Finding Latent Performance Bugs in Systems Implementations

Charles Killian Karthik Nagaraj Ryan Braud James W. Anderson
Salman Pervez Ranjit Jhala
Purdue University University of California, San Diego
{ckillian, knagara, spervez}@cs.purdue.edu {rbraud, jwanderson, jhala}@cs.ucsd.edu

ABSTRACT

Robust distributed systems commonly employ high-level recovery mechanisms enabling the system to recover from a wide variety of problematic environmental conditions such as node failures, packet drops and link disconnections. Unfortunately, these recovery mechanisms also effectively mask additional serious design and implementation errors, disguising them as *latent performance bugs* that severely degrade end-to-end system performance. These bugs typically go unnoticed due to the challenge of distinguishing between a bug and an intermittent environmental condition that must be tolerated by the system. We present techniques that can automatically pinpoint latent performance bugs in systems implementations, in the spirit of recent advances in model checking by systematic state space exploration. The techniques proceed by automating the process of conducting random simulations, identifying performance anomalies, and analyzing anomalous executions to pinpoint the circumstances leading to performance degradation.

By focusing our implementation on the MACE toolkit, MACEPC can be used to test our implementations directly, without modification. We have applied MACEPC to five thoroughly tested and trusted distributed systems implementations. MACEPC was able to find significant, previously unknown, long-standing performance bugs in each of the systems, and led to fixes that significantly improved the end-to-end performance of the systems.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design—*Distributed Systems*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*

General Terms

Performance

Keywords

Mace, MacePC, performance, debugging, distributed systems

FSE-18, November 7–11, 2010, Santa Fe, New Mexico, USA.

1. INTRODUCTION

It is hard to build correct, high-performance distributed systems. As with any concurrent setting, the nastiest bugs are those that are caused by the unexpected temporal interleavings of events. Distributed settings compound this problem by exploding the number of interleavings drastically through their node failures, message reordering, *etc.* Thus, in addition to designing communication protocols and data structures that work in the common case, the developer must account for the fact that nodes participating in the system may join, leave or fail at any moment, and that the network substrate may corrupt, reorder, or drop messages sent between nodes. As a result, the most pernicious problems arise, not from algorithmic issues which affect *every* execution and hence are amenable to profiling, but from relatively rare corner-case node interactions or unexpected packet delays or drops. The resulting performance anomalies are difficult to reproduce, and hence, to find and to fix.

Recently, several authors have proposed techniques to find [7, 13, 29] and debug [6, 17] corner-case *correctness* problems in distributed systems. While ensuring the correctness of distributed systems is a necessity, performance is crucial for these systems and finding and fixing performance bugs can be as important and challenging. Specific challenges in performance debugging include: First, due to the importance of guaranteeing correctness and reliability in the face of all possible event interleavings, developers typically build fail-safes into the system—worst-case recovery mechanisms that periodically kick in and restore the system to a consistent and stable configuration. Unfortunately, these recovery mechanisms sweep serious design and implementation flaws under the rug, by disguising them as *latent performance anomalies* that severely affect the responsiveness, availability, and end-to-end behavior of the system. Second, the standard means of debugging distributed systems is to add code that logs the sequence of events along each execution. Unfortunately, the amount of logging required to accurately correlate events across multiple nodes can impact the performance characteristics of a given live or simulated execution which makes it hard to use log analysis to find performance bugs. Third, even if one could generate high fidelity logs with low overhead, performance problems often only manifest in *long* executions, not the *short* correctness violations returned by model checking [19, 22] or symbolic execution [2, 8]. Thus, to isolate the performance bug, the programmer must undertake the daunting task of wading through hundreds of megabytes of logs comprising tens of thousands of events spread across multiple nodes, manually tracing the complex communication and control flow.

In this paper, we present the MACE Performance Checker, a technique that automates the process of *finding* latent performance bugs in event-based distributed systems implementations, and automates the process of *isolating the root cause* of the error. MACEPC is based upon the insight that a class of performance bugs manifest as *anomalous executions* [5], i.e. executions whose performance is much worse than the expected performance observed along other executions. Consequently, we reduce finding performance bugs to three tasks: (i) determining expected performance of the system, (ii) finding executions whose performance is much worse than expected, and (iii) sifting through anomalous executions to pinpoint circumstances causing the performance degradation.

MACEPC carries out these tasks by marrying state-space exploration with time-based event simulation. First, we show how by sampling the state space of the unmodified implementation we can construct *Event Duration Distributions* (EDDs)—probability distributions that describe how long each low-level event takes to execute. By combining EDDs with a programmer-specified *stopping condition* that describes when the desired task is completed, MACEPC is able to develop a profile of the expected system performance. Second, we show how to combine systematic testing with the EDDs to find anomalous executions that take much longer than expected to reach the stopping condition, i.e., whose performance deviates significantly from expected performance. Third, we show how to analyze the executions explored to isolate the root cause of the performance degradation. We narrow the circumstances with an algorithm that characterizes the *divergence point* of an anomalous execution—the point along the execution such that *before* the divergence, some alternate execution achieves acceptable performance, but immediately *after* the divergence, executions of the system have bad performance. We then demonstrate how collecting event profiles of executions can simplify debugging, by correlating certain types of events with bad performance. Together, these techniques can save programmers a substantial amount of time devoted to diagnosing and fixing the performance bug.

While the techniques behind MACEPC could be used with other event-simulators to find performance problems in simulation, we focus our implementation on the MACE toolkit [12]. MACE is a language and toolkit for building a broad class of event-based distributed systems. We have implemented more than ten significant systems in MACE, most of which were proposed by others. This set includes distributed hash tables [24, 26, 27], application layer multicast and file distribution [3,11,14], network measurement services [4], and consensus protocols [16] ready to run over the Internet. MACE has been in development for seven years, is publicly available for download, and has been used by researchers at Purdue, UCSD, EPFL, Cornell, UT-Austin, UCLA, HP Labs, MSR (Redmond, Silicon Valley, and Asia), and a handful of other universities worldwide in support of their own research and development. MACEPC leverages the MACE toolkit, allowing us to test unmodified, deployable MACE distributed systems *implementations*.

Finally, we present results from applying our techniques to five real, complex distributed systems. We discuss the application of our tool to the systems, subtle performance bugs we found, and our solutions. Many of these bugs are instances of correctness bugs masked by periodic corrective protocols. Importantly, the implementations were already thoroughly tested and had been run across the Internet for several years. Further, while we had been aware of intermittent performance issues and had attempted to diagnose them on multiple occasions, we were unable to do so without MACEPC. On using MACEPC to Distributed Hash Table (DHT) systems, we improved the worst case ring stabilization time for a PASTRY [26] implementation by a factor of 50, and in a highly-tuned BAMBOO [24] implementation we improved both the consitency and latency of Key-Value lookups by 15-30%.

