X-ray: Automating Root-Cause Diagnosis of Performance Anomalies in Production Software

Mona Attariyan[†][*], *Michael Chow*[†], *and Jason Flinn*[†]
University of Michigan[†]   *Google, Inc.*[*]

Abstract

Troubleshooting the performance of production software is challenging. Most existing tools, such as profiling, tracing, and logging systems, reveal *what* events occurred during performance anomalies. However, users of such tools must infer *why* these events occurred; e.g., that their execution was due to a root cause such as a specific input request or configuration setting. Such inference often requires source code and detailed application knowledge that is beyond system administrators and end users.

This paper introduces *performance summarization*, a technique for automatically diagnosing the root causes of performance problems. Performance summarization instruments binaries as applications execute. It first attributes performance costs to each basic block. It then uses dynamic information flow tracking to estimate the likelihood that a block was executed due to each potential root cause. Finally, it summarizes the overall cost of each potential root cause by summing the per-block cost multiplied by the cause-specific likelihood over all basic blocks. Performance summarization can also be performed differentially to explain performance differences between two similar activities. X-ray is a tool that implements performance summarization. Our results show that X-ray accurately diagnoses 17 performance issues in Apache, lighttpd, Postfix, and PostgreSQL, while adding 2.3% average runtime overhead.

1 Introduction

Understanding and troubleshooting performance problems in complex software systems is notoriously challenging. When a system does not perform as expected, system administrators and end users have few options. Explicit error messages are often absent or misleading [58]. Profiling and monitoring tools may reveal symptoms such as heavy usage of a bottleneck resource, but they do not link symptoms to root causes. Interpretation of application logs often requires detailed knowledge of source code or application behavior that is beyond a casual user. Thus, it is unsurprising that up to 20% of misconfigurations submitted for developer support are those that result in severe performance degradation [58] (the authors of this study speculate that even this number is an underestimate).

Why is troubleshooting so challenging for users? The most important reason is that current tools only solve half the problem. Troubleshooting a performance anomaly requires determining *why* certain events, such as high latency or resource usage, happened in a system. Yet, most current tools, such as profilers and logging, only determine *what* events happened during a performance anomaly. Users must manually infer the root cause from observed events based upon their expertise and knowledge of the software. For instance, a logging tool may detect that a certain low-level routine is called often during periods of high request latency, but the user must then infer that the routine is called more often due to a specific configuration setting. For administrators and end users who do not have intimate knowledge of the source code, log entries may be meaningless and the inference to root causes may be infeasible.

In this paper, we introduce a new tool, called X-ray, that helps users troubleshoot software systems without relying on developer support. X-ray focuses on attributing performance issues to root causes under a user's control, specifically configuration settings and program inputs. Why these causes? Numerous studies have reported that configuration and similar human errors are the largest source of errors in deployed systems [10, 11, 24, 25, 30, 32, 34, 58], eclipsing both software bugs and hardware faults. Further, errors such as software bugs cannot be fixed by end users alone.

X-ray does not require source code, nor does it require specific application log messages or test workloads. Instead, it employs binary instrumentation to monitor applications as they execute. It uses one of several metrics (request latency, CPU utilization, file system activity, or network usage) to measure performance costs and outputs a list of root causes ordered by the likelihood that each cause has contributed to poor performance during the monitored execution. Our results show that X-ray often pinpoints the true root cause by ranking it first out of 10s or 100s of possibilities. This is ideal for casual users and system administrators, who can now focus their troubleshooting efforts on correcting the specific input and parameters identified by X-ray.

X-ray introduces the technique of *performance summarization*. This technique first attributes performance costs to very fine-grained events, namely user-level instructions and system calls executed by the application.

---

[*]This work was done when Mona Attariyan attended Michigan.

Then, it uses dynamic information flow analysis to associate each such events with a ranked list of probable root causes. Finally, it summarizes the cost of each root cause over all events by adding the products of the per-event cost and an estimate of the likelihood that the event was caused by the root cause in question. The result is a list of root causes ordered by performance costs.

As described so far, performance summarization reveals which root causes are most costly during an entire application execution. For severe performance issues, such root causes are likely the culprit. However, some performance issues are more nuanced: they may occur only during specific time periods or affect some application operations but not others. Hence, X-ray provides several scoping options. Users may analyze performance during specific time periods, or they may look at a causal path such as a server's processing of a request.

X-ray also supports nuanced analysis via two additional summarization modes. In *differential performance summarization*, X-ray compares the execution of two similar operations and explains why their performance differs. For example, one can understand why two requests to a Web server took different amounts of time to complete even though the requested operations were identical. Differential performance analysis identifies branches where the execution paths of the requests diverge and assigns the performance difference between the two branch outcomes to the root causes affecting the branch conditionals. *Multi-input differential summarization* compares a potentially large number of similar operations via differential analysis and outputs the result as either a ranked list or a graphical explanation.

X-ray is designed to run in production environments. It leverages prior work in deterministic replay to offload the heavyweight analysis from the production system and execute it later on another computer. X-ray splits its functionality by capturing timing data during recording so that the data are not perturbed by heavyweight analysis. X-ray's replay implementation is flexible: it allows insertion of dynamic analysis into the replayed execution via the popular Pin tool [28], but it also enables low-overhead recording by not requiring the use of Pin or instrumentation code during recording.

Thus, this paper contributes the following:

- A demonstration that one can understand why performance issues are occurring in production software without source code, error and log messages, controlled workloads, or developer support.
- The technique of performance summarization, which attributes performance costs to root causes.
- The technique of differential performance summarization for understanding why two or more similar events have different performance.
- A deterministic replay implementation that enables both low-overhead recording and use of Pin binary

instrumentation during replay.

Our evaluation reproduces and analyzes performance issues in Apache, lighttpd, Postfix, and PostgreSQL. In 16 of 17 cases, X-ray identifies a true root cause as the largest contributor to the performance problem; in the remaining case, X-ray ranks one false positive higher than the true root causes. X-ray adds only an average overhead of 2.3% on the production system because the bulk of its analysis is performed offline on other computers.

2 Related work

Broadly speaking, troubleshooting has three steps: detecting the problem, identifying the root cause(s), and solving the problem. X-ray addresses the second step.

Profilers [8, 13, 29, 35, 42, 45, 53], help detect a performance problem (the first step) and identify symptoms associated with the problem (which assists with the second step). They reveal *what* events (e.g., functions) incur substantial performance costs, but their users must manually infer *why* those events executed. Unlike X-ray, they do not associate events with root causes.

Similarly, most tools that target the second step (identifying the root cause) identify events associated with performance anomalies but do not explain why those events occur. Many such tools observe events in multiple components or protocol layers and use the observed causal relationships to propagate and merge performance data. X-trace [22] observes network activities across protocols and layers. SNAP [59] profiles TCP statistics and socket-call logs and correlates data across a data center. Aguilera *et al*. [1] infer causal paths between application components and attribute delays to specific nodes. Pinpoint [15, 16] traces communication between middleware components to infer which components cause faults and the causal paths that link black-box components. These tools share X-ray's observation that causality is a powerful tool for explaining performance events. However, X-ray distinguishes itself by observing causality *within* application components using dynamic binary instrumentation. This lets X-ray observe the relationship between component inputs and outputs. In contrast, the above tools only observe causality external to application components unless developers annotate code.

