PERFORMAL: Formal Verification of Latency Properties for Distributed Systems

TONY NUDA ZHANG*, University of Michigan, USA
UPAMANYU SHARMA*, MIT CSAIL, USA
MANOS KAPRITSOS, University of Michigan, USA

Understanding and debugging the performance of distributed systems is a notoriously hard task, but a critical one. Traditional techniques like logging, tracing, and benchmarking represent a best-effort way to find performance bugs, but they either require a full deployment to be effective or can only find bugs after they manifest. Even with such techniques in place, real deployments often exhibit performance bugs that cause unwanted behavior.

In this paper, we present PERFORMAL, a novel methodology that leverages the recent advances in formal verification to provide rigorous latency guarantees for real, complex distributed systems. The task is not an easy one: it requires carefully decoupling the formal proofs from the execution environment, formally defining latency properties, and proving them on real, distributed implementations. We used PERFORMAL to prove rigorous upper bounds for the latency of three applications: a distributed lock, ZooKeeper and a MultiPaxos-based State Machine Replication system. Our experimental evaluation shows that these bounds are a good proxy for the behavior of the deployed system and can be used to identify performance bugs in real-world systems.

CCS Concepts: • **Software and its engineering → Formal software verification**.

Additional Key Words and Phrases: distributed systems, systems verification, latency, performance

ACM Reference Format:
Tony Nuda Zhang, Upamanyu Sharma, and Manos Kapritsos. 2023. PERFORMAL: Formal Verification of Latency Properties for Distributed Systems. *Proc. ACM Program. Lang.* 7, PLDI, Article 121 (June 2023), 26 pages. https://doi.org/10.1145/3591235

1 INTRODUCTION

This paper presents PERFORMAL, an approach that extends formal verification beyond the confines of proving correctness—to reasoning about the *performance* of distributed systems.

Understanding the performance of distributed systems is a top priority for most companies. Amazon loses 1% of sales for every additional 100 ms in latency, while brokers can lose up to $4 million per millisecond if their platform is 5 ms behind the competition [Einav 2019]. Similarly, a 2017 Akamai study showed that a 100 ms delay can hurt conversion rates—i.e. how many customers complete a transaction—by up to 7% [Akamai 2017].

To meet end-to-end targets, companies pour significant resources into maintaining the performance requirements of each component in a distributed system. These are typically expressed as

*Both authors contributed equally to this research.

Authors' addresses: Tony Nuda Zhang, nudzhang@umich.edu, University of Michigan, Ann Arbor, Michigan, USA; Upamanyu Sharma, upamanyu@mit.edu, MIT CSAIL, Cambridge, Massachusetts, USA; Manos Kapritsos, manosk@umich.edu, University of Michigan, Ann Arbor, Michigan, USA.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2023 Copyright held by the owner/author(s).
2475-1421/2023/6-ART121
https://doi.org/10.1145/3591235

Service Level Objectives (SLOs). Setting and adhering to realistic SLOs is the bedrock of a smooth user experience.

Despite this effort, performance failures are prevalent in real-world distributed systems. An extensive study of over 3500 vital issues in cloud deployments showed that 22% of the issues relate to performance bugs [Gunawi et al. 2014]. For example, ZOOKEEPER-1465 is a critical-priority bug in the startup of the ZooKeeper coordination service [Hunt et al. 2010] that causes a prolonged period of unavailability—minutes instead of seconds—following a leader failure or when starting up the cluster [Jira 2012]. In another example, DynamoDB [Sivasubramanian 2012] became unavailable as its storage servers exceeded the pre-configured time limit for retrieving membership information after a network disruption [DynamoDB 2015]. Both these issues stem from developers misunderstanding how various aspects of the system affect its performance.

This dissonance between expectation and reality arises from the inadequacy of current best-effort techniques which developers use to understand the performance of their systems. The most common approach is to monitor a live deployment and use tracing techniques to identify the root cause when something goes wrong [Barham et al. 2004; Sigelman et al. 2010; Wu et al. 2019]. Unfortunately, this can only find bugs *after* they occur. The second approach is to benchmark the end-to-end performance of the system before it enters production [Cooper et al. 2010; Denaro et al. 2004]. Just like traditional testing, though, it is virtually impossible to expose *all* undesirable behaviors, no matter how thorough the benchmarking.

This paper presents Performal, a framework that allows developers to establish formal latency guarantees about their distributed system. Performal uses a combination of formal verification and isolated performance measurements of individual components, to produce rigorous bounds for the system's *end-to-end execution time* for single requests.

Reasoning formally about the duration of distributed executions may at first seem hopeless. After all, there are components of the end-to-end execution time (e.g. network latency, garbage collection) for which we cannot provide rigorous bounds. And yet, hope remains. The main contribution of Performal is a new abstraction that lets us reason about the duration of distributed executions without being tied to the performance characteristics of the environment in which they execute. Our approach is inspired by Edsger Dijkstra's quote: *The purpose of abstracting is not to be vague, but to create a new semantic level in which one can be absolutely precise* [Dijkstra 1972].

Performal uses a two-tier approach: the upper tier introduces an abstraction we call *symbolic latency*, which allows developers to establish rigorous bounds on the worst-case runtime of the system. This abstraction expresses a distributed execution as a function of the components involved in the execution, without assuming anything about the real time properties of those components. For example, the duration of a simple execution where node sends a k-byte message to another node can be expressed as $sendTime(k) + msgDelay(k) + receiveTime(k)$. Symbolic latency *decouples* the high-level reasoning about the duration of distributed executions from any assumptions about environmental factors—e.g. the real time duration of message delays, local computations, operating system scheduling, disk writes, etc.

The lower tier of Performal embraces the inexactness inherent in the environment and concretizes these symbolic behaviors into real time estimates. It takes into account the measured latency distributions of individual components and uses them to convert the symbolic latency bounds of the first tier into estimated distributions of the worst-case runtime of the system.

We have used the Performal framework to produce the first proofs of *end-to-end latency* of distributed implementations. Specifically, we proved symbolic latency bounds for three applications: a distributed lock; the feature-rich, MultiPaxos-based State Machine Replication system of IronFleet [Hawblitzel et al. 2015, 2017]; and the ZooKeeper coordination service [Hunt et al. 2010]. We also converted two of those bounds to real time distributions, which our evaluation shows are

good estimates of the actual latency distributions observed when running the systems. We further show how a developer can use PERFORMAL's symbolic bounds to identify the ZOOKEEPER-1465 bug in ZooKeeper.

In short, this paper makes the following contributions:

- It introduces *symbolic latency*, a novel abstraction that decouples the environment from the behavior of the distributed system. This allows us to prove rigorous bounds on the worst-case duration of distributed executions (Section 2).
- It introduces a formal model for specifying symbolic latency properties and a methodology for proving such properties on complex, distributed implementations.
- It presents an approach for combining symbolic latency guarantees and individual component measurements to produce estimated distributions of the worst-case real time duration of distributed executions.
- It evaluates the PERFORMAL framework by proving latency properties for three distributed implementations. Our evaluation shows that the bounds produced by our formal proofs are a good proxy for the behavior of these systems and can be used to identify real-world performance bugs.

The rest of this paper is structured as follows. Section 2 gives an overview of PERFORMAL and showcases an end-to-end example. Section 3 and Section 4 describe how to specify and prove symbolic latency properties. Section 5 describes how to convert symbolic latency guarantees to real time estimates. Section 6 evaluates PERFORMAL in terms of proof effort and the practicality of its proven bounds. Finally, Section 7 discusses related work and Section 8 concludes the paper.

2 OVERVIEW OF PERFORMAL

PERFORMAL is a two-tier framework that uses formal reasoning to produce an upper bound for the worst-case duration of a distributed execution, and then uses measurements of individual components to convert this upper bound to a concrete real time distribution.

2.1 Tier 1: Symbolic latency

Central to PERFORMAL is the notion of *symbolic latency*, which decouples the behavior of a distributed system from the environment in which it is executed—e.g. the Linux kernel, the network, an Intel processor, etc. Since we cannot formally prove timing bounds for the environment, this decoupling is essential for rigorous reasoning about the duration of our executions; a rigor untarnished by specific assumptions about how long messages take to be delivered or how long instructions take to be executed.

Symbolic latency measures the *duration of an execution* in an abstract manner: not in units of real time, but as a function of certain operations, such as a message delay, a disk write, or a local function execution. Consider, for example, a simple execution where a node A sends a k-byte message to another node B. We can represent the duration of this distributed execution with the symbolic latency expression $sendTime(k) + msgDelay(k) + receiveTime(k)$.

Here, *sendTime* and *receiveTime* express the time it takes to execute the `send` procedure on node A and the `receive` procedure on node B respectively, while *msgDelay* is the time it takes for the message to be delivered to node B. In this example, *msgDelay* encompasses the time the message spends in the send and receive buffer of both nodes, though one could refine this symbolic step into multiple steps, where each of these constituents are represented separately.

Section 3 defines the formal model that we use to reason about the latency of a distributed execution, while Section 4 describes how we use this model to prove symbolic latency bounds

for three distributed applications: the distributed lock and State Machine Replication libraries introduced in IronFleet [Hawblitzel et al. 2015], and ZooKeeper [Hunt et al. 2010].

2.2 Tier 2: Converting to a Real Time Distribution

Given a symbolic latency bound, we can convert it to a real time representation, as a function of the real time distribution of the individual components represented in the symbolic bound. In the example above, for instance, our end-to-end real time distribution would take into account the distributions of the time it takes to perform each individual action—i.e., the distributions of $sendTime(k)$, $msgDelay(k)$, and $receiveTime(k)$.

In other words, *a symbolic latency bound is a formally derived recipe for combining the latency distributions of individual components into an end-to-end real time distribution.* Section 5 discusses how we acquire the individual component distributions, model them as independent random variables, and combine them into an end-to-end distribution that is an upper bound for the worst case behavior of the system. It also discusses the subtlety of accounting for identical steps happening in parallel, which Performal requires the user to do manually when working with distributions.

2.3 End-to-end Example: Primary/Backup

As an overview of Performal in action, we demonstrate how we use Performal to prove a bound on the latency of a primary/backup distributed system.

The system comprises a primary server and several backup servers. First, the primary server broadcasts a `Request` message to the backups. When a backup server receives a `Request` message, it processes the request locally and then sends an `Acknowledge` response to the primary. The primary waits to receive `Acknowledge` messages from all of the backups, before sending a `Reply` message to an external client. The sending of the message to the client marks the end of the execution.

Our approach. We model the system as a state machine, consisting of the state of each node and of the network. To reason about latency, Performal augments the state of each node with a *timestamp* that represents how long it took, since the beginning of the system's execution, for that node to reach its current state. When a node takes a step, its timestamp is increased by *Step*, where *Step* is a symbolic latency term representing the duration it takes the step to run.