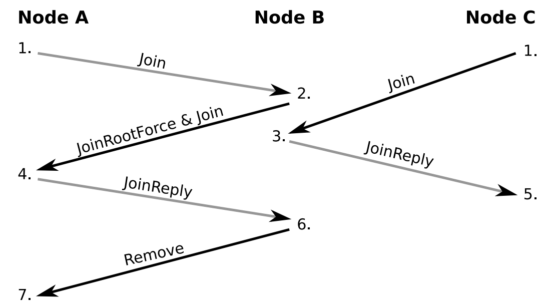

Figure 1: Motivating Example. An example scenario seen in our random tree protocol. (1) Nodes A and C initially attempt to join the tree at node B. (2) Because node A was more fit to be the root of the tree than node B, node B told node A to be the root and attempted to join under it. (3) Meanwhile, upon receiving a Join from node C, node B changes its state to joined since it accepts a child. Later (6), node B rejects the JoinReply from node A as it is already joined. This leaves the system in a state with two trees, corrected only later during the recovery protocol. Note that if either join completed before the other began, this bug would not have occurred.

2. BACKGROUND

Our goal is to automatically isolate latent performance bugs that arise due to corner-case conditions that escaped the developer's attention during system design and implementation. These bugs are particularly hard to find and fix because they cannot be easily detected in the presence of recovery code, nor reproduced due to their infrequency. In particular, we do not aim to automatically discover *algorithmic* bottlenecks arising from poor design; these errors are easier to detect as they hinder *every* execution's performance.

As a motivating example, consider the scenario illustrated in Figure 1 that arose deploying a file distribution application. The application used an overlay tree as the basic reliable multicast communication channel. Under some circumstances, the application stalled when the overlay tree took too long to stabilize. Upon painstakingly sifting through logs from system runs we found that network latencies caused unexpected re-ordering of events, disconnecting certain nodes from the remainder. Later, a recovery protocol restored the system, triggered by its coarse-grained timeout.

Note that the recovery protocol eventually restores the system to a consistent state, and hence there is no correctness error, *per se*. Moreover, existing correctness checkers like [7, 13, 19, 22], ignore quantitative system aspects like time, and hence cannot be used to find latent performance bugs such as the above.

In this section, we first describe the basic system model of a distributed system execution, suitable for model checking. We then describe the timed system model, which integrates the timing information of a time-based simulator with the basic system model.

2.1 Basic System Model

We consider distributed systems implemented as atomic event-driven state-machines. The entire distributed system therefore comprises processes running at the individual nodes together with the network layer that the nodes use to exchange *events*.

The execution of an individual node can be viewed as a state machine that moves from one state to the next by executing the *transition* triggered by reception of events. Events typically come from an application, the network or the timer scheduler. When a node receives an event, it executes a transition by executing the handler function registered to receive the callback. The handler is an arbitrary piece of non-blocking code which executes atomically, and may in turn trigger events asynchronously on other system nodes.

The *state* of a node is the set of all variables of the state machine at that node. The state of all nodes together, combined with the network, timer, and application simulator state comprise the *system state*. An *execution* of the distributed system is an *initial system state* and an ordered set of pairs: $\langle node, event \rangle$. Intuitively, the system evolves as follows: at each step, the system triggers $(event)$ on $(node)$, executing at $node$ the corresponding transition and taking the entire system into a new state.

2.2 Timed System Model

Our goal is to isolate bugs that adversely affect the end-to-end performance of a given distributed system, i.e. bugs that adversely affect the *time* taken to carry out the tasks for which the system was built. Thus, we must extend our system model to track the passage of time, as other time-based simulators do.

In an event driven system, there are two ways in which time advances. The first is the time that elapses between the sending and reception of an event. This includes (1) the passage of time (due to network latency) between the instants a network event is sent and received, and, (2) the passage of time between the instants a timer event is scheduled and fires. The second is the time it takes to execute the transition corresponding to a given event, i.e. to execute the code of the event's handler. While our system model requires event handlers be non-blocking (thus implying they are fast), in practice they take a non-negligible amount of time. To be consistent, a node's time must be updated according to both of these factors.

To account for time, we include a per-node clock, and extend the notion of an *execution* as an initial system state and set of tuples:

$$\langle node, event, start, duration \rangle$$

ordered by the element $start$. Intuitively, the system evolves as follows. At each step, the system picks the next tuple, and updates the system clock for $node$ to be the maximum of its current value and $start$. Next, the system triggers $event$ on $node$, thereby executing at $node$ the corresponding transition. At the end of the transition, the $node$ system clock is incremented by $duration$ and thus the whole system moves to a new state.

For simplicity, in the initial state assume all nodes share the same *start time*. We define the *system time* as the average node time across all the system's nodes. We define the *running time* as the difference between the system time and the start time. To formalize the notion of performance we require that the developer provide a *stopping condition*, a predicate over the system state that is true once the end-to-end task is accomplished. In the motivating example above, our stopping condition is that the system form a spanning tree across all nodes. Thus, the performance of a particular execution is formalized as the *execution time*—the running time when the system satisfies the stopping condition.

3. ALGORITHM

We now present our algorithm that uses developer-specified, high-level stopping conditions to find performance bugs. Figure 2 shows the three phases of the algorithm: *(1) Training,* including both *(a) Event Duration Training,* where we determine the CPU time of each type of (atomic) event-transition, and *(b) Performance Training,* where we use the event durations to determine the "normal" execution time; *(2) Anomaly Detection,* where we search the space of behaviors to find poor performing executions, i.e. whose execution time is significantly higher than normal; and finally, *(3) Anomaly Analysis,* including both *(a) Divergence Detection,* where we analyze the anomalous execution to determine the divergence point,

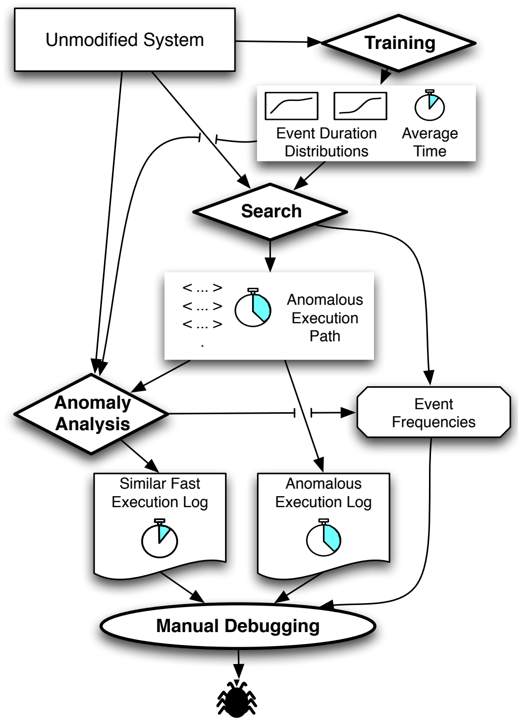

Figure 2: System Architecture. First, MacePC is trained on the unmodified system: a synthetic set of event duration distributions (EDD) are used to create realistic EDD, which are then used to sample the execution space to learn what an "average" execution is. Next, the system and EDD are input into the Explorer **algorithm, and it continues until an anomalous execution is found. Finally, our anomaly analysis algorithm locates the most similar "average" execution, and updates the event frequencies, reporting any correlations to execution time. At this point, any debugging tool or user process can be used to compare the two executions until the source of the bug is found.**

narrowing the search for the performance degradation cause, and *(b) Frequency Correlation,* where we determine if certain event behaviors are related to the performance degradation.