Other tools build or use a model of application performance. Magpie [7] extracts the component control flow and resource consumption of each request to build a workload model for performance prediction. Magpie's per-request profiling can help diagnose potential performance problems. Even though Magpie provides detailed performance information to manually infer root causes, it still does not automatically diagnose why the observed performance anomalies occur. Magpie uses schemas to determine which requests are being executed by high-level components; X-ray uses data and control flow analysis to map requests to lower-level events (instructions

and system calls) without needing schemas from its user.

Cohen *et al*. [19] build models that correlate system-level metrics and threshold values with performance states. Their technique is similar to profiling in that it correlates symptoms and performance anomalies but does not tie anomalies to root causes.

Many systems [14, 20, 60, 61] tune performance by injecting artificial traffic and using machine learning to correlate observed performance with specific configuration options. Unlike X-ray, these tools limit the number of options analyzed to deal with an exponential state space. Spectroscope [46] diagnoses performance changes by comparing request flows between two executions of the same workload. Kasick *et al*. [26] compare similar requests to diagnose performance bugs in parallel file systems. All of the above systems do not monitor causality within a request, so they must hold all but a single variable constant to learn how that variable affects performance. In practice, this is difficult because minor perturbations in hardware, workload, etc. add too much noise. In contrast, X-ray can identify root causes even when requests are dissimilar because it observes how requests diverge at the basic-block level.

Several systems are holistic or address the third step (fixing the problem). PeerPressure [54] and Strider [55] compare Windows registry state on different machines. They rely on the most common configuration states being correct since they cannot infer why a particular configuration fails. Chronus [56] compares configuration states of the same computer across time. AutoBash [50] allows users to safely try many potential configuration fixes.

X-ray uses a taint tracking [33] implementation provided by ConfAid [6] to identify root causes. ConfAid was originally designed to debug program failures by attributing those failures to erroneous configuration options. X-ray re-purposes ConfAid to tackle performance analysis. X-ray might instead have used other methods for inferring causality such as symbolic execution [12]. For instance, S2E [17] presented a case study in which symbolic execution was used to learn the relationship between inputs and low-level events such as page faults and instruction counts. Our decision to use taint tracking was driven both by performance considerations and our desire to work on COTS (common-off-the-shelf) binaries.

X-ray uses deterministic record and replay. While many software systems provide this functionality [2, 18, 21, 23, 36, 49, 52], X-ray's implementation has the unique ability to cheaply record an uninstrumented execution and later replay the execution with Pin.

3   X-ray overview

X-ray pinpoints why a performance anomaly, such as high request latency or resource usage, occurred. X-ray targets system administrators and other end users, though its automated inference should also prove useful to developers. Most of our experience to date comes from troubleshooting network servers, but X-ray's design is not limited to such applications.

X-ray does not require source code because it uses Pin [28] to instrument x86 binaries. X-ray users specify which files should be treated as configuration or input sources for an application. X-ray also treats any data read from an external network socket as an input source. As data from such sources are processed, X-ray recognizes configuration tokens and other root causes through a limited form of binary symbolic execution.

An X-ray user first records an interval of software execution. Section 6.5 shows that X-ray has an average recording overhead of 2.3%. Thus, a user can leave X-ray running on production systems to capture rare and hard-to-reproduce performance issues. Alternatively, X-ray can be used only when performance issues exhibit. X-ray defers heavyweight analysis to later, deterministically equivalent re-executions. This also allows analysis to be offloaded from a production system. Because X-ray analysis is 2–3 orders of magnitude slower than logging, we envision that only the portions of logs during which performance anomalies were observed will be analyzed.

For each application, an X-ray user must specify configuration sources such as files and directories, as well as a filter that determines when a new request begins.

For each analysis, an X-ray user selects a cost metric. X-ray currently supports four metrics: execution latency, CPU utilization, file system usage, and network use. X-ray also has a flexible interface that allows the creation of new metrics that depends on either observed timings or the instructions and system calls executed.

A user also specifies which interval of execution X-ray should analyze. The simplest method is to specify the entire recorded execution. In this case, X-ray returns a list of root causes ordered by the magnitude of their effect on the chosen cost metric. In our experience with severe performance issues, examining the entire execution interval typically generates excellent results.

However, some performance issues are nuanced. An issue may only occur during specific portions of a program's execution, or the problem may affect the processing of some inputs but not others. Therefore, X-ray allows its users to target the analysis scope. For instance, a user can specify a specific time interval for analysis, such as a period of high disk usage.

Alternatively, X-ray can analyze an application as it handles one specific input, such as a network request. X-ray uses both causal propagation through IPC channels and flow analysis to understand which basic blocks in different threads and processes are processing the input. It performs its analysis on only those basic blocks.

A user may also choose to compare the processing of two different inputs. In this case, X-ray does a differential performance summarization in which it first identifies the branches where the processing of the inputs di-

verged and then calculates the difference in performance caused by each divergence. We expect users to typically select two similar inputs that differ substantially in performance. However, our results show that X-ray provides useful data even when selected inputs are very dissimilar.

Finally, a user may select multiple inputs or all inputs received in a time period and perform an n-way differential analysis. In this case, X-ray can either return a ranked list of the root causes of pairwise divergences over all such inputs, or it can display the cost of divergences as a flow graph. We have found this execution mode to be a useful aid for selecting two specific requests over which to perform a more focused differential analysis.

Executions recorded by X-ray can be replayed and analyzed multiple times. Thus, X-ray users do not need to know which cost metrics and analysis scopes they will use when they record an execution.

4 Building blocks

X-ray builds on two areas of prior work: dynamic information flow analysis and deterministic record and replay. For each building block, we first describe the system on which we built and then highlight the most substantial modifications we have made to support X-ray.

4.1 Dynamic information flow analysis

4.1.1 Background

X-ray uses taint tracking [33], a form of dynamic information flow analysis, to determine potential root causes for specific events during program execution. It uses ConfAid [6] for this purpose.

ConfAid reports the potential root cause of a program failure such as a crash or incorrect output. It assigns a unique taint identifier to registers and memory addresses when data is read into the program from configuration files. It identifies specific configuration tokens through a rudimentary symbolic execution that only considers string data and common (glibc) functions that compare string values. For instance, if data read from a configuration file is compared to "FOO", then ConfAid associates that data with token FOO.

