurs proportional end-to-end cross-RPC latency. A viable optimization is to propagate N consecutive ancestor metadata records within each RPC packet, rather than just the immediate parent. DDB can then dispatch concurrent stitching queries to multiple ancestor nodes in the chain, masking the sequential round-trip delays. This optimization trades off against a modest increase in the RPC payload size during routine application execution.

C Implementation Details

Framework Integration. Integrating DDB into existing distributed frameworks requires minimal engineering effort. For high-level orchestrators like Nu, Quicksand, and ServiceWeaver, DDB's instrumentation is embedded entirely within the framework's underlying runtime, achieving complete transparency for end-user application code. For standard RPC libraries like gRPC, minor boilerplate adjustments during service initialization are sufficient to propagate discovery metadata.

Metadata Extraction Barrier. During an RPC invocation, DDB injects causal metadata into a local stack variable to mark the network boundary. Because this variable is read only asynchronously, by an external debugger process at runtime, optimizing compilers treat the assignment as dead code and elide it during compilation. To circumvent this, we insert a `volatile` inline assembly directive as an optimization barrier, forcing the compiler to materialize and preserve the metadata on the physical stack.

Wait-for-Attach. To permit deterministic debugging of early startup routines, DDB provides a wait-for-attach mechanism. When enabled via the `DConnector` module, the application explicitly blocks its initialization path and waits for an external attachment signal. We repurpose the POSIX real-time signal `SIGRTMIN+6` (`SIG40`) as a dedicated wakeup trap. Once the debugging session is established, the central control plane instructs the local `DKnot` daemon to inject `SIG40` into the debuggee, resuming its execution.

Multi-ABI Stack Restoration. Properly stitching causal call stacks dynamically requires rewriting explicit thread hardware registers, strictly governed by the target's Application Binary Interface (ABI). For Go on `x86_64`, overwriting the stack pointer (`RSP`) and instruction pointer (`RIP`) is natively sufficient. In contrast, for C++, the C ABI mandates additionally restoring the frame base pointer (`RBP`). Architectural boundaries present similar strict register requirements; ARM64 (`aarch64`) deployments intrinsically require precise restoration of the Link Register (`lr`) alongside `sp` and `pc` to guarantee that subsequent unwinding operations successfully chain the call stack.

Auxiliary Thread for Expression Evaluation. To inspect complex application states, developers routinely evaluate in-guest functions. Dynamically executing arbitrary application logic inherently necessitates harnessing a live thread context. To prevent side effects or accidental state corruption in the primary application execution, we spawn an isolated, no-op auxiliary thread within each debuggee process dedicated to expression evaluation. During interactive evaluations, `DKnot` resumes only this auxiliary thread while all primary application threads remain frozen.

Debugger with Kubernetes. Modern production container images are typically minimized (distroless) to reduce attack surfaces, making it impractical to bundle debugger binaries alongside microservices. For our ServiceWeaver evaluation over Kubernetes, DDB injects its debugger agents into targeted, running pods using Kubernetes Ephemeral Containers [5]. Because Kubernetes overlay networks are isolated, we deploy a gateway container that exposes a single multiplexed port to the DDB control plane, bridging remote SSH tunnels into the ephemeral debugging containers attached to the target microservices.

D Extended Measurement of Command Handling Latencies

To complement the primary evaluation, we present the full measurements of DDB's command handling latencies across varying deployment scales (ServiceWeaver's `socialnet` app). Specifically, Table 7, Table 8, and Table 9 detail the granular end-to-end response times measured under settings of 38, 62, and 122 concurrent microservice instances.

As illustrated in the results, state-inspection commands are automatically emitted in bulk by the IDE frontend (*e.g.,* VS Code) to dynamically instantiate and fetch local variable scopes for UI rendering. Examples include `stack-list-variables`, `var-create`, `var-list-children`, and `var-update`. Conversely, the `exec-interrupt` command operates as an asynchronous control signal, issued exclusively when a developer manually

Command	Latency at the Scale of 38 Processes (ms)				
	Count	Mean	Median	P95	P99
dbt	2,348	32.5	14.2	128.5	153.9
continue	37	2.2	2.0	3.0	4.2
break-delete	1	1.8	1.8	1.8	1.8
break-insert	1	64.4	64.4	64.4	64.4
exec-interrupt	4	8.1	7.3	7.8	7.8
stack-list-variables	66	3.1	2.6	4.6	12.8
var-create	146	24.4	7.4	48.5	51.2
var-list-children	3	3.0	3.0	3.0	3.0
var-update	11	19.2	4.6	47.6	47.6

Table 7. Measured latencies of commands at the scale of 38 processes.