3.1 Ingredients

We start by presenting the key ingredients of our technique, the notion of an event duration distribution, and the two procedures Simulator and Explorer that are used across multiple phases.

Event Duration Distributions. For the purpose of simulation, we represent the time taken to handle each event type (e.g. the time taken to execute a transition or deliver a message) with a probability distribution generated from actual executions of the system. We call these the *event duration distributions (EDD)*. There are three main advantages EDDs offer over the naive approach of using raw execution times observed during simulation.

Repeatability: After each event, the simulator determines the event duration by randomly sampling the event's distribution. By recording the sequence of random numbers used for sampling, we ensure execution repeatability with exactly the same execution time, regardless of variability in events' actual time. Consequently, even after subsequently modified code—e.g., to add logging, perform time-intensive testing, *etc.*—the performance observed by the simulator remains the same as the original, unmodified code.

Coverage: EDD sampling allows exploration of executions with event timing combinations never seen in a single run. This allows the simulator to explore more variations in timing behavior, including potentially hard-to-find corner cases.

Malleability: Timing distributions for each event enable exploring "what if" scenarios with event times, by changing the distribution for events to see the effects on average execution duration. This allows speculatively evaluating effects of different algorithms, buffering strategies, *etc.,* without actual implementations.

We could construct similar distributions for network latency and bandwidth observed on a particular real-world topology. Currently we take a simpler approach of randomized latencies (cf. Section 4).

Algorithm 1 Simulator

Input: System S
Input: Stopping condition C
Input: Set EDD of timing distributions
  $events$: $\langle node, event, start \rangle$ Queue
  $timedEvents$: $\langle node, event, start, duration \rangle$ Queue
  $realTimes$: $\langle event, realTime \rangle$ Queue
  Initialize S and $events$
  while C **not satisfied** by S **do**
    $\langle node, event, start \rangle = events.\mathsf{pop}()$
    $node.time = \mathsf{max}(node.time, start)$
    $startTime = \mathsf{RealTime}()$
    Simulate $event$ on $node$
    $realTime = \mathsf{RealTime}() - startTime$
    $duration = EDD[event][\mathsf{rand}()]$
    $node.time = node.time + duration$
    $timedEvents.\mathsf{push}(\langle node, event, start, duration \rangle)$
    $realTimes.\mathsf{push}(\langle event, realTime \rangle)$
  return $\langle timedEvents, realTimes \rangle$

Algorithm 1: Simulator implements the core simulation mechanism. This algorithm takes three input parameters: a system to be simulated, a stopping condition, and a set of EDDs. The algorithm constructs three queues: (i) $events$, which holds the currently pending events, (ii) $timedEvents$, which records the sequence of already executed events, and their *sampled* durations, (iii) $realTimes$, which records the sequence of already executed events, and their *real* durations. $events$ is initialized with a small set of events used to bootstrap the system (typically an application initializing event on each node), and $timedEvents$ and $realTimes$ are initially empty. Next, the simulator enters a loop in which it keeps executing pending events until the stopping condition becomes true. At each iteration, the algorithm pops the first pending event (i.e. with the smallest start time) off the $events$ queue. The node running the chosen event sets its clock to the maximum of its current clock value and the scheduled start time of the event. Next, the algorithm records the real time and executes the event at the chosen node, potentially causing other events to be enqueued (e.g., in the case where a new timer or network event is scheduled). To determine how long the event transition took, the algorithm randomly samples the EDD for the chosen event and assigns the result to $duration$. Further, the chosen node advances its clock by $duration$. Next, the algorithm records the event execution by adding the tuples $\langle node, event, start, duration \rangle$ and $\langle event, realTime \rangle$ to the $timedEvents$ and $realTimes$ queues respectively. When the stopping condition is satisfied, the simulation stops and returns $timedEvents$ and $realTimes$. Note that we can trivially ensure there is a pending event by adding a dummy timer that repeatedly fires at long periods. Similarly, we ensure the simulation terminates by encoding a timeout in the stopping condition.

Algorithm 2 Explorer

Input: System S
Input: Stopping condition C
Input: Set EDD of timing distributions
Input: Integer N
  $execs$: $\langle execution \rangle$ Queue
  $eventTimes$: $\langle event, realTime \rangle$ Queue
  for $i = 1$ to N **do**
    Reset system
    $\langle ex, times \rangle = \mathsf{Simulator}(S, C, EDD)$
    Add ex to $execs$
    Add each element of $times$ to $eventTimes$
  return $\langle execs, eventTimes \rangle$

Algorithm 2: Explorer implements a search of the space of executions, via repeated calls to Simulator. This algorithm takes four input parameters: a system to be analyzed, a stopping condition, a set of EDDs, and an integer N corresponding to the number of executions to be explored. The algorithm constructs two queues: (i) $execs$, which holds the explored executions, (ii) $eventTimes$, which holds the recorded times for different events along the explored executions. Both queues are initially empty. The algorithm iterates N times. In each iteration, it calls Simulator and adds the returned $execution$ to the $execs$ queue, and each event-(real)duration tuple to the $eventTimes$ queue. After the loop, it terminates and returns the set of observed executions $execs$ and event-duration tuples. Between each invocation of Simulator, the algorithm resets the system state, which includes tasks such as clearing the simulated network of messages, instantiating new nodes for the next execution, removing scheduled timers, and resetting the random number generator state. Since Simulator is randomized, each invocation of Simulator returns a (possibly) different execution.

Event Duration Independence. Our approach to simulating the passage of time assumes that within a node, event durations are independent, i.e. the durations of different events are uncorrelated. Different EDDs can be provided for nodes, modeling nodes at different speeds, but our implementation does not support temporally correlated durations (e.g. caused by resource competition from short-lived background processes). Nevertheless, this simple approach suffices to explore naturally occurring variations in event orderings and timings, unearthing many interesting performance bugs. We leave modeling potential temporal correlations to future work.

Deterministic Replay. A key property of our simulator is deterministic execution replay. By recording each event tuple during the search phase, the simulator has a path describing the complete execution, and can later replay this path by executing each event on the appropriate node at the time indicated. To provide consistent executions when replaying a path, the simulator must control all sources of non-determinism. We address non-determinism in event orderings by using a simulated source of time, as discussed above. In addition to the event orderings, real systems often make use of non-determinism *within* event handlers for randomized algorithms. When executing in the simulator, systems should use a deterministic simulated random number generator.

3.2 Finding Performance Bugs

Next, we describe each of the phases of our algorithm.