The top-level performance guarantee we prove says that for any execution of the state machine, any `Reply` message is sent by time $PrimaryRequest + BackupRequest + PrimaryAck + 2 \cdot Queuing + 2 \cdot d$ since the start of the execution. Here, d is the message delay and *Queuing* is a symbolic term representing the queuing delay experienced by a message (how long it sits in the destination's network stack before the application starts processing it). Also, *PrimaryRequest* is the step on the primary in which `Request` messages are sent to the backups, *BackupRequest* is the step on a backup which processes a `Request` message, and *PrimaryAck* is the step on the primary which processes an `Acknowledge` message. Figure 1 visualizes how the execution leads to this bound.

Proving the symbolic latency guarantee. The proof of a latency guarantee, like other proofs of safety, is by means of an inductive invariant that implies the guarantee. For the primary/backup example, the invariant is derived as follows.

We assume that the primary starts running at time 0 and that its first step is to broadcast a request to all the backups. Hence, our inductive invariant first says that the `Request` messages must be delivered by time at most $PrimaryRequest + d$.

Next, the invariant says that any `Acknowledge` message sent from a backup to the primary is delivered by time $PrimaryRequest + d + \mathbf{Queuing + BackupRequest + d}$. The bolded portions are the terms in the symbolic latency expression that have been added since the previous step. The term *Queuing* captures the fact that the `Request` message arriving at the destination node may be

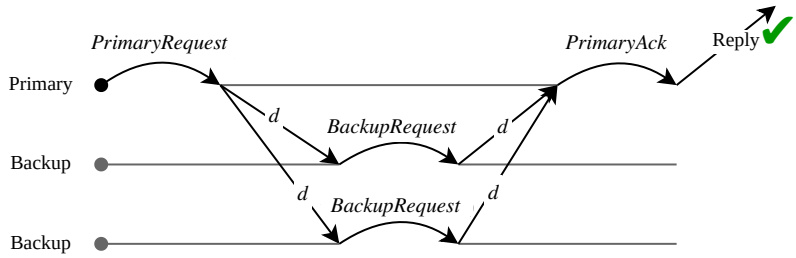

Fig. 1. Diagram showing an execution of the primary/backup system, with two backup servers and one primary. For clarity, we omit the *Queuing* delay in this picture.

queued in the node's network stack for some time before the application program calls `Receive` to retrieve it. *BackupRequest* is the time the backup takes to process the message, after which it immediately sends an `Acknowledge` message that takes at most d more time to reach the primary.

Finally, the invariant maintains that any `Reply` message must be sent by time *PrimaryRequest* + d + *Queuing* + *BackupRequest* + d+ ***Queuing*** + ***PrimaryAck***. This is proved just like the bound on the delivery time of `Acknowledge` messages. Altogether, this invariant is inductive (it is maintained by any step of the system) and implies the performance guarantee.

Converting the symbolic latency bound to a real time bound. After identifying and proving a symbolic latency bound, we can convert that symbolic expression into a real time interpretation by combining it with measurements of our specific environment's low-level runtimes. In particular, we measure the statistical distribution of the message delay d and the component runtimes such as the duration of the function *PrimaryReq*, and combine the individual component runtimes as specified by the symbolic latency expression. We do this by treating each term in the expression as an independent random variable, and applying the standard formulas of probability theory to compute the distribution of the sum of the random variables.

The final result is an estimated upper-bound distribution for the system's runtime—e.g. the 99^{th} percentile of the distribution gives an estimated upper bound for the system's p99 latency. Section 6 includes concrete examples of real-time distributions produced using this methodology and shows that these distributions closely track the latency of the deployed systems.

One subtlety in computing real time distributions is accounting for parallel transitions. For instance, in Figure 1, the two backups run *BackupRequest* in parallel, and the primary waits for both to complete. Failing to account for these parallel transitions can result in an incorrectly skewed distribution. To avoid this, PERFORMAL requires the developer to manually annotate that the *BackupRequest* term in the symbolic latency bound actually corresponds to two parallel *BackupRequest* steps. The final distribution then uses the distribution $\max(\mathit{BackupRequest}_1, \mathit{BackupRequest}_2)$, where the two are independently distributed.

Note that instead of statistical measurements, one can, in principle, use the large body of existing work [Bernat et al. 2003; Blackham et al. 2011; Cazorla et al. 2013; Hansen et al. 2009; Li et al. 2007; Muller and Hoffmann 2019; Sewell et al. 2017] on finding the worst-case execution time (WCET) of a program to derive rigorous bounds on local computations. These techniques, however, are mostly targeted to real-time environments and cannot be applied to off-the-shelf CPUs and network stacks. The vast majority of developers write code for commodity, non-real-time environments and

would benefit from a more general—albeit less rigorous—way to convert their symbolic bounds into real-time bounds.

3 SPECIFYING LATENCY PROPERTIES

In this section, we present a formal model for reasoning about the duration of distributed executions using symbolic latency. This model must be sufficiently expressive for the analysis of real, complex implementations. For example, a count of sequential message delays is inadequate, since it would not capture the duration of local computations and possible queuing delays. Likewise, so is counting the number of state transitions taken by individual nodes because different steps don't necessarily take the same amount of real time. Ultimately, the model must help developers answer questions such as: *What is the end-to-end latency of a client request in a state machine replication system? How long will my system take to recover from a failure?*

Our model builds on top of prior work on the verification of distributed system implementations, namely IronFleet [Hawblitzel et al. 2015, 2017]. In IronFleet, a distributed system is modeled as a state machine that comprises a set of nodes and a collection of network messages. Nodes are implemented via a main-event loop that runs event-handler methods. Every node is modeled as a state machine whose transitions map directly to the event-handler methods. The distributed system state machine takes a transition when one of the individual nodes takes a transition on its local state machine, possibly also updating the network state.

In Performal we extend IronFleet's state machine with explicit support for reasoning about the timing of various events. We start by describing our ***standard*** model for latency in Sections 3.1–3.4. The standard model captures the most common elements involved in a distributed execution, such as message delays, timeouts and function executions. This model is simple enough to allow for low-effort proofs and yet it is expressive enough to reflect the latency characteristics that most developers care about.

We also show how the standard model can be refined to incorporate fine-grained reasoning about queuing delays that may arise as a node processes multiple incoming messages. This ***advanced*** model, presented in Section 3.5, yields a symbolic bound which expresses the system's latency at an even higher granularity, but at the cost of increased proof complexity.

3.1 Symbolic Timestamps

We begin by extending the distributed system state machine with a *timestamp* for each "object" in the system, namely each node in the cluster and each message in the network. In the initial state (i.e., the moment the distributed execution begins) all timestamps are set to 0. During the execution, a node's timestamp is a symbolic latency expression representing the time it took for the node to reach its current state, counted from the start of the system's execution. Meanwhile, a message's timestamp captures the delivery time of the message to its destination. Importantly, timestamps are a *ghost* construct—i.e. they exist only in proof code and are not present in the compiled executable.

Formally, we define a symbolic latency expression E as the sum or max of two other expressions, or an atomic value e representing a component runtime, or an additive identity 0.

$$E \rightarrow E_1 + E_2 \mid \max(E_1, E_2) \mid e \mid 0$$

Symbolic latency expressions also obey a partial ordering $\leq$. We use this ordering to reduce max-expressions such that given $E_1 \leq E_2$, we reduce $\max(E_1, E_2)$ to E_2.

Intuitively, one can think of these expressions simply as undetermined real-valued variables serving as placeholders for component runtimes. In Section 5 we discuss the subtleties of treating them as statistical distributions of random variables.

3.2 System and Properties

At the core of PERFORMAL are temporal mapping functions τ_N and τ_M that respectively map node names and messages to symbolic time expressions E,

$$\tau_N : N \to E \qquad \tau_M : M \to E$$

where N is the set of unique node names in the system, and M is the set of all sent messages in the network. The mappings represent the timestamps of each node and message.

We then define the global state of the system as

$$\mathcal{G} := (\Sigma, M, \tau_N, \tau_M)$$

where Σ is a function that maps node names to their local state. In the initial state, τ_N maps each node to 0, and M is empty.

A *symbolic latency property* ϕ is then a state predicate expressed in terms of Σ, τ_N, τ_M. For instance, in the primary-backup example described above, an example of a symbolic latency property is

$$\begin{aligned}\texttt{reply\_latency}(\Sigma, M, \tau_N, \tau_M) := \forall m \in M, \texttt{is\_reply}(m) \implies \\ \tau_M(m) \leq \mathit{PrimaryRequest} + d + \mathit{Queuing} \\ + \mathit{BackupRequest} + d + \mathit{Queuing} + \mathit{PrimaryAck}\end{aligned}$$

which says that for all reply messages m in the given global state, its timestamp $\tau_M(m)$ is less than the stated symbolic expression. We say that property ϕ is an *inductive invariant* of the system if it holds in the initial state of the system $\mathcal{G}_0$, and, given any state $\mathcal{G}$ satisfying ϕ that steps to $\mathcal{G}'$, then $\mathcal{G}'$ also satisfies ϕ.

3.3 Updating Timestamps

Next, we present a set of rules that govern the evolution of timestamps as the system transitions from one state to the next. These rules aim to capture the semantics of the physical passage of time in a distributed system. Here, we assume that each node's local actions attempt to receive from the network at most once at the beginning of its execution, and send all messages at the end. This is similar to (but stronger than) the reduction obligation used by IronFleet [Hawblitzel et al. 2015].

No-Receive Rule.

$$\frac{\begin{array}{cc} A(n, \Sigma) = (\sigma', M_{\text{out}}) & \Sigma' = \Sigma[n \mapsto \sigma'] \\ \tau'_N = \tau_N[n \mapsto \tau_N(n) + a] & A \text{ does not call } \texttt{receive} \end{array}}{(\Sigma, M, \tau_N, \tau_M) \to (\Sigma', M \uplus M_{\text{out}}, \tau'_N, \tau_M[M_{\text{out}} \mapsto \tau'_N(n) + d])} \quad \text{(NO-RECV)}$$

Consider some local action A that does not receive any messages. When a node n takes step A, it transitions to a new local state σ', and sends a (possibly empty) set of messages M_{out} that is added to the network (using the disjoint union operator $\uplus$). In doing so, the timestamp of n in $\tau'(n)$ increases by the time it takes to execute that procedure, denoted by the symbolic quantity a.

Moreover, the timestamp of each message in M_{out} is assigned the expression $\tau'_N(n) + d$, which represents the time when it will be delivered to its destination. This is d time after n is done executing the procedure A, at the end of which the message is sent. Such an update to τ_M is common across all rules.