As the program executes, ConfAid propagates taint identifiers to other locations in the process's address space according to dependencies introduced via data and control flow. ConfAid analyzes both direct control flow (values modified by instructions on the taken path of a branch depend on the branch conditional) and implicit control flow (values that would have been modified by instructions on paths not taken also depend on the branch conditional). Rather than track taint as a binary value, ConfAid associates a weight with each taint identifier that represents the strength of the causal relationship between the tainted value and the root cause. When Conf-Aid observes the failure event (e.g., a bad output), it outputs all root causes on which the current program con-

trol flow depends, ordered by the weight of that dependence. ConfAid employs a number of heuristics to estimate and limit causality propagation. For instance, data flow propagation is stronger than direct control flow, and both are stronger than indirect control flow. Also, control flow taint is aged gradually (details are in [6]).

4.1.2 Modifications for X-Ray

One of the most important insights that led to the design of X-ray is that the marginal effort of determining the root cause of all or many events in a program execution is not substantially greater than the effort of determining the root cause of a single event. Because a taint tracking system does not know a-priori which intermediate values will be needed to calculate the taint of an output, it must calculate taints for *all* intermediate values. Leveraging this insight, X-ray differs from ConfAid in that it calculates the control flow taint for the execution of every basic block. This taint is essentially a list of root causes that express which, if any, input and configuration values caused the block to be executed; each root cause has a weight, which is a measure of confidence.

We modified ConfAid to analyze multithreaded programs. To limit the scope of analysis, when X-ray evaluates implicit control flow, it only considers alternative paths within a single thread. This is consistent with ConfAid's prior approach of bounding the length of alternate paths to limit exponential growth in analysis time. We also modified ConfAid to taint data read from external sources such as network sockets in addition to data read from configuration files. Finally, we modified ConfAid to run on either a live or recorded execution.

X-ray uses the same weights and heuristics for taint propagation that are used by ConfAid. We performed a sensitivity analysis, described in Section 6.4, on the effect of varying the taint propagation weights—the results showed that the precise choice of weights has little effect on X-ray, but the default ConfAid weights led to slightly more accurate results than other weights we examined.

4.2 Deterministic record and replay

X-ray requires deterministic record and replay for two reasons. First, by executing time-consuming analysis on a recording rather than a live execution, the performance overhead on a production system can be reduced to a few percent. Second, analysis perturbs the timing of application events to such a large degree that performance measurements are essentially meaningless. With deterministic replay, X-ray monitors timing during recording when such measurements are not perturbed by analysis, but it can still use the timing measurements for analysis during replay because the record and the replay are guaranteed to execute the same instructions and system calls.

4.2.1 Background

Deterministic replay is well-studied; many systems record the initial state of an execution and log all non-

deterministic events that occur [2, 9, 21, 36, 49, 52, 57]. They reproduce the same execution, possibly on another computer [18], by restoring the initial state and supplying logged values for all non-deterministic events.

X-ray implements deterministic record and replay by modifying the Linux kernel and glibc library. It can record and replay multiple processes running on one or more computers. For each process, X-ray logs the order of and values returned by system calls and synchronization operations. It also records the timing of signals.

To record and replay multithreaded programs, one must also reproduce the order of all data races [44]. X-ray uses profiling to detect and instrument racing instructions. We execute an offline data race detector [51] on recorded executions. This race detector follows the design of DJIT+ [41]; it reports all pairs of instructions that raced during a recorded execution without false positives or false negatives. X-ray logs the order of the racing instructions during later recordings of the application. If no new data races are encountered after profiling, deterministic replay of subsequent executions is guaranteed. In the rare case where a new race is encountered, we add the racing instructions to the set of instrumented instructions on subsequent runs. Since the vast majority of data races do not change the order or result of logged operations [51], X-ray can often analyze executions with previously unknown data races. X-ray could also potentially search for an execution that matches observed output [2, 36].

4.2.2 Modifications for X-ray

X-ray has a custom replay implementation because of our desire to use Pin to insert binary instrumentation into replayed executions. The simplest strategy would have been to implement record and replay with Pin itself [37]. However, we found Pin overhead too high; even with zero instrumentation, just running applications under Pin added a 20% throughput overhead for our benchmarks.

To reduce overhead, X-ray implements record and replay in the Linux kernel and glibc. Thus, Pin is only used during offline replay. This implementation faces a substantial challenge: from the point of view of the replay system, the replayed execution is not the same as the recorded execution because it contains additional binary instrumentation not present during recording. While Pin is transparent to the application being instrumented, it is *not* transparent to lower layers such as the OS.

X-ray's replay system is *instrumentation-aware*; it compensates for the divergences in replayed execution caused by dynamic instrumentation. Pin makes many system calls, so X-ray allocates a memory area that allows analysis tools run by Pin to inform the replay kernel which system calls are initiated by the application (and should be replayed from the log) and which are initiated by Pin or the analysis tool (and should execute normally).

X-ray also compensates for interference between re-sources requested by the recorded application and resources requested by Pin or an analysis tool. For instance, Pin might request that the kernel `mmap` a free region of memory. If the kernel grants Pin an arbitrary region, it might later be unable to reproduce the effects of a recorded application `mmap` that returns the same region. X-ray avoids this trap by initially scanning the replay log to identify all regions that will be requested by the recorded application and pre-allocating them so that Pin does not ask for them and the kernel does not return them. X-ray also avoids conflicts for signal handlers, file handles, and System V shared memory identifiers.

Finally, the replay system must avoid deadlock. The replay system adds synchronization to reproduce the same order of system calls, synchronization events, and racing instructions seen during recording. Pin adds synchronization to ensure that application operations such as memory allocation are executed atomically with the Pin code that monitors those events. X-ray initially deadlocked because it was unaware of Pin locking. To compensate, X-ray now only blocks threads when it knows Pin is not holding a lock; e.g., rather than block threads executing a system call, it blocks them prior to the instruction that follows the system call.

5   Design and implementation

X-ray divides its execution into online and offline phases. The offline phase is composed of multiple replayed executions. This design simplifies development by making it easy to compose X-ray analysis tools out of modular parts.

5.1   Online phase

Since the online phase of X-ray analysis runs on a production system, X-ray uses deterministic record and replay to move any activity with substantial performance overhead to a subsequent, offline phase. The only online activities are gathering performance data and logging system calls, synchronization operations and known data races.

X-ray records timestamps at the entry and exit of system calls and synchronization operations. It minimizes overhead by using the x86 timestamp counter and writing timestamps to the same log used to store non-deterministic events. The number of bytes read or written during I/O is returned by system calls and hence captured as a result of recording sources of non-determinism.

5.2   First offline pass: Scoping

The first offline pass maps the scope of the analysis selected by an X-ray user to a set of application events. While X-ray monitors events at the granularity of user-level instructions and system calls, it is sufficient to identify only the basic blocks that contain those events since the execution of a basic block implies the execution of all events within that block.

A user may scope analysis to a time interval or to the processing of one of more inputs. If the user specifies a time interval, X-ray includes all basic blocks executed by any thread or process within that interval. If the user scopes the analysis to one or more inputs, X-ray identifies the set of basic blocks that correspond to the processing of each input via *request extraction*.