Phase 1a: Event Duration Training. First, we build EDDs that describe how long each type of event-transition takes to execute. To compute these distributions, we pick a value N such that N random executions "cover" all events (i.e. each event occurs in at least one of the N executions). If we later determine coverage was incomplete, we can either increase N and re-train, or substitute a similar event's distribution. We define a seed EDD for each event, where its duration is distributed uniformly over some fixed interval (such as 1-10ms). Next, we execute the Explorer on the system, the stopping condition, the seed EDDs, and N. When the search is complete, we discard the returned set of executions, and use the $\langle event, realTime\rangle$ tuples (returned in $realTimes$), to compute EDDs for each event using the cumulative frequency distribution.

Phase 1b: Performance Training. Next, we use the EDDs computed in phase 1a to quantify what should be deemed as anomalous performance. To this end, we determine the "typical" time it takes the system to carry out its high-level task, i.e. average time taken to reach the stopping condition. Concretely, we invoke the Explorer algorithm on the system, the stopping condition, the EDDs computed in phase 1a and another N. We discard the returned set of event-time tuples and use the returned set of executions $execs$ to compute the values of the first and third quartiles of the execution time. We use the definition of "mild outliers" [18] ($Q_3 + 1.5 \times (Q_3 - Q_1)$) to flag anomalous executions.

Phase 2: Anomaly Detection. In phase 2 we explore the space of behaviors to find executions with poor performance, i.e. whose execution time is anomalous as computed by phase 1b. Concretely, we run Explorer with the same parameters as before, except we use a large N, and terminate the search when we find an execution that falls outside the bounds determined in phase 1b. The use of different random number generator implementations allow various search algorithms to be employed. For example, our prior work on the MACE Model Checker (MACEMC) [13] used an iterative bounded depth-first-search generator for exhaustive testing, which is impractical when simulating microsecond-granularity timings. In our experience, a basic randomizing generator is sufficient to detect many performance bugs. We leave to future work further exploration of other generators such as a best-first generator.

Phase 3a: Divergence Detection. An anomalous execution can consist of tens of thousands of events. The prospect of sifting through all these events to find the performance bug would daunt the hardiest systems developer. In this phase, we analyze the execution to pinpoint the *divergence point*, the first event along the execution that leads to stable performance degradation. By doing so, the developer may skip past execution prefixes which may lead to good performance, focusing on the remainder.

Divergence detection is an adaptation of the technique for finding a critical transition pioneered in MACEMC. The insight behind the algorithm is as follows. Many performance problems are caused by corner-case race conditions which cause latent performance bugs. Prior to the race condition occurring in an execution, branching the execution and following a different path avoiding the race condition leads to a good execution. Thus we identify prefixes of the execution which have *not* experienced the race condition, and narrow down where the race condition takes place. Our experience indicates that knowing the divergence point can allow the programmer to ignore large portions of the execution trace: in one case nearly 62% of the 22000 events could be ignored, saving many hours of debugging time. More precision is not provided by this technique because what it identifies in the execution is where the race condition finished being enqueued onto the pending events list, not when it actually occurs.

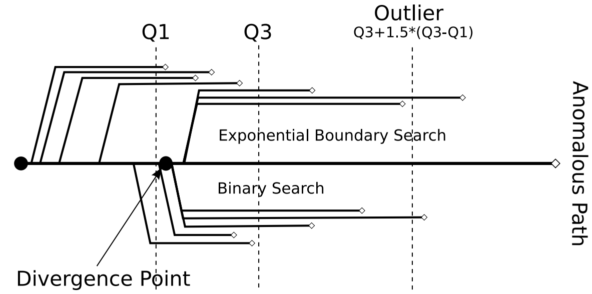

Figure 3: We first perform an exponential search (shown above the anomalous execution) to determine bounds for the divergence point, then a binary search (shown below the anomalous execution) to isolate the divergence point. Note that to avoid finding cases which are only slightly non-anomalous, any execution exceeding Q3 will be considered a performance failure.

As illustrated in Figure 3, we begin by initializing the *prefix* length to one event, replaying the portion of the path overlapping with the prefix, and then performing up to k random walks. If any of the random walks satisfies the timing constraint the race condition has not occurred, so we double the prefix length and repeat. Eventually, we reach a part of the path where none of the k random executions satisfy the timing constraint, and thus we have lower and upper bounds on the divergence point. Note that for the timing constraint we use the third quartile rather than the outlier threshold. Otherwise the analysis will find branches that perform only slightly under the outlier threshold, distracting rather than enhancing the debugging. Though this can further decrease the precision of the analysis, the goal is to confidently identify states which are safe to ignore. The algorithm's second phase isolates the divergence point by conducting a binary search of the lower and upper bounds. The algorithm terminates, producing an execution satisfying the timing constraint with the longest common prefix to the anomalous execution.

Phase 3b: Frequency Correlation. To further simplify the debugging process, we also collect event frequencies and the execution time for executions simulated during phases 2 and 3a. Computing this data is straightforward from the $eventTimes$ queue returned with each execution.

We automatically compute the correlation and scatter plot of each type of event with the execution time. This information can quickly direct developers' attention to the events whose presence or absence is related to the performance bug. In the three situations we have applied this technique thus far, after manually discarding event types that are trivially correlated with path length, such as periodic timer expiration, remaining high correlations were directly or indirectly related to the bug, drawing developer attention to important parts of the implementation.

An example of using event frequencies is shown in Figure 4. This plot is from the motivating example in Figure 1. It shows a correlation between the event described by "receiving a JoinReply and sending a Remove message" and the execution time. Had this tool been available when we first diagnosed this bug, we would have seen quickly that this event is correlated with longer paths, which would have helped us find the bug more quickly.

4. IMPLEMENTATION DETAILS

While the techniques behind MACEPC can be applied to arbitrary event-driven systems implementations and simulators, our implementation targets systems implemented using the MACE framework [12]. Focusing on MACE significantly simplified the effort in building MACEPC, particularly given MACEMC as a starting point. MACE structures each system component as a *service* implemented using C++ objects. Each service object is structured as a state machine whose states correspond to valuations of the object's member fields. Each event's transition is implemented as an atomically executed, non-blocking C++ method call, which can asynchronously send events to itself through timers, or non-local nodes through network messaging. Finally, MACE combines the high-level service object specifications with "scaffolding code" that handles event dispatch, serialization, callbacks, timers, *etc.* to generate C++ code that is ready to run on live networks.

In the remainder of this section, we describe the necessarily implementation changes to enable the implementation of MACEPC. Note that unlike many discrete event simulators, the goal of MACEPC is not the accurate simulation of the distributed system, but instead to be accurate enough to expose interesting performance bugs. This allows us to keep our event-simulator quite simple.

Scalable Network Simulation

Like other distributed systems simulators, MACEPC must simulate network delays. MACEPC seeks a middle-ground between using a complete network simulator modeling congestion, and using a simplistic model which only considers access link bandwidth. We also considered using measured delay distributions as for events, but found a simple model was sufficient to expose interesting bugs.

The time taken to send a message from a source to a receiver is computed as the sum of four factors: (i) a *propagation delay* fixed based on configuration, that models the delay to transfer data from source to destination, (ii) a *transmission delay* that models the bandwidth between the source and receiver and the number of simultaneous flows sharing the link, and (iii) a *random delay* drawn from a Pareto distribution, to model router queues.