Receive-Timeout Rule.

$$\frac{\begin{array}{cc} A(n, \Sigma) = (\sigma', M_{\text{out}}) & \Sigma' = \Sigma[n \mapsto \sigma'] \\ \tau'_N = \tau_N[n \mapsto \tau_N(n) + T_0(u) + a] & A \text{ calls } \texttt{receive} \text{ but times out} \end{array}}{(\Sigma, M, \tau_N, \tau_M) \to (\Sigma', M \uplus M_{\text{out}}, \tau'_N, \tau_M[M_{\text{out}} \mapsto \tau'_N(n) + d])} \quad \text{(RECV-TIMEOUT)}$$

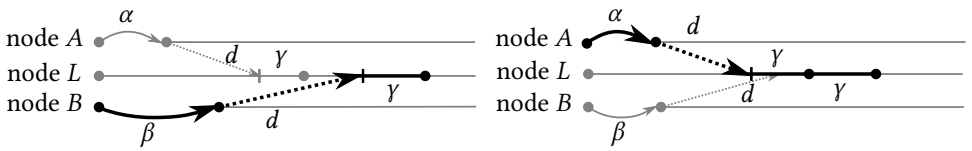

Fig. 2. A network race between messages from two different nodes. Solid lines indicate nodes taking steps and dashed lines indicate a message transmitted over the network. The bold lines indicate the critical path for evaluating runtime. The diagram on the right shows an execution where B's message arrives while L is still processing A's message, and thus experiences queuing delay.

An action A involving `receive` with (possibly zero or infinite) timeout u will keep checking the node's message queue for up to u time for a message to dequeue. If the timeout is finite, one possibility is that `receive` times out, in which case $T_0(u)$ time elapsed, followed by the method's execution. Here, $T_0(u)$ is a symbolic latency value representing the actual time taken on a `receive` timeout. It is the time from when we invoke `receive` to when the `receive` procedure returns.

Receive-Queuing Rule.

$$\frac{\begin{array}{cc} A(n,\Sigma,m) = (\sigma', M_{\text{out}}) & \Sigma' = \Sigma[n \mapsto \sigma'] \\ \tau'_N = \tau_N[n \mapsto \max(\tau_N(n), \tau_M(m)) + a] & \tau_N(n) \le \tau_M(m) + \mathit{Queuing} \quad A \text{ receives } m \end{array}}{(\Sigma, M, \tau_N, \tau_M) \longrightarrow (\Sigma', M \uplus M_{\text{out}}, \tau'_N, \tau_M[M_{\text{out}} \mapsto \tau'_N(n) + d])} \quad \text{(Recv-Q)}$$

If a node n calling `receive` does not time out and returns a message m, there are two cases: (1) m had already arrived at a time $\tau_M(m)$ prior to `receive` being called at $\tau_N(n)$, so `receive` immediately returns m; and (2) node n calling `receive` at time $\tau_N(n)$ finds an empty message queue upon entry and waits until a message m is delivered, returning as soon as m arrives at time $\tau_M(m)$. In both cases, after the message is received, the method handler processes the message. The 'max' operation in the rule captures these two scenarios.

The term *Queuing* represents the time that a message spends in the message queue, between its delivery at a node and when it is returned by a call to `receive`.

Example. To show these rules in action, let us analyze a small system consisting of three nodes A, B, and L, and a network race between messages from A and B to L as illustrated in Figure 2. This situation is similar to the backups in the primary/backup system from Section 2.3 sending a message to the primary; the `Acknowledge` messages from the backups race to arrive at the primary.

We let α (resp. β) be symbolic latency values that denote the runtime of a non-receiving method that sends a message $\texttt{msg}_A$ (resp. $\texttt{msg}_B$) after some local computation from node A (resp. B). Finally, node L has a single action which attempts to receive messages with infinite timeout and then processes a message in time γ. We want to formally analyze the time it takes L to process both messages from A and B.

The behavior of L starts with a transition that receives and processes $\texttt{msg}_A$ followed by a transition that receives and processes $\texttt{msg}_B$. We denote the local timestamps of L in its sequence of states in this behavior as (L_0, L_1, L_2). The (No-Recv) rule states that $\tau_M(\texttt{msg}_A) \le \alpha + d$ and $\tau_M(\texttt{msg}_B) \le \beta + d$. After the first transition, the (Recv-Q) rule gives $L_1 \le \max(L_0, \alpha + d) + \gamma = \alpha + d + \mathit{Queuing} + \gamma$. After the second transition, the rules now yield $L_2 \le \max(L_1, \beta + d) + \gamma = \beta + d + \mathit{Queuing} + \gamma$. In particular, the *Queuing* term captures the possible queuing delay experienced by the second message as illustrated by the right diagram in Figure 2.

3.4 Latency Guarantees

With the above model, one can state symbolic latency guarantees by bounding timestamps of nodes and messages at interesting events. For instance, to prove that a system will reply to a client's request within time T, one can define a predicate on the state of the distributed system that is true if and only if all reply messages in the network have a timestamp bounded by T. The invariant that this predicate holds in all states in all formal timestamped behaviors is a *symbolic latency guarantee*, which provides a time bound for the worst case behavior of the system.

A subtle but noteworthy point is that our latency guarantees are explicitly designed to *not* be a strengthening of liveness properties. A property carrying both liveness and latency obligations would be of the form "the system eventually performs the desired action, and does so within time T". This would require the developer to prove liveness, a task that is notoriously difficult [Hawblitzel et al. 2015]. Instead, our guarantees assert that *if* the desired action ever happens, it will be done within time T. As such, they are safety properties, which are significantly easier to prove. Liveness properties can, of course, still be proven, if one is willing to put in the required effort.

3.5 Refining the Symbolic Model

The model that we present is not set in stone. Our rules can be updated to balance the tradeoff between fidelity and proof effort. Below, we give an example of how they can be—and have been—modified to provide additional fidelity.

3.5.1 Reasoning About Queueing Delays. We consider how one can replace the standard symbolic model presented in Section 3.3 with a more detailed model that explicitly reasons about queuing delay. In reality, the queuing delay experienced by a message m consists of multiple sub-steps, namely the processing of each message that is already in the queue when m arrives. Yet, the (RECV-Q) rule in our standard model simplifies this by hiding the complexity behind a "blanket" variable that represents all queuing delay, as we have found that queuing delay is rarely a major concern for developers trying to understand the latency of their systems, affording us the opportunity to model queuing at a coarse granularity. More importantly, this coarse granularity greatly improves the tractability—and thus, the practicality—of latency proofs.

Yet if a developer wishes to reason about queuing delays with full rigor, they can replace our standard model with the **advanced** model, which supports fine-grain reasoning about queuing at the cost of increased proof complexity. In the advanced model, the (RECV-Q) rule is dropped. In its place, we add a new piece of state `dts` for each node n, represented by the mapping $\tau_{\texttt{dts}}$, and a new (ARRIVAL-TIME) rule.

$$
\begin{aligned}
\tau'_{\texttt{dts}} &= \tau_{\texttt{dts}} && \text{on a non-receiving step,}\\
\tau'_{\texttt{dts}} &= \tau_{\texttt{dts}}[n \mapsto \tau_N(n) + T_0(u)] && \text{on a receive time out, and}\\
\tau'_{\texttt{dts}} &= \tau_{\texttt{dts}}[n \mapsto \tau_M(m)] && \text{on a successful receive of message } m.
\end{aligned}
$$

$$
\frac{\begin{array}{ccc} A(n,\Sigma,m) = (\sigma', M_{\text{out}}) & \Sigma' = \Sigma[n \mapsto \sigma'] & M' = M \uplus M_{\text{out}} \\ \tau'_N = \tau_N[n \mapsto \max(\tau_N(n), \tau_M(m)) + a] & \tau'_{\texttt{dts}} = \tau_{\texttt{dts}}[n \mapsto \tau_M(m)] & A \text{ receives } m \\ \tau_N(n) \le \tau_M(m) & \tau_{\texttt{dts}} \le \tau_M(m) & \end{array}}{(\Sigma, M, \tau_N, \tau_M, \tau_{\texttt{dts}}) \rightarrow (\Sigma', M', \tau'_N, \tau_M[M_{\text{out}} \mapsto \tau'_N(n) + d], \tau'_{\texttt{dts}})}
\quad \text{(ARRIVAL-TIME)}
$$

Intuitively, for each node n, $\tau_{\texttt{dts}}(n)$ records either the time the most-recently processed message was delivered, or the last time the message queue was determined to be empty, whichever of the two is more recent.

Arrival Time Rule. This states that a message received in a step must be delivered after all previously received messages and after the previous `receive` timeout. This holds in a real execution because messages are removed from the queue in the order they are delivered, so a message returned by `receive` must have been delivered after all messages returned by previous calls to `receive`. Moreover, if a previous call to `receive` timed out, and a later call returns a message, it is not possible for that message to have been delivered prior to the time out.

3.5.2 Reasoning About Heterogeneous Networks. In the rules presented above, we use a single term d to represent the delay of every link in the network. Doing so is *sound*, as d represents an upper bound on message delay. I.e., it is safe to treat all links as if they performed like a link slower than each of them. This allows for simpler proofs with fewer variables involved, at the expense of producing not-so-tight symbolic bounds in the case that the network is vastly heterogeneous. While not pursued in this paper, the Performal methodology fully supports parameterizing message delays with their source and destination as $d_{(src,dst)}$, should the developer wish to explicitly model a heterogeneous network.

Altogether, the flexibility of our framework allows developers to use a pay-as-you-go approach. In situations which require capturing a wider variety of real-world events, such as queuing delays, one can enrich the symbolic model to capture these additional behaviors.

4 PROVING LATENCY PROPERTIES

In this section we show how we can use our model to prove rigorous symbolic latency properties for a distributed system implementation. We express these properties in terms of upper bounds on method executions, message delays and timeouts. Our proofs use the Dafny language and verifier [Leino 2010], which in turn uses the Z3 SMT solver [De Moura and Bjørner 2008]. We demonstrate the efficacy of Performal by using it to prove the latency properties of several distributed applications. These include two that were implemented as part of the IronFleet project: a distributed lock system and a feature-rich, MultiPaxos-based State Machine Replication system called IronRSL [Hawblitzel et al. 2015]. We also demonstrate Performal's ability to reason about failures by proving an end-to-end bound for how long it takes to process a request in IronRSL when the leader node crashes.

We first describe the assumptions used by our proofs. We then summarize the latency properties that we proved, before delving into the proofs of IronRSL to illustrate how symbolic latency proofs are carried out.

4.1 Assumptions

Our proofs aim to provide bounds on how long a certain execution will take—e.g. how long it would take for the system to process one request. As such, they focus on the request at hand and all the messages sent by nodes to service that request.

To avoid reasoning about the latency impact of arbitrary queuing delays, our proofs assume that there are no other messages, other than the ones pertaining to the processing of the current request. For the same reason, we also assume that messages are not duplicated in the network.

Indeed, if we relaxed our assumptions and allowed arbitrary external or duplicate messages, we would have to contend with the possibility of a network denial-of-service attack, under which there are no performance guarantees. Of course, a middle ground is also possible, where a certain amount of external or duplicate messages exist, but we have opted against adopting this model for the sake of simpler proofs.