5.2.1 Request extraction

Request extraction identifies the basic block during which each request (program input) was processed. For the applications we have examined to date, inputs are requests received over network sockets. However, the principles described below apply to other inputs, such as those received via UI events.

Since the notion of a request is application-dependent, X-ray requires a filter that specifies the boundaries of incoming requests. The filter is a regular expression that X-ray applies to all data read from *external sockets*, which we define to be those sockets for which the other end point is not a process monitored by X-ray. For instance, the Postfix filter looks for the string HELO to identify incoming mail. Only one filter must be created for each protocol (e.g., for SMTP or HTTP).

Request extraction identifies the causal path of each request from the point in application execution when the request is received to the point when the request ends (e.g., when a response is sent). X-ray supports two methods to determine the causal path of a request within process execution. These methods offer a tradeoff between generality and performance. Both are implemented as Pin tools that are dynamically inserted into binaries.

The first method is designed for simple applications and multi-process servers. It assumes that a process handles a single request at a time, but it allows multiple processes to concurrently handle different requests (e.g., different workers might simultaneously process different requests). When a new request is received from an external socket, X-ray taints the receiving process with a unique identifier that corresponds to the request. X-ray assumes that the process is handling that request until the process receives a message that corresponds to a different request or until the request ends (e.g., when the application closes the communcation socket). A new taint may come either from an external source (in which case, it is detected by the input data matching the request filter), or it may come from an internal source (another process monitored by X-ray), in which case the request taint is propagated via the IPC mechanism described below.

The second method directly tracks data and control flow taint. When a request is received from an external socket, X-ray taints the return codes and data modified by the receiving system call with the request identifier. X-ray propagates taint within an address space as a process executes. It assigns each basic block executed by the process to at most one request based on the control flow taint of the thread at the time the basic block is executed. Untainted blocks are not assigned to a request. A block tainted by a single identifier is assigned to request corresponding to that identifier. A block tainted by multiple identifiers is assigned to the request whose taint identifier has the highest weight; if multiple identifiers have the same weight, the block is assigned to the request that was received most recently.

Comparing the two methods, the first method has good performance and works well for multi-process servers such as Postfix and PostgreSQL. However, it is incapable of correctly inferring causal paths for multithreaded applications and event-based applications in which a single process handles multiple requests concurrently. The second method handles all application types well but runs slower than the first method. X-ray uses the second method by default, but users may select the first method for applications known to be appropriate.

Request extraction outputs a list of the basic blocks executed by each request. Each block is denoted by a <id, address, count> tuple. The *id* is the Linux identifier of the thread/process during recording, *address* is the first instruction of the block in the executable, and *count* is the number of instructions executed by the process prior to the first instruction of the basic block. Thus, *count* differentiates among multiple dynamic executions of a static basic block. Since deterministic replay executes exactly the same sequence of application instructions, the *count* of each block matches precisely across multiple replays and, thus, serves as a unique identifier for the block during subsequent passes.

5.2.2 Inter-process communication

Replayed processes read recorded data from logs rather than from actual IPC channels. X-ray establishes separate mechanisms, called *side channels*, to communicate taint between processes and enforce the same causal ordering on replayed processes that was observed during the original recording. For instance, a process that blocked to receive a message on a socket during recording will block on the side channel during replay to receive the taint associated with the message.

Side channels propagate taint from one address space to another. X-ray supports several IPC mechanisms including network and local sockets, files, pipes, signals, fork, exit, and System V semaphores. During replay, when a recorded system call writes bytes to one of these mechanisms, X-ray writes the data flow taint of those bytes to the side channel. X-ray merges that taint with the control flow taint of the writing thread. Even mechanisms that do not transfer data (e.g., signals) still transfer control flow taint (e.g., the control flow of the signal handler is tainted with the control flow taint of the signaler).

When replay is distributed, one computer acts as the replay master. Processes running on other computers register with the master; this allows each replay process to
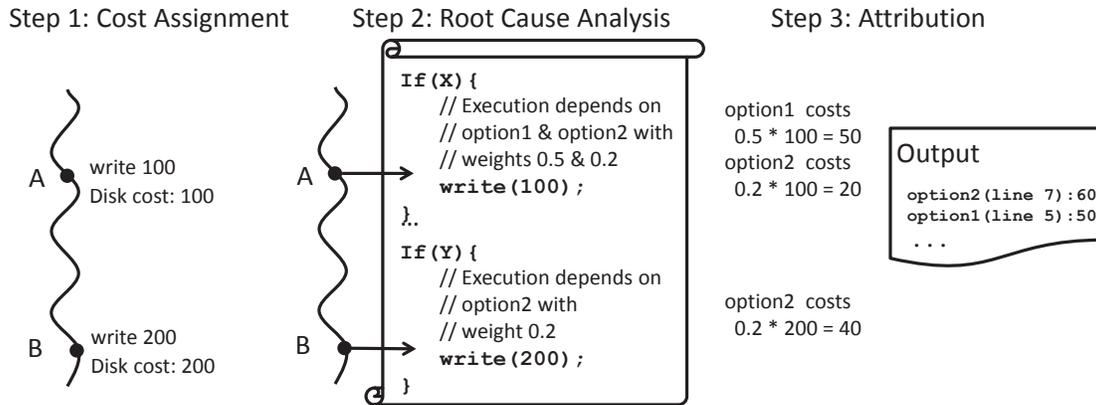
Step 1: Cost Assignment   Step 2: Root Cause Analysis   Step 3: Attribution

Figure 1. Example of performance summarization

determine which sockets are external and which connect to other replay processes. For simplicity, all side channels pass through the master, so replay processes on other computers read to and write from side channels by making RPCs to a server running on the master.

5.2.3 Attributing performance costs

During the first pass, X-ray also attributes performance costs to all *events* (application instructions and system calls executed) within the chosen scope. As a performance optimization, all events within a single basic block are attributed en masse because all are executed if the block is executed.

Recall that X-ray users may currently choose latency, CPU utilization, file system use, or network throughput as cost metrics. The latency of each system call and synchronization operation is recorded during online execution. X-ray attributes the remaining latency to user-level instructions. From the recorded timestamps in the log, it determines the time elapsed between each pair of system calls and/or synchronization events. X-ray dynamic instrumentation counts the number of user-level instructions executed in each time period. It divides the two values to estimate the latency of each instruction.

To calculate CPU utilization, X-ray counts the instructions executed by each basic block. To calculate file system and network usage, it observes replayed execution to identify file descriptors associated with the resource being analyzed. When a system call accesses such descriptors, X-ray attributes the cost of the I/O operation to the basic block that made the system call.

5.3 Second pass: performance summarization

Performance summarization, in which costs are attributed to root causes, is performed during the second execution pass. X-ray currently supports three modes: (1) basic summarization, which computes the dominant root causes over some set of input basic blocks, (2) differential summarization, which determines why the processing of one input had a different cost than the processing

of another input, and (3) multi-request differential summarization, which computes the differential cost across three or more inputs.