By default, the propagation delay is set to 1ms, and the bandwidth between peers at 8000 Kbps. However, each of these is configurable for different node pairs. To simulate the common case where a node's first-hop link is the bottleneck, we count the number of active outgoing flows the node has to all other nodes above a certain size, and use this value to divide the available bandwidth. For example, if the bandwidth is configured at 8000 Kbps, and there are two simultaneous flows, the transmission delay for a message corresponds to the time it takes to send the message over a 4000 Kbps link. We assume that all flows with a certain threshold number of bytes in transmission are receiving their "fair share" from TCP. In particular, we exclude "light" flows from this equation, to avoid equally penalizing these flows. In our current implementation, this threshold is set to 300 bytes.

While the above does not perfectly mirror the behavior of networks, it allows MACEPC to simulate the network with enough efficiency and fidelity to unearth tricky performance bugs. We also validated that MACEPC could find a known topology-specific bug by configuring the latencies accordingly across system nodes.

Time-based Simulation

Since we were adapted MACEMC to build MACEPC, we had to add time to its execution model. In doing so, a pending event queue was added in place of querying simulators for possible events at each step. This modification had a dramatic effect on simulation complexity, enabling MACEPC to run simulations at much larger scale than is practical using MACEMC.

Random Number Generators

All non-determinism in the implementations is mapped to calls to the MACE random number generator library, including selection of which event to execute. As a result, we can explore different search techniques for the state space by implementing different random number generators. When we first developed MACEPC, it used the default MACEMC random number generator that conducted a bounded depth-first-search. However, the degree of randomness in simulating microsecond level times rendered the search ineffective. Next, we explored a search technique which emulated the other random number generator by picking a fixed number of random candidates to explore in every step rather than an exhaustive search. This was more practical, but added complexity without providing any sort of guarantees about coverage. Currently, we use a uniform random number generator, which provides state space sampling rather than exhaustive search. We leave as future work using a best-first random number generator that intentionally pushes the system into poor performing states.

Preparing a System for MACEPC

Finally, to use MACEPC to find performance bugs, the user must write: (i) one or more "test harnesses" or driver applications that are used by MACEPC to execute initialize and execute the application, and (ii) a stopping condition that is used by MACEPC to determine when the system has completed its task. For example, to find performance anomalies in a file distribution system, the user could write (i) a driver application in which one node publishes the file and the other nodes request the file, and (ii) a stopping condition that holds when each (receiver) node has finished downloading the file. The compiled driver application is linked with the MACE system object files and simulator specific MACE libraries for message queuing and delivery, system time, timer scheduling and random number generation, and MACEPC uses the resulting system to find performance bugs.

5. EXPERIENCES

We ran MACEPC on MACE implementations of BULLET′ [14], PASTRY [26], BAMBOO [24], CHORD [27], and a random tree protocol (RANDTREE) as described below, and found significant, previously unsolved performance issues with each. The following examples are intended to give an understanding of the types of bugs we were able to find and the utility of the tool. We try to describe only enough of the system being executed to understand the bugs described and the stopping conditions used.

All the systems we evaluated have been extensively tested in live runs, and PASTRY, BAMBOO, CHORD, and RANDTREE were tested for correctness using MACEMC. The BULLET′ implementation, though not checked with MACEMC, was significantly performance tuned as a major differentiator relative to contemporaneous systems.

The experiments below were run in a variety of network configurations. MACEPC ran on a single machine with an Intel Core2Duo processor at 3GHz with 4GB of RAM. RANDTREE was tested at small scale and yielded fast executions. BAMBOO was tested on 20-40 nodes, and the remaining systems were tested in configurations of 40-100 nodes. Note that it is impractical to perform any kind of exhaustive search using standard model checking techniques on systems of this size. Executions run in MACEPC took anywhere from 8 seconds to 3 minutes. This execution time is highly variable based on two primary factors: (1) the length of a path reaching the stopping condition, and (2) the per-event processing in MACEPC. For some systems, such as PASTRY, the stopping condition, a per-event processing task, has a complexity of $O(n^2 \log n)$ for n nodes, so paths take much longer to explore as the size of the configuration grows.

The time spent searching for anomalous executions varied depending on the number of paths MACEPC had to search before finding an anomaly, but the bugs we found appeared relatively early. The longest part of MACEPC execution was the divergence detection phase, which has to run $O(k \log s)$ steps, where k is the number of random paths at each step (a parameter impacting the error), and s is the depth of the divergence point. Anomaly analysis ranged from 1 hour to 22 hours to run. Though 22 hours is on the long side, the search was unattended, and did allow us to ignore nearly 62% of the 22000 events in the anomalous execution.

5.1 BULLET′

BULLET′ is a mesh-based, peer-to-peer file distribution protocol similar in functionality to BITTORRENT [1]. Each node contacts a source node, receives a set of initial peers to join, and then begins downloading a file in parallel from other nodes. Before discussing the bugs we found, there are a few relevant implementation details to discuss. First, BULLET′ uses two transports – one for data and a second for control messages. The data transport is configured to only buffer one block's worth of data at a time, while the control transport is configured to buffer an unbounded number of messages. The TCP transport in MACE has the ability to buffer messages so that it can send them asynchronously without using service resources. Limiting the buffer length allows BULLET′ to quickly react to changing network conditions and application requests, since buffered messages cannot be cancelled. BULLET′ actively detects when message blocks will not fit in the queue, and schedules a timer to send them later. However, "Diff" messages, or those that inform a peer about blocks a node possesses, are also sent over the data transport, and are not similarly protected against a full buffer. Second, BULLET′ is structured on top of RANSUB [15], a gossip protocol that periodically delivers a changing random subset of mesh participants to each node.

Stopping Condition. The stopping condition for BULLET′ is simple – all nodes should complete downloading the file.

Anomalies. In the first experiment with BULLET′, we used a setup of 100 nodes downloading a 20MB file. MACEPC found the first anomaly after 134 runs, representing an 11 second execution which exceeded the 9.5 second upper bound (as computed in Section 3.2). Each execution took approximately 18 seconds of real time, and terminated in between 56000-57000 simulator steps.

After examining the times that each individual node took in this execution, we determined that only one slow node limited overall system performance. Anomaly analysis showed that the discrepancy was based on the timing of one particular message the slow node sent to one of its peers. Upon further investigation, we realized that in the "good" execution, the slow node's message caused a Diff to be sent to it successfully, which happened to contain information about two blocks that were never successfully sent by any other node. The slow node was then able to request these blocks from this peer and complete the file download. In the anomalous execution, the message was timed such that the Diff message sent from the slow node's peer was dropped by a full transport. As a result, the slow node never learned about the two blocks, leaving it stuck since it did not know about any peers who had the missing blocks. Eventually, RANSUB delivered a new set of candidate nodes, and the slow node was able to join one of them and retrieve the missing blocks. However, waiting on RANSUB was responsible for the delay, causing the anomalous execution.