Similarly, our proofs involving node failures assume an upper bound on the number of failures during the execution. Otherwise, there can be no performance guarantees if view changes can happen an unbounded number of times. Most distributed systems also lose liveness if failures exceed a certain threshold.

4.2 Symbolic Latency Properties Proven with Performal

Table 1. Symbolic latency bounds proven using Performal. The applications marked as "advanced" were proved using the advanced model, while the rest were proved using the standard model. In these expressions, n and f are respective parameters for system size and degree of fault tolerance. In the IronRSL bounds, the function $ActionsUpTo(k)$ is the sum of the symbolic latency terms representing actions 1 through k in the enumeration of actions that a node can take.

Application	Performal's symbolic time bound
Distributed lock system (advanced)	$n \cdot (Accept + Grant + d)$
ZooKeeper (advanced)	$SendFollowerInfo + d + ProcessFollowerInfo \cdot 2f + d +$ $ProcessLeaderInfo + d + ProcessEpochAck \cdot 2f +$ $PrepareSync \cdot f +$ **SendSnapshot** $\cdot f + SendNewLeader \cdot f + d$ **ProcessSnapshot** $+ ProcessNewLeader + d + ProcessAck \cdot f$
IronRSL end-to-end latency (no failures)	$4 \cdot d + ProcessMessage + ActionsUpTo(4) +$ $ActionsUpTo(7) + 3 \cdot Queuing$
IronRSL end-to-end latency (with 1 failure)	$2 \cdot ViewTimeout + HeartbeatPeriod + 2 \cdot d + (VariousSteps) +$ $2 \cdot d + ProcessMessage + 2 \cdot Queueing + ActionsUpTo(3) +$ $3 \cdot d + 2 \cdot Queueing + ActionsUpTo(7) +$ $ProcessMessage + MaybeDoPropose$
IronRSL end-to-end latency (advanced, no failures)	$4 \cdot d + (f + 5) \cdot AllActions + ProcessMessage +$ $ActionsUpTo(4) + ActionsUpTo(7)$

Table 1 lists the systems for which we proved symbolic latency properties using Performal. We discuss the simplest one here, the distributed lock, while the IronRSL and ZooKeeper proofs will be presented in more detail later (Sections 4.4 and 6.6).

The distributed lock system consists of a ring of n nodes passing a lock token around the ring. The symbolic time we proved is the maximum time it takes for the lock to traverse the ring exactly once. In the symbolic bound, *Grant* is the runtime of the `Grant` method, which sends the lock to the next node, and *Accept* is the runtime of the `Accept` method, which receives the lock from the previous node in the ring. The symbolic bound we proved is quite intuitive: moving the lock to a new node takes $Accept + Grant + d$, and this happens n times total.

4.3 How to Prove Latency Properties

A symbolic latency property is an invariant on the value of a certain event's symbolic timestamp. For example, one can prove the invariant: "if a client sends a request at time 0 and later receives a response, the symbolic timestamp of the response will be at most T", for some symbolic time expression T.

Our experience with these proofs suggests that the methodology for proving these latency invariants is quite different from proving the invariants of a traditional functional correctness property. In those proofs, one typically starts from the desired property and strengthens it until it becomes inductive.

That approach, however, doesn't work for latency invariants. Here, one typically doesn't initially know what is the precise bound they will prove. Instead, they only know their "target" event—e.g. the client receiving a response—whose symbolic timestamp they need to bound. They can do so by starting at the initial event—e.g. the client sending a request—and proving invariants on the symbolic timestamps of an increasingly wider horizon of events. In a Paxos system, for example, one can start by proving an invariant on when the request will be received by a replica; then use that bound to prove when the request will be proposed; then accepted; then learned, etc., eventually deriving a bound on when the response will arrive at the client.

As such, the process of proving latency properties was for us also a process of *understanding* the performance of a system. This process gave us insights into *why* a system exhibits the performance that it does and what we can do to improve it.

Proof sharding. The best way to visualize a symbolic latency proof is as a line of dominoes. The line starts with some initial event—e.g., the client sending a request. This domino falling leads to a series of other dominoes falling—e.g., the request being received; proposed; accepted, etc.—until the final domino falls when the client receives a response. As we mentioned above, the ultimate goal is to establish an invariant on the symbolic timestamp of the final domino.

This visualization highlights the fact that latency proofs are not monolithic. One can in fact establish individual bounds on how long each domino will take to fall once the previous domino has fallen. We can then compose these individual bounds in an end-to-end bound for the entire execution. This modular approach facilitates the developer's task, as they only have to reason about one part of the system at a time.

Reasoning about failures. Notably, the introduction of crash failures does not require any additional modeling treatment—they are modeled using ordinary state machine transitions. In fact, a crash failure is simply the *absence* of any future transitions by the failed node. The only consequence of such failures on the global system state is that actions are triggered on live nodes when they detect (usually via a timeout mechanism) that some node has failed.

4.4 Case Study: IronRSL

IronRSL is a State Machine Replication library that was implemented as part of the IronFleet project [Hawblitzel et al. 2015, 2017]. At its heart lies a MultiPaxos consensus protocol [Lamport 1998, 2001] that ensures all replicas execute client requests in the same order.

We proved a bound for the end-to-end request latency of the system in the absence of failures and view changes; and in the case where the initial Paxos leader may crash. The resulting symbolic latency expressions (Table 1) are upper bounds for the time from when the client sends a request to when it receives a reply.

Using proof sharding to reduce complexity. Paxos is notoriously complex, especially when it comes to reasoning about its view change mechanism. To tame this complexity, we used our proof sharding technique to split the overall proof into smaller, more manageable sub-proofs. In our end-to-end proof where the initial leader may fail, we separately prove: (1) how long it takes to detect the failure and trigger a view change; (2) how long phase 1 of Paxos—i.e., electing a new leader—lasts; and (3) how long phase 2—i.e., learning a proposed value—lasts. Composing these bounds gives us a bound for when a value will be learned after the client sends a request. At this point, we only have to reason about the final and simplest domino falling: the client receiving the response.

Proving each bound. Just as we split up the full execution into multiple, simpler bounds, the proof for a particular bound also involves considering separate stages of progress in the system.

For instance, consider phase 2 of Paxos, where the leader broadcasts `Propose` messages, and waits for `Accepted` responses. Initially, the system has no `Propose` messages sent. After some local steps, the leader broadcasts `Propose` messages, thereby making progress towards completing phase 2. Then, the leader waits for `Accepted` messages. As soon as it gets a quorum of such messages, it begins taking local steps to execute the client's request and send a reply back to the client.

In the proof, each of these different stages of progress has its own invariant. Starting in one stage, a step might stay in that stage, or might lead into the next one. The ultimate inductive invariant is the disjunction of the invariants for the different stages of progress that the system could be in.

5 CONVERTING TO REAL TIME

Once we have a symbolic latency bound, we can move on to the second tier of PERFORMAL, which allows us to convert this symbolic bound into an estimated real time bound. This is done by first gathering distributions of the time it takes to execute each individual component involved in the symbolic bound, and then using the symbolic expression as a recipe to compose these individual distributions into a real time distribution of how long the end-to-end execution will take.

Note that the result is an upper bound for the worst-case behavior of the system, not the expected behavior. In practice, we find that the real time distributions we extract in our evaluation track the observed behavior of the system well enough to be useful (see Section 6 for more details).

5.1 Collecting Real Time Information

In PERFORMAL, we use measurements for determining individual component runtimes. We run each component in isolation to extract a detailed distribution of its runtime within a given environment. This includes extracting a distribution for message transmission delays. This approach is reminiscent of the hybrid WCET approach, in which measurements of basic blocks are used together with static analysis of control flow to derive the WCET of an entire program [Betts et al. 2010], but at the distributed system level.

5.2 Combining Real Time Distributions

To reason about distributions of runtime, we treat the uninterpreted quantities in symbolic latency expressions as random variables. When we say that our estimated distribution for latency is an upper bound for real runtime, we mean that the distribution exhibits *first-order stochastic dominance* [Hadar and Russell 1969]. A distribution with cumulative density function G is said to first-order stochastically dominate a distribution F if $G(x) \leq F(x)$ for all x. This means that for all x, a sample of G is less likely than a sample of F to be below x. In also means that for any percentile p, the p-th percentile of G upper bounds p-th percentile of F. Thus, finding stochastically dominating distributions will allow us to find bounds on percentile runtimes of the system.

In converting symbolic latency expressions to distributions, we assume that different executions of the same operation, such as different calls to the same method or network delay on different messages, have independent runtime. For instance, the timestamp at a node after two calls to method m should be $r_{m,1} + r_{m,2}$, where $r_{m,1}$ and $r_{m,2}$ represent two independent and identically distributed random variables.

One complication with converting to distributions is knowing when operations happen in parallel. For instance, the distribution of time spent waiting for a single message delay is different from the time spent waiting for 100 parallel message delays. In the latter case, the chances of a single message being very slow are high, and the accurate distribution is $\max(d_1, \ldots, d_{100})$, where each d_i is an independent variable with the same distribution.

The distributed lock system, for example, has a single path of execution. Messages are not sent in parallel and nodes do not have to wait for multiple messages to make progress. In this case,

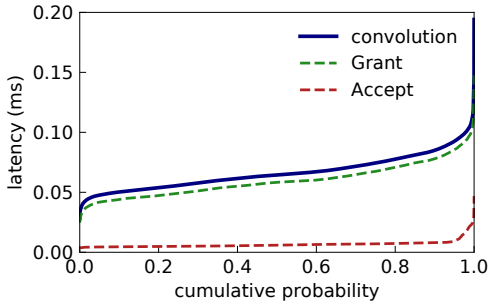

Fig. 3. The convolution shows the computed distribution of a `Grant` followed by `Accept`, which is the result of a convolution of the two underlying probability density functions.

we can simply convert the symbolic latency bound of the distributed lock system into a sum of n independent `Grant`, `Accept`, and message delay distributions.

In Paxos, however, operations sometimes happen in parallel. In the symbolic latency for IronRSL, two instances of d represent parallel message delays, namely the leader broadcasting `Prepare` messages to all acceptors, and then waiting to receive $f + 1$ responses. Hence, we convert the IronRSL bound to real runtime by treating two of the d's in the symbolic latency expression as having the distribution of $\max(d_1, \ldots, d_{f+1})$, where each d_i is an independent network delay.

This last decision—determining which parts of the symbolic latency expressions are executed in parallel—is currently done manually using our knowledge of the protocol.