Command	Latency at the Scale of 62 Processes (ms)				
	Count	Mean	Median	P95	P99
dbt	4,303	32.1	13.9	144.5	156.3
continue	8	2.8	2.7	3.6	3.6
break-delete	1	6.9	6.9	6.9	6.9
break-insert	2	46.9	46.9	8.5	8.5
exec-interrupt	4	15.4	16.1	17.3	17.3
stack-list-variables	28	6.1	3.8	12.9	30.3
var-create	78	24.6	16.3	49.6	50.1
var-list-children	3	5.6	3.8	3.8	3.8

Table 8. Measured latencies of commands at the scale of 62 processes.

Command	Latency at the Scale of 122 Processes (ms)				
	Count	Mean	Median	P95	P99
dbt	14,573	26.5	13.3	110.1	160.1
continue	7	4.7	4.8	4.9	4.9
break-insert	1	81.1	81.1	81.1	81.1
exec-interrupt	6	26.1	25.6	29.0	29.0
stack-list-variables	19	7.6	4.4	15.7	15.7
var-create	43	24.4	18.8	49.8	69.9

Table 9. Measured latencies of commands at the scale of 122 processes.

asserts a global pause sequence via the debugger control panel.

E Migration Continuity (Nu and Quicksand)

DDB's design guarantees extensibility to dynamic microsecond-scale environments like Nu [73] and Quicksand [72]. Specifically, DDB preserves two forms of debugging continuity: (i) when a *computation migrates*, all established breakpoint intents automatically follow the execution context; and (ii) when *heap state migrates*, the distributed backtrace reconstructs the global call graph with all caller-visible local variables intact.

Architectural Assumptions. To orchestrate heap continuity, DDB leverages two core architectural properties natively provided by these advanced execution frameworks: (i) *Symmetric Address-Space Layouts.* The underlying frameworks deploy an identical, universally linked binary across all cluster nodes with Address Space Layout Randomization (ASLR) strictly disabled. Consequently, virtual memory addresses and object layouts are guaranteed identical cluster-wide. This implies that DDB's inspection-time restoration can safely map distributed objects directly to the same virtual addresses without requiring complex pointer swizzling or relocation maps. (ii) *Logical Heap Locator API.* The frameworks expose a queryable API that, given a stable logical context identifier, returns the specific remote host and process currently owning a migrating heap region. DDB invokes this locator exclusively on-demand during a distributed backtrace.

Breakpoints Follow Computations. Environments like Nu and Quicksand enforce that any specific logical computation unit is resident on exactly one host at any given time. DDB decouples breakpoints from physical memory addresses, instead expressing them as logical *intents* broadcast to all attached `DKnot` daemons. As the orchestrator spawns new processes to handle migrating workloads, DDB's service discovery captures them so they inherit the global breakpoint configurations. Consequently, whichever physical process currently hosts the migrating computation reports the hit, and DDB focuses the debugger on that entrypoint.

Reconstructing Stack Frames Post-Migration. Because Nu and Quicksand pin the originating sender thread on its native process while its outbound RPC remains outstanding, the stack frames remain sound even if surrounding threads or heaps relocate. However, to reveal on-heap objects reached from a caller frame whose heap has already migrated, DDB executes a sequence of *on-demand heap restorations* during each traceback hop:

1. **Presence Verification.** For every reconstructed caller frame, DDB queries the local runtime to verify whether the caller's target heap remains physically present in the local address space.
2. **Surgical Heap Restoration.** If the heap is absent, DDB queries the framework's locator API to pinpoint the current physical owner. It then coordinates a remote dump of the latest synchronized heap state and writes these bytes into the reconstructed caller's address space at the same virtual addresses, rendering all pointers and local variables valid for live IDE inspection.
3. **Ephemeral Cleanup.** Upon intercepting a `continue` command, DDB removes any temporary restorations, reverting process memory to its original state before execution resumes.