The second experiment with BULLET′ used only a 2MB file, but MACEPC found the second anomaly in less than 10 executions. In this case, the anomalous execution took around 5.5 seconds, whereas a normal execution took no more than 2 seconds of simulated time. To debug this case, we once again examined individual node completion times to learn that only a few nodes were slow. The first thing obvious from inspecting the slow nodes' logs was that no blocks were received until approximately 5 seconds into the run, then they quickly retrieved all the blocks. We found that the node did not acquire any peers for the first 5 seconds. The slow node did receive a list of candidate peers from the source when it joined. However, its attempts to join each of them failed because no candidate could accept another peer. Thus, the node waited 5 seconds for RANSUB to deliver it a set of new peers.

Improvements. To fix our first anomalous condition, we simply changed Diff messages to be sent over the control transport instead of the data transport. This eliminated the problem of Diffs being dropped by the transport, and ensured that all nodes received all intended Diff messages.

The second performance problem was based on the fact that when a node is rejected by potential peers, it could have to wait for RANSUB to deliver it new ones. To overcome this idle time, we changed the JoinReject message to contain the list of the rejecting node's peers. Then, when a node receives the JoinReject message, it has a set of other nodes it can try to join.

Note that in both cases, the overall system execution was *correct*. However, just as corner cases in execution can lead to errors, similar corner cases can lead to unexpectedly slow performance. Automated state space exploration techniques appear to be well suited to automatically find both types of conditions.

5.2 PASTRY

PASTRY is a well-known Distributed Hash Table (DHT) protocol that enables nodes to self-organize into a ring structure. Each node in the ring takes an address in a circular address space, and becomes responsible for the address space in the immediate vicinity of its own address. The PASTRY protocol organizes the ring to enable routing to any address using a path of no more than $\log(n)$ hops.

The primary functionality we are concerned with is the stabilization of the PASTRY network. PASTRY's performance can be measured by how long it takes nodes to finish their self-organization protocol. This protocol includes two basic components. The *active join component* allows a joining node to connect to an existing node and follows a protocol to find where in the network it should insert itself. A node N joins by routing a "join" message to its closest peer in the address space, who tells N of its address-space neighbors by sending N its "leafset". Along the way, it also gathers information about other nodes in the system. This protocol will execute correctly as long as only one node is joining at a time, and in the absence of departing nodes.

To handle multiple simultaneous node joins and departures, the *maintenance component* periodically exchanges leafset information with peers to ensure that each node's state is accurate. This protocol component can correct a wide variety of errors, mistakes, and dropped messages, and therefore mask many performance bugs.

Stopping Condition. The PASTRY stopping condition is the time all nodes' routing information has stabilized to a consistent state. This involves checking both that each node knows its immediate neighbors (important for correctness and fault tolerance), and that the routing distance between all pairs of nodes is logarithmic in the number of nodes.

Anomalies. During the training of PASTRY anomaly conditions, we stopped the training early. In the first 6 paths, the average execution times for the 40 pastry nodes (in simulated seconds) were:

$$(60.0073, 60.0061, 60.0085, 40.0051, 20.0369, 20.0313)$$

Variations of this magnitude were immediate indicators of a performance problem. We re-ran MACEPC having it save any execution that took longer than 50 (simulated) seconds. 22 of the first

40 paths took longer than 50 seconds, the longest taking 100.0044 seconds in 8848 simulator steps. All executions were within a second of a multiple of 20 seconds, a clue that the execution time was being dominated by the behavior of a system timer firing every 20 seconds.

Anomalies such as these were not unexpected. In earlier experiments, we had observed that the PASTRY implementation required either a long stabilization period after first being started, or an extended stagger-start period where nodes are slowly started over the first 30 seconds of the experiment. Running MACEMC over PASTRY did not flag these problems because PASTRY was still *eventually* reaching the stopping condition. We also did not attempt to extensively debug this performance problem in live executions because of the difficulties of debugging a distributed system, and the knowledge that we could just wait for it to stabilize before conducting other experiments using PASTRY.

Anomaly analysis on the longest path indicated that the performance had not diverged before step 5681. In both fast and slow executions, all nodes are trying to join at once. Since the bootstrap node does not add a joining node until *after* each joining node has confirmed that it is finished joining, multiple simultaneous joining nodes will initially believe they are in a 2-node ring with just them and the bootstrap node. (Each contacts the bootstrap node, who initially knows no one. It responds, telling them about just itself, and does not add them to its leafset until after they confirm they have finished joining.) Then all nodes nearly simultaneously announce themselves as new members of this small ring, largely oblivious to the other recently-arrived nodes.

In the ensuing mayhem of the active join protocol, the nodes closest in the address space to the bootstrap node are successful, while the state of other nodes further away depends on the precise order in which nodes are added and removed from the leafset of the bootstrap node. Unlucky nodes take one or more executions of the periodic maintenance protocol to finally correct their state. Each time the periodic protocol executes, a node will move k neighbors nearer to their correct position in the ring. If n nodes join simultaneously, and $n >> k$, this can take a very long time.

Improvements. The basic problem is that waiting 20 seconds for each execution of the periodic protocol is a huge performance penalty during ring construction. Scheduling the protocol more frequently would help, at the added cost of higher overhead in the common case (such fixes are only required during times of high node churn). An adaptive timer could be used, though its design would likely involve difficult and network-specific tuning parameters.

After exploring a range of options, we settled on the following solution. The problems essentially occur for nodes who are replaced in the bootstrap node's leafset but know nothing about nodes near them in the address space. Thus, we notify a node with the current leafset when *removing* it from the leafset. When it receives the leafset, it will be informed of several nodes which are closer to its correct place in the ring. This same information would be received at a later time when the maintenance protocol runs, but at that point the information will be more out-of-date and less relevant. It is important that the node get this particular version of the leafset, because the later version will only contain nodes close to the bootstrap node, not to the evicted node. After making these improvements, all execution times varied in length from 1.5-2 (simulated) seconds.

5.3 BAMBOO

BAMBOO is an enhancement to the original PASTRY protocol, designed to better handle network churn and cause lower network overheads than the very verbose PASTRY active join protocol. As such, the performance metrics of interest and the stopping condition are the same.

BAMBOO accomplishes its lower overhead by shortening the active join protocol to only contain information about the leafset, rather than other information gathered along the way. It also reduces the amount of state in the maintenance protocol, thus reducing overhead while still converging to good routing paths.

We used MACEPC on our BAMBOO implementation and found three interesting performance bugs. We describe two below:

- BAMBOO keeps track of successors and predecessors, but does not always know which one a node will be. If there is a poor balance of node identifiers, a node initially suspected to be a predecessor could later end up in the successor set. To accommodate this, there was code in BAMBOO to take a node pushed out of one set and check to see if it belongs in the other set. However, a bug caused the variables maintaining the boundary of the set to be updated *before* checking to see if the boundary belonged in the other set. Thus, this caused such nodes to be forgotten, and the problem to be later corrected by the maintenance protocol.
- Despite this active join change, unusual orders of joining nodes sometimes cause peers to not learn of each other. Again, this is because each peer first gathers information, then announces its liveness. This causes immediate neighbors to sometimes be wrong until the maintenance protocol executes. To fix this problem, we added an extra field to the "inform" message the BAMBOO implementation uses to confirm that it is alive. This additional field indicates whether the informant believes it is adjacent to the informed. If any informed node disagrees with this adjacency information, it triggers the maintenance protocol early to fill in gaps.