5.3.1 Basic performance summarization

Basic performance summarization individually analyzes the per-cause performance cost and root cause of all events. It then sums the per-event costs to calculate how much each root cause has affected application performance.

Figure 1 shows how basic performance summarization works. In the first pass, X-ray determines which basic blocks are within the analysis scope and assigns a performance cost to the events in each block. In the second pass, X-ray uses taint tracking to calculate a set of possible root causes for the execution of each such block. Essentially, this step answers the question: "how likely is it that changing a configuration option or receiving a different input would have prevented this block from executing?" X-ray uses ConfAid to track taints as weights that show how strongly a particular root cause affects why a byte has its current value (data flow) or why a thread is executing the current instruction (control flow).

X-ray next attributes a per-block cost to each root cause. This attribution is complicated by the fact that ConfAid returns only an ordered list of potential root causes. Weights associated with causes are relative metrics and do not reflect the actual probability that each cause led to the execution of a block. We considered several strategies for attribution:

- **Absolute weight.** The simplest strategy multiplies each per-cause weight by the per-block performance cost. This is an intuitive strategy since X-ray, like ConfAid, aims only to achieve a relative ranking of causes.

- **Normalized weight.** The weights for a block are normalized to sum to one before they are multiplied by the performance cost. This strategy tries to calculate an absolute performance cost for each cause. However, it may strongly attribute a block to

---

a root cause in cases where the block's execution is likely not due to *any* root cause.

- **Winner-take-all.** The entire per-block cost is attributed to the highest ranking cause (or equally shared in the case of ties).
- **Learned weights.** Based on earlier ConfAid results [6], we calculate the probability that a cause ranked 1st, 2nd, 3rd, etc. is the correct root cause. We use these probabilities to weight the causes for each block according to their relative rankings. Note that the benchmarks used for deciding these weights are completely disjoint from the benchmarks used in this paper.

Based on the sensitivity study reported in Section 6.4, we concluded that X-ray results are robust across all attribution strategies that consider more than just the top-ranked root cause. X-ray uses the absolute weight strategy by default because it is simple and it had slightly better results in the study.

X-ray calculates the total cost for each root cause by summing the per-block costs for that cause over all basic blocks within the analysis scope; e.g., in Figure 1, the per-block costs of `option2` are 20 and 40, so its total cost is 60). X-ray outputs potential root causes as a ranked list ordered by cost; each list item shows the token string, the config file or input source, the line number within the file/source, and the total cost.

5.3.2 Differential performance summarization

Differential performance summarization compares any two executions of an application activity, such as the processing of two Web requests. Such activities have a common starting point (e.g., the instruction receiving the request), but their execution paths may later diverge.

Figure 2 shows an example of differential performance summarization. X-ray compares two activities by first identifying all points where the paths of the two executions diverge and merge using longest common sub-sequence matching [31]. In the figure, the execution paths of the activities are shown by the solid and dashed lines, and the conditional branches where the paths diverge are denoted as C1 and C2.

X-ray represents the basic blocks that processed each request as a string where each static basic block is a unique symbol. Recorded system calls and synchronization operations give a partial order over blocks executed by multiple threads and processes. Any blocks not so ordered executed concurrently; they do not contain racing instructions. X-ray uses a fixed thread ordering to generate a total order over all blocks (the string) that obeys the recorded partial order. The matching algorithm then identifies divergence and merge points.

X-ray uses taint tracking to evaluate why each divergence occurred. It calculates the taint of the branch conditional at each divergence point. Note that since X-ray uses dynamic analysis, loops are simply treated as a se-
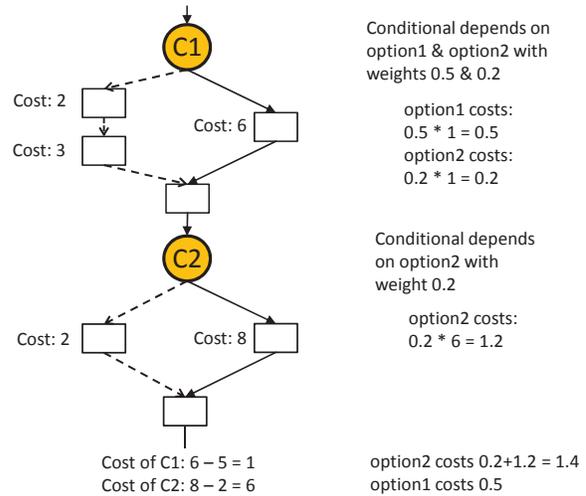

Figure 2. Differential performance summarization

ries of branch conditionals (that happen to be the same instruction). The cost of a divergence is the difference between the performance cost of all basic blocks on the divergent path taken by the first execution and the cost of all blocks on the path taken by the second execution. As in the previous section, X-ray uses the absolute weight method by default. Finally, X-ray sums the per-cause costs over all divergences and outputs a list of root causes ordered by differential cost. In Figure 2, `option2` is output before `option1` because its total cost is greater.

5.3.3 Multi-input differential summarization

Pairwise differential summarization is a powerful tool, but it is most useful if an X-ray user can identify two similar inputs that have markedly different performance. To aid the user in this task, X-ray has a third performance summarization mode that can graphically or numerically compare a large number of inputs.

Multi-input summarization compares inputs that match the same filter. The processing path of these inputs begins at the same basic block (containing the system call that receives the matching data). The subsequent processing paths of the various inputs split and merge. Typically, the paths terminate at the same basic block (e.g., the one closing a connection). If they do not, X-ray inserts an artificial termination block that follows the last block of each input. This allows the collection of input paths to be viewed as a lattice, as shown in Figure 3 for an example with three unique paths.

X-ray discovers this lattice by first representing each input path as a string (as it does for pairwise differential analysis). It then executes a greedy longest common sub-sequence matching algorithm [31] in which it first merges the two strings with the smallest edit distance to form a common path representation, then merges the closest remaining input string with the common representation, etc. The common representation is a graph
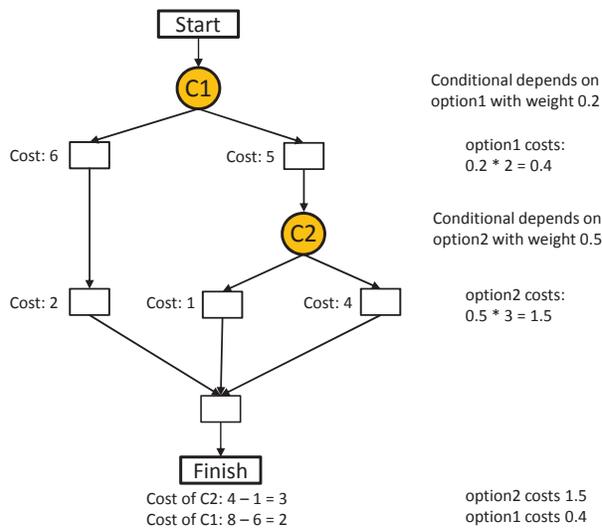
Figure 3. Multi-input differential summarization

where each vertex is a branch at which input paths diverged or a merge point at which input paths converged. Each edge is a sub-path consisting of the basic blocks executed for all inputs that took a common sub-path between two vertexes.