Computing end-to-end distributions. To reason about distributions of runtime, we have to compute the sum and max of individual operation distributions. Independence allows us to compute the sum of two distributions using a Gaussian convolution [Petters 2007]. Briefly, given two real independent random variables X and Y with probability density functions f_X and f_Y, their sum $Z = X + Y$ has a distribution equal to the convolution $f_Z(z) = \int_{-\infty}^{\infty} f_X(z - t) f_Y(t) dt$. In practice, we do not have complete probability densities for component runtimes, but use the histograms from our measurements as an approximation. We use a discrete convolution of the histograms to compute the sum of independent random variables. For instance, Figure 3 shows the distribution for `Accept`, `Grant`, and for an `Accept` and `Grant` in sequence, of our distributed lock system. Finally, the distribution of $\max(X, Y)$ is computed as $P(\max(X, Y) \leq z) = P(X \leq z) \cdot P(Y \leq z)$ for independent random variables. Using measurements and these primitives, we can convert symbolic latency expressions into distributions of system runtime.

Computing a numerical upper bound. Note that there are other ways to interpret symbolic bounds other than full-blown statistical distributions. In some cases, a simple numerical upper bound may suffice, instead of an entire distribution. In that case, computing an end-to-end upper bound is simple: one can "plug in" the upper bound of each component into the symbolic bound. For instance, if we know that all method executions take at most 10 ms and network delay is at most 30 ms, then the symbolic bound for the distributed lock system with n nodes can be converted to a real time upper bound as $n \cdot (Accept + Grant + d) \leq n(10 + 10 + 30)$ ms $= 50n$ ms. Thus, in an environment with method runtime below 10 ms and network delay below 30 ms, we know that it takes no more than $50n$ ms for the lock to traverse the ring.

5.3 Replacing Measurements with Proofs

So far, our approach for converting a symbolic bound to a real-time bound is one that we consider the most practical: measuring individual components and using the symbolic bound to combine their distributions into one that stochastically dominates the runtime distribution of the system.

It is possible, however, to introduce even more rigor into this conversion by reducing the reliance on empirical measurements. For instance, a rigorous way to estimate a bound on the real time duration of a component is to use worst case execution time (WCET) analysis, which is used in the real time systems community as a foundation for establishing real time guarantees [Blackham et al. 2011; Cazorla et al. 2013; Hansen et al. 2009; Li et al. 2007; Muller and Hoffmann 2019; Sewell et al. 2017]. Of course, this added rigor comes at the cost of effort and generality. WCET analysis requires a hand-crafted, real-time model of the CPU and thus cannot be applied to off-the-shelf environments such as those found in most datacenter deployments, whose CPUs are too complex for such analysis.

6 EVALUATION

Our evaluation demonstrates that PERFORMAL is useful in estimating latency bounds of practical distributed systems. In particular, we aim to answer the following questions:

- Does PERFORMAL's estimated upper bound track the actual latency of a simple system across different workloads and configurations? (Section 6.1)
- Does PERFORMAL's estimated upper bound track the actual latency of a complex, full-feature system? (Section 6.2)
- Does PERFORMAL's estimated upper bound remain accurate across different deployment environments and in the face of node failures? (Section 6.3, 6.4)
- Can PERFORMAL identify performance issues of real distributed systems in production? (Section 6.6)

We address these questions by using PERFORMAL to derive claims about the latency behaviors of a series of implementations. First, we use the distributed lock system to evaluate the soundness of PERFORMAL's latency estimates, and measure the sensitivity of PERFORMAL to various configurations and workloads. We then use IronRSL (Section 4.4) to evaluate PERFORMAL's applicability to a complex, feature-rich distributed system such as a Paxos-based State Machine Replication library. For each system, we use Dafny's Go compiler to convert Dafny source code into Go, and generate executables using a Go compiler.

To predict the system's latency, PERFORMAL requires as input the distributions of each component in the symbolic latency expression. This includes the empirical cumulative distribution function (ECDF) of each local state machine step, and the ECDF of message delays in the network. To measure the execution time of each local step, we instrument the Go source code to record timestamps of each step's entry and exit. To measure network latency, we implemented in Go a Network Agent that runs on each machine. Each Network Agent periodically broadcasts a 16 byte message via UDP, and responds to each incoming message by forwarding it back to the sender. It records the round-trip time (RTT) of each message it sends. We observe that under 1 KB, RTT does not vary with message size. Like Pingmesh [Guo et al. 2015], our RTT includes application, OS and hardware latencies across the network stack, as this is the latency perceived by the application.

We evaluate PERFORMAL in two environments, namely (1) an on-premise local cluster of up to 14 machines running Ubuntu 16.04 connected over a 1 Gbps network (star topology), each equipped with an Intel Xeon E5-1620 3.50 GHz processor and 32 GB of memory; and (2) a wide area network on Amazon EC2 using "large M5" instances.

6.1 Evaluating Performal on Varying System Configurations and Workloads

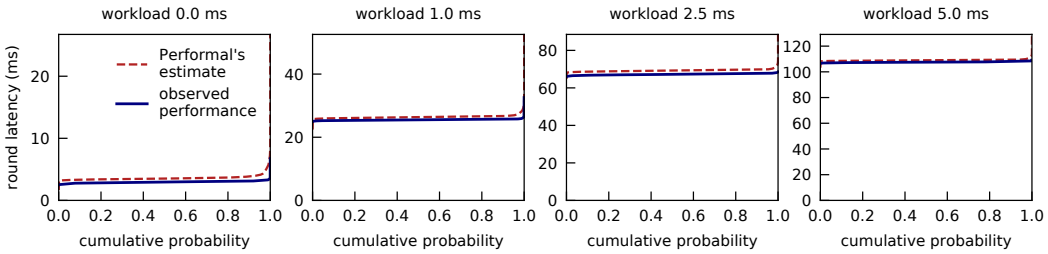

Fig. 4. Observed vs. estimated latency distributions of the distributed lock system, for one round in a cluster of size 10. A round is defined as the time it takes for the lock to traverse the ring once.

Table 2. Comparison of Performal's estimated latency upper bounds against the observed maximums for different workloads in the distributed lock system, on a cluster of size 10.

Workload	Observed max	Performal's bound
0.0 ms	6.7 ms	74.9 ms
1.0 ms	32.6 ms	108.2 ms
2.5 ms	68.5 ms	144.6 ms
5.0 ms	109.1 ms	185.7 ms

In the first experiment, we evaluate how well Performal estimates the behavior of a simple system across varying configurations and workloads. That is, for each configuration, the estimated upper bound on latency should not be exceeded by any observed latency, yet should be as close to the actual latency as possible.

This experiment is done on our distributed lock system. It consists of n nodes in a ring, passing a single lock token. Each node can `Grant` the lock to the next node if it current holds it, or `Accept` the lock from its predecessor. We reason about the time taken for the lock to traverse the ring once if each node passes the lock as soon as it receives it; this symbolic bound is listed in Table 1.

We run the distributed lock system on our local cluster, and compare Performal's predictions on ring sizes of 2, 4, 6, 8 and 10; and workloads of 0, 1, 2.5 and 5 milliseconds. Workload is defined as the minimum execution time required for the node to execute a `Grant` or an `Accept` step, enforced by respective sleep invocations within the procedures. For each workload, empirical distributions of the execution times of `Grant` and `Accept` steps are measured as input to Performal.

The graphs in Figure 4 illustrate Performal's estimated latency distributions is indeed an upper bound for the observed latencies. In particular, for all percentiles p, the p-th percentile latency in Performal's estimate upper-bounds the p-th percentile of the observed latency. Moreover, up to the 90-th percentile, Performal's upper bound remains tight with respect to the observed latency.

Significant divergence occurs only at the tail-end of the distributions. This degree of 'pessimism' at the tail is expected because the worst-case latency computed by Performal corresponds to *every* single action taking its *worst* measured execution time, which is unlikely to occur in actual runs of the system, especially with many nodes involved. Table 2 quantifies the divergence by comparing the observed and estimated maximum latencies.

In general, the same trends are observed across all workloads and cluster sizes. We conclude that Performal produces estimated latency bounds that are reasonably tight with respect to the

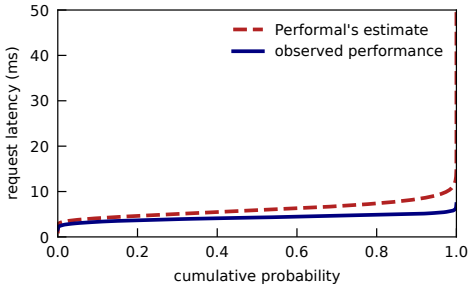

Fig. 5. Observed vs. estimated latency distribution of IronRSL for a system with 3 replicas ($f = 1$) on the local cluster.

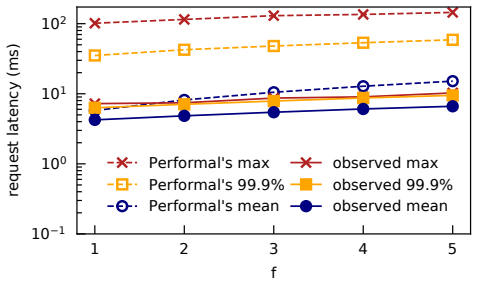

Fig. 6. Observed vs. estimated mean, 99.9 percentile, and worse-case latency of IronRSL across cluster sizes on the local cluster.

observed latency distributions across a variety of configurations and workloads. Meanwhile, the tail latency predicted by PERFORMAL provides a strong-confidence upper bound on the worst-case performance of the system.

6.2 Evaluating PERFORMAL on Large Complex Systems

Our next experiment evaluates PERFORMAL on a practical distributed system. We choose IronRSL for its complexity, and for the ubiquity of Paxos-based replication in the wild. In these experiments on our local cluster, we measure the end-to-end latency of a single request as perceived by the client, with fault-tolerance levels of $f = 1$ through $f = 5$. Figure 5 demonstrates that PERFORMAL's estimated latency safely bounds the observed latency for an $f = 1$ cluster; this graph is representative of other cluster sizes. Figure 6 confirms that although PERFORMAL's predictions tend to be pessimistic, they are safe upper bounds that well reflect the decay in performance as cluster size increases.

6.3 Evaluating PERFORMAL on Cloud Environments

We evaluate whether PERFORMAL generalizes to environments beyond our on-premise local cluster. We run IronRSL with three replicas in two Amazon EC2 configurations: (1) a single-region deployment with each replica in distinct availability zones in Ohio; and (2) a cross-region deployment with one replica in California, Oregon and Ohio respectively. The client is co-located with the Paxos leader in Ohio.

In such settings, round-trip times vary greatly between each pair of nodes, such that using a single symbolic term to represent all message delays will produce inaccurate real time estimates. Instead, we use different variables to denote the message delays between node pairs, resulting in the request latency bound that is a refinement of the generic bound we showed in Table 1, to account for this new, heterogeneous environment. In this case, a message delay d is parameterized by its endpoints, where ℓ denotes the site of the leader and cr denotes the site of the closest replica from the leader:

$$\begin{aligned} &\mathit{ProcessMessage} + \mathit{ActionsUpTo}(4) + \mathit{ActionsUpTo}(7) \\ &+ 3 \cdot \mathit{Queuing} + 2 \cdot d(\{\mathit{client}, \ell\}) + 2 \cdot d(\{\ell, cr\}) \end{aligned}$$

Currently, the respective inter-site delays used are determined manually using our knowledge of the protocol, where the closest replica from the leader characterizes the end-to-end latency of a request. Otherwise, one could also bake into PERFORMAL proofs the latency asymmetry between nodes, as described in Section 3.5.2; we did not do that for the sake of simplicity.