After implementing these fixes, MACEPC was unable to find any executions which take longer than 1 second. Encouraged by this improvement, we conducted a real evaluation similar to the one described in the BAMBOO paper [24] comparing the original MACE BAMBOO implementation with our fixed implementation. In this test, we ran a set of 300 instances of BAMBOO using ModelNet [28] on a cluster of 5 machines, each with 16 GB of RAM, dual, 64-bit quad-core Intel Xeon CPUs, and 4 Gigabit Ethernet connections, running Gentoo Linux 2.6.27. The experiment measures the consistency and latency of BAMBOO routing lookup information as the median session time of a node ranges from 84 seconds to 672 seconds. To measure consistency, every set of 10 nodes share a common lookup schedule, and lookups are considered consistent if a majority of responses indicate the same target node. The results are shown in Figures 5 and 6. The version produced by fixing the bugs found using MACEPC allowed us to deliver better consistency, eliminating roughly $15 - 30\%$ of inconsistent lookups, while also reducing latency by up to 15% under high degrees of churn. These results demonstrate that the bugs we fix using MACEPC translate to actual improvements in the protocols we test, not just arcane nuances tickled by rare execution paths.

5.4 CHORD and RANDTREE

We have also applied MACEPC to the MACE implementations of CHORD and RANDTREE, and have found performance bugs in each. These include the example bug illustrated in Figure 1. The CHORD bugs were similar in flavor to the PASTRY bugs, with similar solutions. The RANDTREE bugs were similar types of problems to the one in Figure 1, and caused the recovery protocol to correct them. The recovery protocol only executes every 10-60

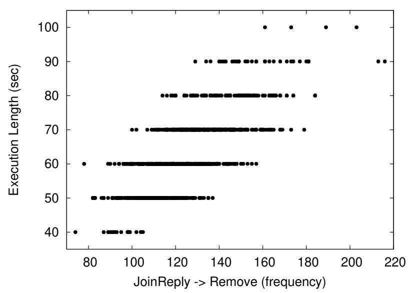

Figure 4: Positive correlation between execution length and erroneous event.

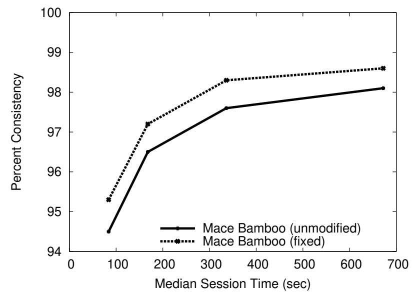

Figure 5: Percentage of lookups returning a consistent result.

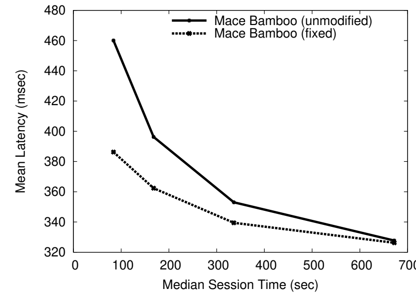

Figure 6: Mean latency of consistent lookups.

seconds in a typical installation, since it is only supposed to correct for network partitions, and not protocol bugs. Thus, the bugs in RANDTREE were responsible for very slow tree formation when many nodes join at once, and correcting them allowed these scenarios to proceed in under 1 (simulated) second rather than tens of seconds.

6. LIMITATIONS

While this approach to finding performance bugs has been successful and seems quite promising, it is not a panacea. We describe some limitations of our approach.

First, we run the real system under particular workloads. Thus, our system can only find performance bugs manifested by those particular workloads, though the use of randomized simulation allows us to exercise multiple behaviors for each workload.

Second, because our systematic exploration considers executions based on realistic distributions of event timings under a particular environment, it will not cover as many code paths as a traditional model checker [7, 13, 19, 22, 29]. This means it may have to be run separately under different deployment environments (e.g., different network conditions, etc.). The alternative approach of considering all possible performance conditions appears impractical.

Thus, our present implementation has not considered complicated execution search strategies or temporal correlations between individual event timings, because they have not yet been needed. The design supports these possibilities, and we anticipate that additional performance bugs may be isolated with additional detail.

7. RELATED WORK

MACEPC is related to several techniques for finding errors in software systems.

Fault Isolation. Our work is related to ideas in the fault isolation literature such as *delta debugging* [30] which systematically searches a space of possible faults to isolate the one that triggers a particular bug. A variant of this idea is *predicate switching* [31] which flips branches along a path in order to explore alternate executions, thereby isolating the branch that exposes a particular bug. These techniques focus on correctness problems, and find bugs that cause well defined “crashes"; in contrast, our work attempts to isolate performance problems by finding executions that take anomalously long.

Systematic Execution Exploration. MACEPC is closely related to tools which find correctness bugs by systematically executing different paths through unmodified implementations. VERISOFT views the entire system as several *processes* communicating through message queues, semaphores and other IPCs. It schedules these processes and traps calls that access shared resources. By choosing the process to execute at each such trap point, the scheduler can exhaustively explore all possible interleavings of the processes' executions using a *stateless search*, thereby finding a variety of errors. CMC [19], also directly executes the code and explores different executions by interposing at the OS scheduler level. MODIST [29] also directly executes the code, and is focused on transparent execution and checking of fail-stop and divergence errors. CHESS [22] improves the effectiveness of systematic exploration using iterative context bounding [20] and fair stateless search [21]. JAVAPATHFINDER [10] checks Java programs by interposing at the JVM level, in a manner analogous to CMC. MACEMC [13] combines VERISOFT-style stateless search with random walks to find *liveness* bugs. Furthermore, MACEMC uses a binary search over the erroneous execution to pinpoint the *critical transition* before which the system could have recovered to a live state, but after which the recovery becomes impossible. Our notion of a divergence point is an adaption that differs in that it is easier to tell whether a state is dead (i.e. can never recover to a live state) than whether performance has degraded. To be conservative, our technique finds the divergence point when paths begin to show signs of degradation. All the above ignore time and hence cannot be used to find errors related to performance anomalies.

Performance Tracking. PIP [23] is a concurrent, complementary technique for finding bugs in distributed systems. PIP is an annotation language and an expectation checker which can be applied to executions. PIP provides a way to visualize distributed path-flow in a system, and to write expectations to validate system paths. By writing a set of execution validators, the idea is that you can find performance bugs by looking at any non-validated paths. MACEPC is easier to use as (i) it does not require a live deployment of the system, (ii) it can automatically test a wide variety of executions, and (iii) it does not require careful manual examination of every possible distributed path-flow. X-TRACE [25] allows developers to better understand the performance of their system by using extensions to the existing protocol stack to trace the flow of messages across protocol layers, networks and applications. Like PIP, X-TRACE is focused on debugging particular live live executions, whereas MACEPC automatically finds executions with anomalous performance. Finally, TREND-PROF [9] allows users to measure the empirical computational complexity of implementations by plotting the performance of the system across a range of input sizes. Divergences in expected behavior can pinpoint bottlenecks in the code e.g. functions whose run-times grow faster than linearly with input size. It is not clear if such techniques can be adapted to the uncertain environment of distributed systems.