Next, X-ray determines the cost of all divergences. Intuitively, the cost (or benefit) of taking a particular branch should be the difference between the lowest-cost execution that can be achieved by the path taken subtracted from the lowest-cost execution that can be achieved by the path not taken. The weight of a graph edge is the sum of the costs for each block along the edge averaged over all requests that executed the blocks in that edge (e.g., this might be calculated by summing the average latency for each block). X-ray calculates the shortest path on the reverse graph from the termination node to every divergence node for each possible branch. At each branch where paths diverged, X-ray calculates the cost of taking a particular branch to be the difference between the shortest path if that branch is taken and the shortest path from any branch available at that node. For instance, in Figure 3, the cost of conditional branch C2 is 3 (subtracting the cost of the right branch from the left). For C1, the cost is 2 because the shortest path taking the left branch is 8 and the shortest path taking the right branch is 6. The per-divergence cost is then merged with the per-root-cause taints of the branch conditional.

X-ray offers two modes for displaying this data. The first is a numerical summarization that integrates the per-cause costs over all divergences in the graph and displays all root causes in order of total cost. The second method shows the lattice graph, with each divergence node displaying the cost and reasons for the divergence, and the width of each edge representing the number of inputs that traversed the particular sub-path. An X-ray user can use

this graph to identify the most surprising or costly divergences, then select two representative inputs that took opposite branches at that point for a more detailed pairwise comparison. The simpler ordered list is appropriate for casual users. The richer graphical output may be best for power users (or possibly developers).

Multi-path analysis sometimes produced erroneous results due to infeasible shortest paths. These paths arise because X-ray uses taint tracking rather than symbolic analysis. Consider two divergences points that test the same condition. If the `true` outcome has the shortest path after the first test and the `false` outcome has the shortest path after the second test, the shortest path is infeasible because the same condition cannot evaluate to two different values. X-ray uses a statistical analysis to infer infeasible paths. Given a sufficient set of input path samples that pass through two divergence vertexes, if the partition generated by the branch each path took at the first vertex is isomorphic to the partition generated by the branch each took at the second vertex, X-ray infers that the two divergences depend on the same condition and does not allow a shortest path that takes a conflicting path through the two vertexes.

5.4 Fast forward

For long-running applications, replaying days of execution to reach a period of problematic performance may be infeasible. Periodic checkpointing of application state is insufficient because most applications read configuration files at the start of their execution. Thus, the execution after a checkpoint is missing data needed to trace problems back to configuration root causes.

X-ray uses a *fast forward* heuristic to solve this problem. After configuration data is read, X-ray associates dirty bits with each taint to monitor the amount of taint changing during request handling. When less than n% of taint values have been changed by the first n requests after reading a taint source, X-ray considers configuration processing to have quiesced. It saves the current taint values and fast forwards execution to a point that is at least n requests prior to the period of execution being investigated (or to the next opening of a configuration file). It restores saved taints into the address space(s) of the application and resumes instrumented execution.

Use of the fast forward heuristic is optional because it may lead to incorrect results when configuration values are modified as a result of processing requests unmonitored by X-ray. However, we have not seen this behavior in any application to date.

6 Evaluation

Our evaluation answers the following questions:

- How accurately does X-ray identify root causes?
- How fast can X-ray troubleshoot problems?
- How much overhead does X-ray add?

Application	Test	Description of performance test cases
Apache	1	The number of requests that can be handled in one TCP connection is set too low. Reestablishing connections delays some requests [5].
	2	Directory access permissions are based on the domain name of the client sending the request, leading to extra DNS lookups [4].
	3	Logging the host names of clients sending requests to specific directories causes extra DNS requests for files in those directories [4].
	4	Authentication for some directories causes high CPU usage peaks when files in those directories are accessed [3].
	5	Apache can be configured to generate content-MD5 headers calculated using the message body. This header provides an end-to-end message integrity with high confidence. However, for larger files, the calculation of the digests causes high CPU usage [27].
	6	By default, Apache sends eTags in the header of HTTP responses that can be used by clients to avoid resending data in the future if file contents have not changed. However, many mobile phone libraries do not correctly support this option [43].
Postfix	1	Logging more information for a list of specific hosts causes excessive disk activity when one host is the computer running Postfix [38].
	2	Postfix can be configured to examine the body of the messages against a list of regular expressions known to be from spammers or viruses. This setting can significantly increase the CPU usage for handling a received message if there are many expression patterns [40].
	3	Postfix can be configured to reject requests from blacklisted domains. Based on the operators specified, Postfix performs extra DNS calls, which significantly increases message handling latency [39].
PostgreSQL	1	PostgreSQL tries to identify the correct time zone of the system for displaying and interpreting time stamps if the time zone is not specified in the configuration file. This increases the startup time of PostgreSQL by a factor of five.
	2	PostgreSQL can be configured to synchronously commit the write-ahead logs to disk before sending the end of the transaction message to the client. This setting can cause extra delays in processing transactions if the system is under a large load [48].
	3	The frequency of taking checkpoints from the write-ahead log can be configured in the PostgreSQL configuration file. More frequent checkpoints decrease crash recovery time but significantly increase disk activity for busy databases [47].
	4	Setting the delay between activity rounds for the write-ahead log writer process causes excessive CPU usage [47].
	5	A background process aggressively collects database statistics, causing inordinate CPU usage [47].
lighttpd	1	Equivalent to Apache bug 1.
	2	Equivalent to Apache bug 4.
	3	Equivalent to Apache bug 6.

Table 1. Description of the performance test cases used for evaluation

6.1 Experimental Setup

We used X-ray to diagnose performance problems in four servers with diverse concurrency models: the Apache Web server version 2.2.14, the Postfix mail server version 2.7, the PostgreSQL database version 9.0.4, and the lighttpd Web server version 1.4.30. Apache is multithreaded; new connections are received by a listener thread and processed by worker threads. In Postfix, multiple utility processes handle each part of a request; on average, a request is handled by 5 processes. In PostgreSQL, each request is handled by one main process, but work is offloaded in batch to utility processes such as a write-ahead log writer. The lighttpd Web server is event-driven; one thread multiplexes handling of multiple concurrent requests using asynchronous I/O. We ran all experiments on a Dell OptiPlex 980 with a 3.47 GHz Intel Core i5 Dual Core processor and 4 GB of memory, running a Linux 2.6.26 kernel modified to support deterministic replay.

6.2 Root cause identification

We evaluated X-ray by reproducing 16 performance issues (described in Table 1) culled from the cited performance tuning and troubleshooting books, Web documentation, forums, and blog posts. To recreate each issue, we either modified configuration settings or sent a problematic sequence of requests to the server while we recorded server execution. We also used X-ray to troubleshoot an unreported performance issue (described below) that was hampering our evaluation.