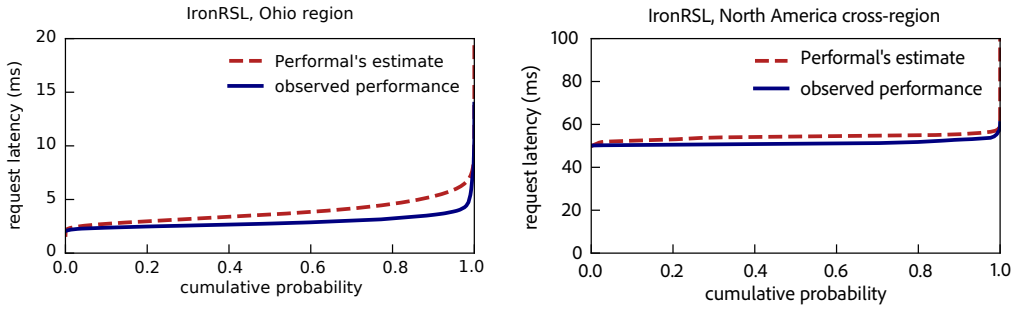

Fig. 7. Observed vs. estimated latency distributions of IronRSL with 3 replicas ($f = 1$) on Amazon EC2.

Figure 7 demonstrates that by relying only on localized measurements of the network and individual components, Performal maintains high-fidelity estimates of end-to-end latency in both the single-region and cross-region configurations. Performal's bounds about the mean, median and maximum of the distributions remain sound, i.e., above those of the observed behaviors. We conclude that the Performal methodology provides practical latency bounds in diverse deployment environments, including wide area cloud networks.

6.4 Evaluating Performal with Node Failures

Our final experiments validate Performal's end-to-end estimates when nodes may crash while a request is processed. We again run IronRSL with three replicas on Amazon EC2's Ohio region. The initial leader is now designed to crash some time t after it receives the request from the client. We gradually increase t and measure the end-to-end latency of the request—this includes the time it takes for the upcoming leader to detect the failure, complete the leader election phase, and to propose and execute the request.

We compare observed latencies with Performal's estimated bound on the request latency given that the failure may occur at *any* time throughout the execution (symbolic bound listed in Table 1), as shown in Figure 8.

When the leader crashes before $t = 2.0$ ms, it hasn't yet collected a quorum of `Accepted` messages. Here, the latency is largest as request processing is blocked until the next replica establishes itself as the leader. Between $t = 2.0$ ms to 2.5 ms, the original leader may sometimes have enough time to execute the request before crashing. Hence, the average end-to-end latency is lower as progress is blocked only in some executions. Beyond 2.5 ms, the original leader always has enough time to execute the request, and thus the latency returns to levels seen in the failure-free case of below 5 ms. Performal's upper bound reflects the worst-case behavior among all these cases.

6.5 Proof Effort

The number of source lines of code (SLOC) in our proofs were 197 SLOC for formalizing the framework of Section 3, 406 SLOC for our distributed lock service symbolic latency proof, and 1429 SLOC for our IronRSL latency proof without failure and 4666 SLOC with failure.

Accounting for failures generally increases proof effort, as it triggers code paths that are otherwise not exercised. In IronRSL, a leader failure triggers a leader election phase in the protocol that does not execute otherwise. Another reason for the higher proof effort is that failures can happen at any moment, so invariants must account for multiple possible scenarios. Despite the increased effort, such proofs are not fundamentally different from failure-free proofs.

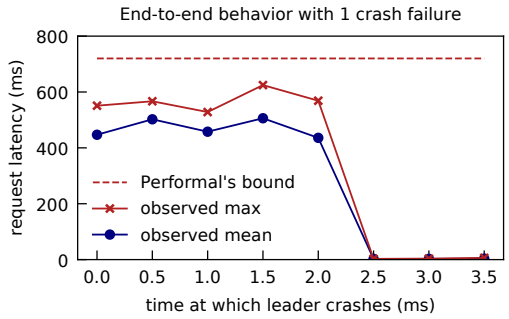

Fig. 8. Observed end-to-end request latency when crashing the Paxos leader at various intervals after it receives the request, compared to PERFORMAL's estimated upper bound on the end-to-end latency given that the leader may crash.

For reference, IronFleet's refinement proof for functional correctness was 3379 SLOC and its liveness proof was 7869 SLOC for IronRSL. In exchange for the modest effort needed to write a symbolic latency proof, PERFORMAL allows one to establish a hard symbolic guarantee that can be applied to various runtime environments.

6.6 Case Study: ZooKeeper

We now evaluate how PERFORMAL can be used to identify performance issues in systems that *do not* behave as expected. We pick ZooKeeper [Hunt et al. 2010] as our case study for its widespread use in critical cloud infrastructure, and for its prevalence in a detailed bug study [Gunawi et al. 2014].

In this case study, we focus on bug zk-1465 [Jira 2012] in version 3.4.3 of ZooKeeper. Our goal is to use PERFORMAL's symbolic latency derivation to identify the root cause of zk-1465: a prolonged period of unavailability when starting up the cluster or following a leader failure. In particular, when starting the cluster from a cold state, each node loads any existing snapshots from disk and the elected leader then synchronizes its state with the followers. The bug involves the leader invariably sending its *entire* snapshot to its followers, even when unnecessary, e.g. when the snapshots of each node were already deemed identical. This severely impacts availability since the leader will only start accepting client connections once this synchronization phase is complete.

We implement as a protocol in Dafny the start-up phase of version 3.4.3 of ZooKeeper, where the leader establishes communication with its followers and synchronizes its state. In this Dafny version of ZooKeeper, we translate the multi-threaded leader into a sequential state machine that non-deterministically takes a step on one of its threads. This is the source of the coefficients of f in the symbolic bounds in Table 1: f parallel actions in the Java implementation of ZooKeeper become f sequential actions in our version of the protocol, where f is the degree of fault tolerance of the system. This translation does not affect how the bug manifests; the bug hinders performance similarly regardless of sequential or parallel execution.

We first proved a symbolic bound for the time interval between the start of the handshake to the conclusion of synchronization in the presence of zk-1465, when each node is initialized with identical snapshots. This resulted in the following invariant: the condition in the leader code that decides if a leader-follower pair are in the same state will always (erroneously) evaluate to false. As such, the leader invariably defaults to sending an entire snapshot of its state.

We then proved the corresponding symbolic latency bound with a patch for zk-1465 in place. Indeed, under the patch, we can prove a new invariant that, given all nodes begin in the same state, the leader will never decide to send its full snapshot. This results in a tighter symbolic latency

bound, where the expensive operations marked in bold (Table 1) are replaced with the sending and processing of a message denoting that no synchronization is necessary.

Notably, in this case PERFORMAL helps developers identify both the bug and its root cause using only symbolic bounds, without having to convert these bounds into real time estimates and without having to perform any measurements.

7 RELATED WORK

Distributed System Performance Analysis. Debugging performance often focuses on distributed tracing [Barham et al. 2004; Erlingsson et al. 2012; Fonseca et al. 2007; Las-Casas et al. 2018; Mace et al. 2018; Sigelman et al. 2010; Tammana et al. 2018] and analysis of logged timing data [Benavides et al. 2019; Whittaker et al. 2018; Wu et al. 2019]. Tracing tracks the path that requests take through a system during runtime, helping a developer understand why a particular request was slow. However, tracing can only diagnose performance issues *after* they appear in production.

Benchmarking and testing [Cooper et al. 2010; Denaro et al. 2004] study the performance of a system outside of production. However, they require a deployment similar in scale to production. Moreover, these techniques provide no coverage guarantees and may miss corner-case problems.

There are works on statically finding performance bugs. Some use model checking to find scenarios that lead to degraded performance [Killian et al. 2010] or explore behaviors in search of anomalies [Suminto et al. 2015]. Others have focused on forecasting performance using critical path analysis [Saidi et al. 2009]. These tools do not provide any *guarantees* about the performance of distributed executions. There are also works on finding bounds on execution time for network functions [Iyer et al. 2022, 2019], but they do not deal with performance of a distributed system. Finally, a recent work [Mahgoub et al. 2022] estimates the latency of serverless applications, but the statistical analysis is based on user-defined DAG models of the system, as opposed to a formally verified system that may have arbitrary communication patterns.

Lastly, network calculus [Cruz 1991; Le Boudec and Thiran 2001] is another means to reason about the performance of networked systems, and has been used by a variety of work to provide network latency guarantees [Jang et al. 2015; Sariowan et al. 1999; Stoica et al. 2000; Zhang et al. 2022; Zhu et al. 2016, 2017]. These work complement PERFORMAL in enabling predictable network delays. In contrast, PERFORMAL includes detailed reasoning about how both network latency and local control flow affect overall latency.

Distributed Protocol Analysis. There are a variety of abstract models based on automata, process algebra, and petri nets used to check properties of general real-time systems [Alur 1999; Alur and Dill 1994; Berthomieu and Diaz 1991; David et al. 2010; De Prisco et al. 1997; Hennessy and Regan 1995; Nicollin and Sifakis 1992; Ramchandani 1973; Yi et al. 1995], sometimes used in conjunction with model checking [Alur et al. 1993]. These models are often designed to check for *correctness* of protocols designed using real time assumptions [Vereijken 1994], which is different from reasoning about the performance of a system. Additionally, protocol analysis misses potential complications introduced when elegant models are turned into actual implementations [Lee 2012].

WCET. Tools for finding the worst-case execution time (WCET) of local programs use a blend of fine-grained static analysis [Li et al. 2007] and measurement [Cazorla et al. 2013; Hansen et al. 2009], with some focusing on probabilistic guarantees [Bernat et al. 2003]. Measurements are typically used to capture the runtime of components that are too hard to reason about statically [Petters et al. 2007]. These techniques have been used to verify timing guarantees for software, like the seL4 microkernel [Blackham et al. 2011; Sewell et al. 2017] with the aim of creating real-time systems. In contrast, we aim to reason about performance in a non-real-time environment, using

an off-the-shelf kernel and network stack. In the spirit of hybrid WCET [Betts et al. 2010], we use measurements to estimate component runtimes; albeit in a distributed setting.

Program Analysis. Prior work on bounding resource consumption, such as memory footprint or evaluation steps, uses type systems for verifying and inferring resource bounds [Carbonneaux et al. 2015; Hoffmann et al. 2012, 2017; Hoffmann and Hofmann 2010]. Other work focuses on proving a program's running time complexity [Charguéraud and Pottier 2019; Guéneau et al. 2018; Haslbeck and Lammich 2022; McCarthy et al. 2016]. Most of these works do not infer real-time bounds for execution, and are designed for non-distributed code. Recent work [Muller and Hoffmann 2019] that aims to infer real-time bounds makes use of WCET, but still focuses on non-distributed code.