8. CONCLUSIONS

We have presented MACEPC, a technique that *finds* and *isolates* performance bugs in unmodified distributed systems code by searching the execution space for executions that perform far worse than is typical. MACEPC starts by training itself using the run-times of actual events from real executions. Next, MACEPC uses the distributions to explore a large number of executions, looking for executions that take abnormally long to complete. Finally, upon finding an anomalous execution, MACEPC carries out systematic search for the most similar execution that does not exhibit the performance bug. The two executions, along with an automatically identified divergence point—the step after which it becomes impossible for the execution to achieve acceptable performance—serve dually to direct the developer to a portion of the execution believed to contain the bug, and to attest that the bug does not occur before the divergence point. Further, event frequency data is available to the programmer, and its correlation with performance can help guide the developer to focus on relevant types of events. We have applied MACEPC to five mature systems, finding long-standing performance bugs in each. Relative to running experiments on nodes spread across the Internet, or even on a local-area network emulator, we have found that our performance checker significantly simplifies and speeds the task of performance debugging and does not require expensive, manually inserted logging that often obfuscates the underlying bug.

Acknowledgements. We would like to thank Amin Vahdat for useful discussions and feedback on the paper and techniques. This work was supported by the National Science Foundation under Grant Nos. CCF-0644361, CNS-0720802, and a gift from Microsoft Research.

9. REFERENCES

[1] Bittorrent. http://bitconjurer.org/BitTorrent.

[2] CADAR, C., DUNBAR, D., AND ENGLER, D. R. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI* (2008).

[3] CASTRO, M., DRUSCHEL, P., KERMARREC, A.-M., NANDI, A., ROWSTRON, A., AND SINGH, A. SplitStream: High-bandwidth content distribution in cooperative environments. In *SOSP* (2003).

[4] DABEK, F., COX, R., KAASHOEK, F., AND MORRIS, R. Vivaldi: A decentralized network coordinate system. In *SIGCOMM* (Portland, Oregon, 2004).

[5] ENGLER, D. R., CHEN, D. Y., AND CHOU, A. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *SOSP* (2001), pp. 57–72.

[6] GEELS, D., ALTEKAR, G., MANIATIS, P., ROSCOE, T., AND STOICA, I. Friday: Global comprehension for distributed replay. In *NSDI* (2007).

[7] GODEFROID, P. Model checking for programming languages using Verisoft. In *POPL* (1997).

[8] GODEFROID, P., KLARLUND, N., AND SEN, K. Dart: directed automated random testing. In *PLDI* (2005).

[9] GOLDSMITH, S., AIKEN, A., AND WILKERSON, D. S. Measuring empirical computational complexity. In *ESEC/SIGSOFT FSE* (2007), pp. 395–404.

[10] HAVELUND, K., AND PRESSBURGER, T. Model checking Java programs using Java Pathfinder. *Software Tools for Technology Transfer (STTT) 2(4)* (2000), 72–84.

[11] JANNOTTI, J., GIFFORD, D. K., JOHNSON, K. L., KAASHOEK, M. F., AND JAMES W. O'TOOLE, J. Overcast: Reliable Multicasting with an Overlay Network. In *OSDI* (2000).

[12] KILLIAN, C., ANDERSON, J. W., BRAUD, R., JHALA, R., AND VAHDAT, A. Mace: Language support for building distributed systems. In *PLDI* (2007).

[13] KILLIAN, C., ANDERSON, J. W., JHALA, R., AND VAHDAT, A. Life, death, and the critical transition: Detecting liveness bugs in systems code. In *NSDI* (2007).

[14] KOSTIĆ, D., BRAUD, R., KILLIAN, C., VANDEKIEFT, E., ANDERSON, J. W., SNOEREN, A. C., AND VAHDAT, A. Maintaining high bandwidth under dynamic network conditions. In *USENIX ATC* (2005).

[15] KOSTIĆ, D., RODRIGUEZ, A., ALBRECHT, J., BHIRUD, A., AND VAHDAT, A. Using Random Subsets to Build Scalable Network Services. In *USITS* (2003).

[16] LAMPORT, L. The part-time parliament. *ACM Trans. Comput. Syst. 16*, 2 (May 1998), 133–169.

[17] LUI, X., LIN, W., PAN, A., AND ZHANG, Z. Wids checker: Combating bugs in distributed systems. In *NSDI* (2007).

[18] MOORE, D. S., AND MCCABE, G. P. *Introduction to the Practice of Statistics*, 3rd ed. W.H. Freeman, New York, 1999.

[19] MUSUVATHI, M., PARK, D., CHOU, A., ENGLER, D., AND DILL, D. CMC: A pragmatic approach to model checking real code. In *OSDI* (2002).

[20] MUSUVATHI, M., AND QADEER, S. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI* (2007).

[21] MUSUVATHI, M., AND QADEER, S. Fair stateless model checking. In *PLDI* (2008).

[22] MUSUVATHI, M., QADEER, S., BALL, T., BASLER, G., NAINAR, P. A., AND NEAMTIU, I. Finding and reproducing heisenbugs in concurrent programs. In *OSDI* (2008).

[23] PATRICK REYNOLDS, CHARLES KILLIAN, J. L. W. J. C. M. M. A. S., AND VAHDAT, A. Pip: Detecting the unexpected in distributed systems. In *NSDI* (2006).

[24] RHEA, S., GEELS, D., ROSCOE, T., AND KUBIATOWICZ, J. Handling churn in a dht. In *USENIX ATC* (2004).

[25] RODRIGO FONSECA, GEORGE PORTER, R. H. K. S. S., AND STOICA, I. X-trace: A pervasive network tracing framework. In *NSDI* (2007).

[26] ROWSTRON, A., AND DRUSCHEL, P. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Middleware* (2001).

[27] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, F., AND BALAKRISHNAN, H. Chord: A scalable peer to peer lookup service for internet applications. In *SIGCOMM* (2001).

[28] VAHDAT, A., YOCUM, K., WALSH, K., MAHADEVAN, P., KOSTIĆ, D., CHASE, J., AND BECKER, D. Scalability and Accuracy in a Large-Scale Network Emulator. In *OSDI* (2002).

[29] YANG, J., CHEN, T., WU, M., XU, Z., LIU, X., LIN, H., YANG, M., LONG, F., ZHANG, L., AND ZHOU, L. MODIST: Transparent Model Checking of Unmodified Distributed Systems . In *NSDI* (2009).

[30] ZELLER, A. Yesterday, my program worked. today, it does not. why? In *ESEC / SIGSOFT FSE* (1999), pp. 253–267.

[31] ZHANG, X., GUPTA, N., AND GUPTA, R. Locating faults through automated predicate switching. In *ICSE* (New York, NY, USA, 2006), ACM, pp. 272–281.