For each test, Table 2 shows the scope and metric used for X-ray analysis. The metric was either suggested by

the problem description or a bottleneck resource identified by tools such as `top` and `iostat`. The next column shows where true root cause(s) of the problem were ranked by X-ray. X-ray considered on average 120 possible root causes for the Apache tests, 54 for Postfix, 54 for PostgreSQL, and 48 for lighttpd (these are the average number of tokens parsed from input and configuration files). The last column shows how long X-ray offline analysis took. The reported results do not use the fast-forward heuristic—however, X-ray achieves the same results when fast-forward is enabled.

Our overall results were very positive. X-ray ranked the true root cause(s) first in 16 out of 17 tests. In several cases, multiple root causes contribute to the problem, and X-ray ranked all of them before other causes. In two of the above cases, the true root cause is tied with one or two false positives. In the remaining test, X-ray ranked one false positive higher than the true root causes. Further, the analysis time is quite reasonable when compared to the time and effort of manual analysis: X-ray took 2 minutes and 4 seconds on average to identify the root cause, and no test required more than 9 minutes of analysis time. We next describe a few tests in more detail.

6.2.1 Apache

Apache test 1 shows the power of differential analysis. The threshold for the number of requests that can reuse the same TCP connection is set too low, and reestablishing connections causes a few requests to exhibit higher latency. To investigate, we sent 100 various requests to the Apache server using the *ab* benchmarking tool. The requests used different HTTP methods (GET

Application	Test	Analysis scope	Analysis metric	Correct root cause(s) (rank)	Analysis time
Apache	1	Differential	Latency	**MaxKeepAliveRequests** (1st)	0m 44s
	2	Differential	Latency	**Allow** (t-1st), **domain** (t-1st)	0m 40s
	3	Differential	Latency	**On** (1st), **HostNameLookups** (2nd)	0m 43s
	4	Request	CPU	**AuthUserFile** (1st)	0m 43s
	5	Differential	CPU	**On** (1st), **ContentDigest** (2nd)	0m 44s
	6	Differential	Network	**Input(eTag)** (t-1st)	0m 42s
Postfix	1	Request	File system	**debug_peer_list** (t-1st), **domain** (t-1st)	8m 10s
	2	Request	CPU	**body_checks** (1st)	2m 38s
	3	Request	Latency	**reject_rbl_client** (1st)	2m 18s
PostgreSQL	1	Time interval	CPU	**timezone** (1st)	6m 59s
	2	Request	Latency	**wal_sync_method** (2nd), **synchronous_commit** (3rd)	2m 04s
	3	Time interval	File system	**checkpoint_timeout** (1st)	3m 06s
	4	Time interval	CPU	**wal_writer_delay** (1st)	2m 33s
	5	Time interval	CPU	**track_counts** (1st)	1m 51s
lighttpd	1	Differential	Latency	**auth.backend.htpasswd.userfile** (1st),	0m 34s
	2	Request	CPU	**Input(eTag)** (t-1st),	0m 24s
	3	Differential	Network	**server.max-keep-alive-requests** (1st),	0m 24s

This table shows the type of X-ray analysis performed, the ranking of all true root causes in the ordered list returned by X-ray and X-ray's execution time. The notation, t-1st, shows that the cause was tied for first.

Table 2. X-ray results

and POST) and asked for files of different sizes.

We used X-ray to perform a differential summarization of two similar requests (HTTP GETs of different small files), one of which had a small latency and one of which had a high latency. X-ray identified the `MaxKeepAliveRequests` configuration token as the highest-ranked contributor out of 120 tokens. Based on this information, an end user would increase the threshold specified by that parameter; we verified that this indeed eliminates the observed latency spikes. In the next section, we vary the requests compared for this test to examine how the accuracy of differential analysis depends on the similarity of inputs.

In Apache test 6, the root cause of high network usage is the client's failure to use the HTTP conditional eTag header. A recent study [43] found that many smartphone HTTP libraries do not support this option, causing redundant network traffic. X-ray identifies this problem via differential analysis, showing that it can sometimes identify bad client behavior via analysis of servers. We verified that correct eTag support substantially reduces network load.

6.2.2 Postfix

The first Postfix test reproduces a problem reported in a user's blog [38]—emails with attachments sent from his account transferred very slowly, while everything else, including mail received by IMAP services, had no performance issues. Using `iotop`, the user observed that one child process was generating a lot of disk activity. He poured through the server logs and saw that the child process was logging large amounts of data. Finally, he scanned his configuration file and eventually realized that the `debug_peer_list`, which specifies a list of hosts that trigger logging, included his own IP address. Like many configuration problems, the issue is obvious once explained, yet even an experienced user still spent hours identifying the problem. Further, this level of analysis is

beyond inexperienced users.

In contrast, X-ray quickly and accurately pinpoints the root cause. We simply analyzed requests during a period of high disk usage. X-ray identified the `debug_peer_list` parameter and a token corresponding to our network domain as the top root causes. Since changing either parameter fixes the problem, the user described above could have saved much time with this important clue. Also, no manual analysis such as reading log files was required, so even an inexperienced user could benefit from these results.

6.2.3 PostgreSQL

The first PostgreSQL test is from our own experience. Our evaluation started and stopped PostgreSQL many times. We noticed that our scripts ran slowly due to application start-up delay, so we used X-ray to improve performance. Since `top` showed 100% CPU usage, we performed an X-ray CPU analysis for the interval prior to PostgreSQL receiving the first request.

Unexpectedly, X-ray identified the `timezone` configuration token as the top root cause. In the configuration file, we had set the `timezone` option to `unknown`, causing PostgreSQL to expend a surprising amount of effort to identify the correct time zone. Based on this clue, we specified our correct time zone; we were pleased to see PostgreSQL startup time decrease by over 80%. Admittedly, this problem was esoteric (most users do not start and stop PostgreSQL repeatedly), but we were happy that X-ray helped solve an unexpected performance issue.

In PostgreSQL test 2, X-ray ranked the `shared_buffers` configuration token higher than both true root causes. Manual analysis showed that this token controls the number of database buffers and hence is tested repeatedly by the loop that initializes those buffers. This adds a control flow taint to all database buffers that does not age rapidly due to the large number of such buffers. Such taint could be eliminated by

specifically identifying initialization logic, but we have yet to determine a sound method for doing so.

6.2.4 lighttpd

We chose to evaluate lighttpd to stress X-ray's flow analysis with an event based server in which one thread handles many concurrent requests. Three of the bugs that we examined for Apache have no clear analog in lighttpd. For the remaining three bugs, we introduced similar problems in lighttpd by modifying its configuration file and/or sending erroneous input. X-ray ranked the true root cause first in two tests; in the remaining test, the true root cause was tied for first with two other parameters. From this, we conclude that the flow-based request identification works well with both multithreaded (Apache) and event-based (lighttpd) programs.