Formal Verification of Distributed Systems. Recent years have seen a lot of work on using formal verification to reason about centralized [Chajed et al. 2018, 2019; Chen et al. 2015; Ileri et al. 2018; Nelson et al. 2019, 2017] and distributed systems [Hawblitzel et al. 2015, 2017; Lesani et al. 2016; Wilcox et al. 2015], and on automating proofs for distributed protocols [Feldman et al. 2019; Hance et al. 2021; Ma et al. 2019; Padon et al. 2016; Yao et al. 2022, 2021]. Notably, the liveness proofs in IronFleet [Hawblitzel et al. 2015] also proved invariants about a request's progress through the system, but they were coarse-grained by design. Since those proofs were not meant to be converted to real time bounds, it was sufficient to assume the existence of some bound T such that all local actions' execution does not exceed T. In contrast, PERFORMAL presents a fine-grained model where each action is accounted for separately. PERFORMAL's proof sharding strategy of decomposing a protocol into stages of progress is also similar to the idea of phase structures presented in [Feldman et al. 2019]. We leave to future work the exploration of how the structure of latency invariants can be exploited to allow for automatic derivations. Overall, all of these systems focus on functional correctness and liveness, and do not provide any insight into system performance.

8 CONCLUSION

PERFORMAL shows that formal verification can be used in a new way: not just to prove a system's correctness, but also to reason about facets of its performance.

PERFORMAL uses a combination of formal verification and individual component measurements to produce rigorously-derived upper bounds for the worst case latency behavior of the system. Our evaluation shows that PERFORMAL is powerful enough to reason about complex distributed systems and to provide real time bounds that are a good proxy for their deployed performance.

This paper is not meant to be the final word in this area. There are several important challenges ahead in reasoning about the performance of distributed systems, such as proving throughput properties. Our hope is that the first steps taken by this paper will demonstrate the feasibility of proving performance properties and will inspire further work in this direction.

ACKNOWLEDGMENTS

We thank Boyu Tian for his early work on this project. We also thank our shepherd Zachary Tatlock and the anonymous PLDI reviewers for their insightful feedback. This work is partially supported by the National Science Foundation grants 2118512 and 2045541, and a gift from Meta.

DATA-AVAILABILITY STATEMENT

The source code used to produce the results in this paper are presented as an artifact hosted by Zenodo [art 2023]. Instructions to reproduce the results are documented in the artifact.

REFERENCES

2023. *Artifact for Performal: Formal Verification of Latency Properties for Distributed Systems*. Zenodo. https://doi.org/10.5281/zenodo.7812534

Akamai. 2017. The state of online retail performance. https://www.akamai.com/newsroom/press-release/akamai-releases-spring-2017-state-of-online-retail-performance-report

Rajeev Alur. 1999. Timed Automata. In *Computer Aided Verification*, Nicolas Halbwachs and Doron Peled (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 8–22.

Rajeev Alur, Costas Courcoubetis, and David Dill. 1993. Model-checking in dense real-time. *Information and computation* 104, 1 (1993), 2–34.

Rajeev Alur and David L Dill. 1994. A theory of timed automata. *Theoretical computer science* 126, 2 (1994), 183–235.

Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. 2004. Using Magpie for request extraction and workload modelling.. In *OSDI*, Vol. 4. 18–18.

Zachary Benavides, Keval Vora, and Rajiv Gupta. 2019. DProf: Distributed Profiler with Strong Guarantees. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 156 (Oct. 2019), 24 pages. https://doi.org/10.1145/3360582

Guillem Bernat, Antoine Colin, and Stefan Petters. 2003. *pwcet: A tool for probabilistic worst-case execution time analysis of real-time systems*. University of York, Department of Computer Science.

B. Berthomieu and M. Diaz. 1991. Modeling and verification of time dependent systems using time Petri nets. *IEEE Transactions on Software Engineering* 17, 3 (1991), 259–273.

Adam Betts, Nicholas Merriam, and Guillem Bernat. 2010. Hybrid measurement-based WCET analysis at the source level using object-level traces. In *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Bernard Blackham, Yao Shi, Sudipta Chattopadhyay, Abhik Roychoudhury, and Gernot Heiser. 2011. Timing Analysis of a Protected Operating System Kernel. In *IEEE Real-Time Systems Symposium*. IEEE Computer Society, Vienna, Austria, 339–348.

Quentin Carbonneaux, Jan Hoffmann, and Zhong Shao. 2015. Compositional Certified Resource Bounds. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 467–478. https://doi.org/10.1145/2737924.2737955

Francisco J. Cazorla, Tullio Vardanega, Eduardo Quiñones, and Jaume Abella. 2013. Upper-bounding Program Execution Time with Extreme Value Theory. In *13th International Workshop on Worst-Case Execution Time Analysis (OpenAccess Series in Informatics (OASIcs), Vol. 30)*, Claire Maiza (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 64–76. https://doi.org/10.4230/OASIcs.WCET.2013.64

Tej Chajed, Frans Kaashoek, Butler Lampson, and Nickolai Zeldovich. 2018. Verifying concurrent software using movers in CSPEC. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 306–322. https://www.usenix.org/conference/osdi18/presentation/chajed

Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. 2019. Verifying Concurrent, Crash-Safe Systems with Perennial. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 243–258. https://doi.org/10.1145/3341301.3359632

Arthur Charguéraud and François Pottier. 2019. Verifying the Correctness and Amortized Complexity of a Union-Find Implementation in Separation Logic with Time Credits. *Journal of Automated Reasoning* 62, 3 (2019), 331–365. https://doi.org/10.1007/s10817-017-9431-7

Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2015. Using Crash Hoare Logic for Certifying the FSCQ File System. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 18–37. https://doi.org/10.1145/2815400.2815402

Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

R.L. Cruz. 1991. A calculus for network delay. I. Network elements in isolation. *IEEE Transactions on Information Theory* 37, 1 (1991), 114–131. https://doi.org/10.1109/18.61109

Alexandre David, Kim G. Larsen, Axel Legay, Ulrik Nyman, and Andrzej Wasowski. 2010. Timed I/O Automata: A Complete Specification Theory for Real-Time Systems. In *Proceedings of the 13th ACM International Conference on Hybrid Systems: Computation and Control* (Stockholm, Sweden) *(HSCC '10)*. Association for Computing Machinery, New York, NY, USA, 91–100. https://doi.org/10.1145/1755952.1755967

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

Roberto De Prisco, Butler Lampson, and Nancy Lynch. 1997. Revisiting the Paxos algorithm. In *Distributed Algorithms*, Marios Mavronicolas and Philippas Tsigas (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 111–125.

Giovanni Denaro, Andrea Polini, and Wolfgang Emmerich. 2004. Early Performance Testing of Distributed Software Applications. In *Proceedings of the 4th International Workshop on Software and Performance* (Redwood Shores, California) *(WOSP '04)*. Association for Computing Machinery, New York, NY, USA, 94–103. https://doi.org/10.1145/974044.974059

Edsger W. Dijkstra. 1972. The Humble Programmer. *Commun. ACM* 15, 10 (oct 1972), 859–866. https://doi.org/10.1145/355604.361591

DynamoDB. 2015. https://aws.amazon.com/message/5467D2/

Yoav Einav. 2019. Amazon Found Every 100ms of Latency cost them 1 percent in sales. https://www.gigaspaces.com/blog/amazon-found-every-100ms-of-latency-cost-them-1-in-sales

Úlfar Erlingsson, Marcus Peinado, Simon Peter, Mihai Budiu, and Gloria Mainar-Ruiz. 2012. Fay: Extensible Distributed Tracing from Kernels to Clusters. *ACM Trans. Comput. Syst.* 30, 4, Article 13 (Nov. 2012), 35 pages. https://doi.org/10.1145/2382553.2382555

Yotam M. Y. Feldman, James R. Wilcox, Sharon Shoham, and Mooly Sagiv. 2019. Inferring Inductive Invariants from Phase Structures. In *Computer Aided Verification*, Isil Dillig and Serdar Tasiran (Eds.). Springer International Publishing, Cham, 405–425.

Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. 2007. X-Trace: A Pervasive Network Tracing Framework. In *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation* (Cambridge, MA) *(NSDI'07)*. USENIX Association, USA, 20.

Armaël Guéneau, Arthur Charguéraud, and François Pottier. 2018. A Fistful of Dollars: Formalizing Asymptotic Complexity Claims via Deductive Program Verification. In *Programming Languages and Systems*, Amal Ahmed (Ed.). Springer International Publishing, Cham, 533–560.

Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. 2014. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SOCC '14)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/2670979.2670986

Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. 2015. Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (London, United Kingdom) *(SIGCOMM '15)*. Association for Computing Machinery, New York, NY, USA, 139–152. https://doi.org/10.1145/2785956.2787496

Josef Hadar and William R. Russell. 1969. Rules for Ordering Uncertain Prospects. *The American Economic Review* 59, 1 (1969), 25–34. http://www.jstor.org/stable/1811090

Travis Hance, Marijn Heule, Ruben Martins, and Bryan Parno. 2021. Finding Invariants of Distributed Systems: It's a Small (Enough) World After All. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 115–131. https://www.usenix.org/conference/nsdi21/presentation/hance

Jeffery Hansen, Scott Hissam, and Gabriel A Moreno. 2009. Statistical-based wcet estimation and validation. In *9th international workshop on worst-case execution time analysis (WCET'09)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

Maximilian P. L. Haslbeck and Peter Lammich. 2022. For a Few Dollars More: Verified Fine-Grained Algorithm Analysis Down to LLVM. *ACM Trans. Program. Lang. Syst.* 44, 3, Article 14 (jul 2022), 36 pages. https://doi.org/10.1145/3486169

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. 2015. IronFleet: Proving Practical Distributed Systems Correct *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 1–17. https://doi.org/10.1145/2815400.2815428

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. 2017. IronFleet: Proving Safety and Liveness of Practical Distributed Systems. *Commun. ACM* 60, 7 (jun 2017), 83–92. https://doi.org/10.1145/3068608

M. Hennessy and T. Regan. 1995. A Process Algebra for Timed Systems. *Inf. Comput.* 117, 2 (mar 1995), 221–239. https://doi.org/10.1006/inco.1995.1041

Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2012. Resource Aware ML. In *Computer Aided Verification*, P. Madhusudan and Sanjit A. Seshia (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 781–786.

Jan Hoffmann, Ankush Das, and Shu-Chun Weng. 2017. Towards Automatic Resource Bound Analysis for OCaml. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. Association for Computing Machinery, New York, NY, USA, 359–373. https://doi.org/10.1145/3009837.3009842

Jan Hoffmann and Martin Hofmann. 2010. Amortized Resource Analysis with Polynomial Potential. In *Programming Languages and Systems*, Andrew D. Gordon (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 287–306.

Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-Free Coordination for Internet-Scale Systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference* (Boston, MA) *(USENIXATC'10)*. USENIX Association, USA, 11.

Atalay Ileri, Tej Chajed, Adam Chlipala, Frans Kaashoek, and Nickolai Zeldovich. 2018. Proving confidentiality in a file system using DiskSec. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 323–338. https://www.usenix.org/conference/osdi18/presentation/ileri

Rishabh Iyer, Katerina Argyraki, and George Candea. 2022. Performance Interfaces for Network Functions. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 567–584. https://www.usenix.org/conference/nsdi22/presentation/iyer

Rishabh Iyer, Luis Pedrosa, Arseniy Zaostrovnykh, Solal Pirelli, Katerina Argyraki, and George Candea. 2019. Performance Contracts for Software Network Functions. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 517–530. https://www.usenix.org/conference/nsdi19/presentation/iyer

Keon Jang, Justine Sherry, Hitesh Ballani, and Toby Moncaster. 2015. Silo: Predictable Message Latency in the Cloud. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (London, United Kingdom) *(SIGCOMM '15)*. Association for Computing Machinery, New York, NY, USA, 435–448. https://doi.org/10.1145/2785956.2787479

Apache Jira. 2012. https://issues.apache.org/jira/browse/ZOOKEEPER-1465

Charles Killian, Karthik Nagaraj, Salman Pervez, Ryan Braud, James W. Anderson, and Ranjit Jhala. 2010. Finding Latent Performance Bugs in Systems Implementations. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Santa Fe, New Mexico, USA) *(FSE '10)*. Association for Computing Machinery, New York, NY, USA, 17–26. https://doi.org/10.1145/1882291.1882297

Leslie Lamport. 1998. The Part-time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (May 1998), 133–169. https://doi.org/10.1145/279227.279229

Leslie Lamport. 2001. Paxos Made Simple. *ACM SIGACT News (Distributed Computing Column)* 32, 4 (Dec. 2001), 51–58.

Pedro Las-Casas, Jonathan Mace, Dorgival Guedes, and Rodrigo Fonseca. 2018. Weighted Sampling of Execution Traces: Capturing More Needles and Less Hay. In *Proceedings of the ACM Symposium on Cloud Computing* (Carlsbad, CA, USA) *(SoCC '18)*. Association for Computing Machinery, New York, NY, USA, 326–332. https://doi.org/10.1145/3267809.3267841

Jean-Yves Le Boudec and Patrick Thiran. 2001. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Springer-Verlag, Berlin, Heidelberg.

Edward A. Lee. 2012. Verifying Real-Time Software is Not Reasonable (Today). In *Proceedings of the 8th International Conference on Hardware and Software: Verification and Testing* (Haifa, Israel) *(HVC'12)*. Springer-Verlag, Berlin, Heidelberg, 2. https://doi.org/10.1007/978-3-642-39611-3_2

K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning* (Dakar, Senegal) *(LPAR'10)*. Springer-Verlag, Berlin, Heidelberg, 348–370. http://dl.acm.org/citation.cfm?id=1939141.1939161

Mohsen Lesani, Christian J. Bell, and Adam Chlipala. 2016. Chapar: Certified Causally Consistent Distributed Key-Value Stores. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 357–370. https://doi.org/10.1145/2837614.2837622

Xianfeng Li, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. 2007. Chronos: A timing analyzer for embedded software. *Science of Computer Programming* 69, 1-3 (2007), 56–67.

Haojun Ma, Aman Goel, Jean-Baptiste Jeannin, Manos Kapritsos, Baris Kasikci, and Karem A. Sakallah. 2019. I4: Incremental Inference of Inductive Invariants for Verification of Distributed Protocols *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 370–384. https://doi.org/10.1145/3341301.3359651

Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. 2018. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. *ACM Trans. Comput. Syst.* 35, 4, Article 11 (Dec. 2018), 28 pages. https://doi.org/10.1145/3208104

Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. 2022. ORION and the Three Rights: Sizing, Bundling, and Prewarming for Serverless DAGs. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 303–320. https://www.usenix.org/conference/osdi22/presentation/mahgoub

Jay McCarthy, Burke Fetscher, Max New, Daniel Feltey, and Robert Bruce Findler. 2016. A Coq Library for Internal Verification of Running-Times. In *Functional and Logic Programming*, Oleg Kiselyov and Andy King (Eds.). Springer International Publishing, Cham, 144–162.

Stefan K Muller and Jan Hoffmann. 2019. Combining Source and Target Level Cost Analyses for OCaml Programs. (2019).

Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. 2019. Scaling Symbolic Evaluation for Automated Verification of Systems Code with Serval *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 225–242. https://doi.org/10.1145/3341301.3359641

Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. 2017. Hyperkernel: Push-Button Verification of an OS Kernel. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. ACM, New York, NY, USA, 252–269. https://doi.org/10.1145/3132747.3132748

Xavier Nicollin and Joseph Sifakis. 1992. An overview and synthesis on timed process algebras. In *Real-Time: Theory in Practice*, J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 526–548.

Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. 2016. Ivy: Safety Verification by Interactive Generalization. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 614–630. https://doi.org/10.1145/2908080.2908118

Stefan M Petters. 2007. *Execution-time profiles.* Technical Report. Technical report, NICTA, Sydney, Australia.

Stefan M Petters, Patryk Zadarnowski, and Gernot Heiser. 2007. Measurements or static analysis or both?. In *7th International Workshop on Worst-Case Execution Time Analysis (WCET'07)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

Chander Ramchandani. 1973. *Analysis of asynchronous concurrent systems by timed Petri nets.* Ph. D. Dissertation. Massachusetts Institute of Technology.

Ali G. Saidi, Nathan L. Binkert, Steven K. Reinhardt, and Trevor Mudge. 2009. End-to-End Performance Forecasting: Finding Bottlenecks before They Happen. *SIGARCH Comput. Archit. News* 37, 3 (June 2009), 361–370. https://doi.org/10.1145/1555815.1555800

H. Sariowan, R.L. Cruz, and G.C. Polyzos. 1999. SCED: a generalized scheduling policy for guaranteeing quality-of-service. *IEEE/ACM Transactions on Networking* 7, 5 (1999), 669–684. https://doi.org/10.1109/90.803382

Thomas Sewell, Felix Kam, and Gernot Heiser. 2017. High-Assurance Timing Analysis for a High-Assurance Real-Time Operating System. *Real-Time Syst.* 53, 5 (sep 2017), 812–853. https://doi.org/10.1007/s11241-017-9286-3

Benjamin H Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. Dapper, a large-scale distributed systems tracing infrastructure. (2010).

Swaminathan Sivasubramanian. 2012. Amazon DynamoDB: A Seamlessly Scalable Non-Relational Database Service. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (Scottsdale, Arizona, USA) *(SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 729–730. https://doi.org/10.1145/2213836.2213945

I. Stoica, H. Zhang, and T.S.E. Ng. 2000. A hierarchical fair service curve algorithm for link-sharing, real-time, and priority services. *IEEE/ACM Transactions on Networking* 8, 2 (2000), 185–199. https://doi.org/10.1109/90.842141

Riza O. Suminto, Agung Laksono, Anang D. Satria, Thanh Do, and Haryadi S. Gunawi. 2015. Towards Pre-Deployment Detection of Performance Failures in Cloud Distributed Systems. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*. USENIX Association, Santa Clara, CA. https://www.usenix.org/conference/hotcloud15/workshop-program/presentation/suminto

Praveen Tammana, Rachit Agarwal, and Myungjin Lee. 2018. Distributed Network Monitoring and Debugging with SwitchPointer. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 453–456. https://www.usenix.org/conference/nsdi18/presentation/tammana

J.J. Vereijken. 1994. *Fischer's protocol in timed process algebra.* Technische Universiteit Eindhoven.

Michael Whittaker, Cristina Teodoropol, Peter Alvaro, and Joseph M. Hellerstein. 2018. Debugging Distributed Systems with Why-Across-Time Provenance. In *Proceedings of the ACM Symposium on Cloud Computing* (Carlsbad, CA, USA) *(SoCC '18)*. Association for Computing Machinery, New York, NY, USA, 333–346. https://doi.org/10.1145/3267809.3267839

James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. 2015. Verdi: A Framework for Implementing and Formally Verifying Distributed Systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 357–368. https://doi.org/10.1145/2737924.2737958

Yang Wu, Ang Chen, and Linh Thi Xuan Phan. 2019. Zeno: Diagnosing Performance Problems with Temporal Provenance. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 395–420. https://www.usenix.org/conference/nsdi19/presentation/wu

Jianan Yao, Runzhou Tao, Ronghui Gu, and Jason Nieh. 2022. DuoAI: Fast, Automated Inference of Inductive Invariants for Verifying Distributed Protocols. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 485–501. https://www.usenix.org/conference/osdi22/presentation/yao

Jianan Yao, Runzhou Tao, Ronghui Gu, Jason Nieh, Suman Jana, and Gabriel Ryan. 2021. DistAI: Data-Driven Automated Invariant Learning for Distributed Protocols. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 405–421. https://www.usenix.org/conference/osdi21/presentation/yao

Wang Yi, Paul Pettersson, and Mats Daniels. 1995. Automatic verification of real-time communicating systems by constraint-solving. In *Formal Description Techniques VII*. Springer, 243–258.

Yiwen Zhang, Gautam Kumar, Nandita Dukkipati, Xian Wu, Priyaranjan Jha, Mosharaf Chowdhury, and Amin Vahdat. 2022. Aequitas: Admission Control for Performance-Critical RPCs in Datacenters. In *Proceedings of the ACM SIGCOMM 2022*

Conference (Amsterdam, Netherlands) *(SIGCOMM '22)*. Association for Computing Machinery, New York, NY, USA, 1–18. https://doi.org/10.1145/3544216.3544271

Timothy Zhu, Daniel S. Berger, and Mor Harchol-Balter. 2016. SNC-Meister: Admitting More Tenants with Tail Latency SLOs. In *Proceedings of the Seventh ACM Symposium on Cloud Computing* (Santa Clara, CA, USA) *(SoCC '16)*. Association for Computing Machinery, New York, NY, USA, 374–387. https://doi.org/10.1145/2987550.2987585

Timothy Zhu, Michael A. Kozuch, and Mor Harchol-Balter. 2017. WorkloadCompactor: Reducing Datacenter Cost While Providing Tail Latency SLO Guarantees. In *Proceedings of the 2017 Symposium on Cloud Computing* (Santa Clara, California) *(SoCC '17)*. Association for Computing Machinery, New York, NY, USA, 598–610. https://doi.org/10.1145/3127479.3132245

Received 2022-11-10; accepted 2023-03-31