6.3 Differential analysis

Experimental methods for analyzing differential performance [14, 26, 46] often require that inputs be identical except for the variable being examined. Unlike these prior methods, X-ray's differential analysis analyzes application control flow and determines the root cause for each divergence between processing paths. Our hypothesis is that this will enable X-ray to generate meaningful results even for inputs that have substantial differences.

To validate this hypothesis, we repeated the first Apache test. Instead of selecting similar requests, we selected the pair of requests that were most different: a small HTTP POST that failed and a very large HTTP GET that succeeded. Somewhat surprisingly, X-ray still reported `MaxKeepAliveRequests` as the top root cause. The reason was a bit fortuitous: in our benchmark, the `MaxKeepAliveRequests` option happened to increase the latency of the small request, so the latency due to the misconfiguration exhibited as a performance degradation, while the difference in request input exhibited as a performance improvement.

We verified this by reversing the order of the two requests so that the large request was slowed by connection re-establishment rather than the small request. In this case, X-ray reported differences in the input request data as the largest reason why the processing of the large request is slower. It incorrectly ranked the `DocumentRoot` parameter second because the root is appended to the input file name before data is read from disk. `MaxKeepAliveRequests` ranked third.

We conclude that differential analysis does not always require that two requests be substantially similar in order to identify root causes of performance anomalies. Differences in input will of course rise to the top of the ranked list. However, after filtering these causes out, the true root cause was still ranked very near the top in our experiment, so a user would not have had to scan very far.

Finally, we applied multi-request differential analysis to this test by sending 100 requests of varying

| Strategy | False positives | | | | True cause |
	0	1	2	3+	unranked
Absolute	21	2	0	0	0
Normalized	20	0	3	0	0
Winner-take-all	15	3	1	2	2
Learning	20	2	1	0	0

For each strategy, this shows the number of false positives ranked above each of the 23 true root causes from Table 2. The winner-take-all strategy failed to identify 2 true root causes.

Table 3. Accuracy of attribution strategies

types (GET and POST), sizes, and success/failure outcomes. When we compared all 100 requests and filtered out input-related causes, the true root cause was ranked second (behind the `ServerRoot` token). For an end user, this mode is especially convenient because the user need not identify specific requests to compare. For power users and developers, the graphical version of the multi-path output shows the specific requests for which `MaxKeepAliveRequests` causes path divergences.

6.4 Sensitivity analysis

Section 5.3.1 described four strategies for attributing performance to root causes. Table 3 summarizes the results of running all tests in Section 6.2 with each strategy. There are 23 true root causes in the 17 tests. The second column shows the number of these for which no false positive is higher in the final X-ray rankings. The next column shows the number for which 1 false positive is ranked higher, etc. The final column shows true causes that did not appear at all in X-ray's rankings.

The winner-take-all strategy is substantially worse than the other strategies because the true root cause ranks second or third for many basic blocks, and so its impact is underestimated. All other strategies are roughly equivalent, with the absolute strategy being slightly more accurate than the other two. We conclude that the X-ray algorithm is not very sensitive to the particular attribution algorithm as long as that algorithm considers more than just the top cause for each basic block.

As described in Section 4.1, X-ray uses ConfAid's taint aging heuristics: control flow taint is multiplied by a weight of 0.5 when it is merged with data flow taint or when a new tainted conditional branch is executed. We performed a sensitivity analysis, shown in Table 4, that examined the effect of changing this weight. Note that a weight of 0 is equivalent to the winner-take-all strategy, and a weight of 1 does not age taint at all. While the default weight of 0.5 produced slightly better results than other weights, all values within the range 0.125–0.875 had roughly equivalent results in our experiments.

6.5 X-ray online overhead

We measured online overhead by comparing the throughput and latency of applications when they are

| Weight | False positives | | | | True cause |
	0	1	2	3+	unranked
0	15	3	1	2	2
0.125	19	2	2	0	0
0.25	20	3	0	0	0
0.5	21	2	0	0	0
0.75	20	0	1	2	0
0.875	20	0	1	2	0
1	8	3	2	10	0

For each weight, this shows the number of false positives ranked above each of the 23 true root causes from Table 2.

Table 4. Accuracy when using different weights

recorded by X-ray to results when the applications run on default Linux without recording. For Apache and lighttpd, we used ab to send 5000 requests for a 35 KB static Web page with a concurrency of 50 requests at a time over an isolated network. For Postfix, we used smtp-source to send 1000 64 KB mail messages. For PostgreSQL, we used pgbench to measure the number of transactions completed in 60 seconds with a concurrency of 10 transactions at a time. Each transaction has one SELECT, three UPDATEs, and one INSERT command.

Figure 4 shows X-ray adds an average of 2.3% throughput overhead: 0.1% for Apache, 4.7% for Postfix, 3.5% for PostgreSQL, and 0.8% for lighttpd. These values include the cost of logging data races previously detected by our offline data race detector. This overhead is consistent with similar deterministic replay approaches [18]. Latency overheads for Apache, PostgreSQL, and lighttpd are equivalent to the respective throughput overheads; Postfix has no meaningful latency measure since its processing is asynchronous. The recording log sizes were 2.8 MB for Apache, 1.6 MB for lighttpd, 321 MB for PostgreSQL, and 15 MB for Postfix. Apache and lighttpd have smaller logs because they use sendfile to avoid copying data.

6.6 Discussion

X-ray's accuracy has exceeded our expectations. One reason for this is that many performance issues, like the Postfix example in Section 6.2.2, are obvious once explained. Without explanation, however, searching for the root cause is a frustrating, "needle-in-a-haystack" process. Performance summarization is essentially a brute-force method for searching through that haystack. The obvious-once-explained nature of many performance problems has another nice property: X-ray's accuracy is not very sensitive to the exact heuristics it employs, so many reasonable choices lead to good results.

X-ray's most significant limitation is that it does not track taint inside the OS so it cannot observe causal dependencies among system calls. For instance, X-ray cannot determine when one thread blocks on a kernel queue waiting for other threads or kernel resources. In addition,
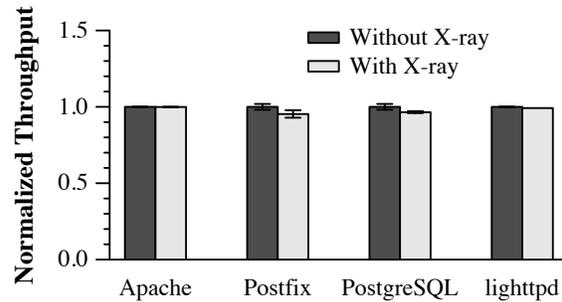
Each dataset shows server throughput with and without X-ray recording, normalized to the number of requests per second without X-ray. Higher values are better. Each result is the mean of at least 10 trials; error bars are 95% confidence intervals.

Figure 4. X-ray online overhead

system call parameter values often affect the amount of work performed by the kernel. X-ray currently addresses this on an ad-hoc basis; e.g., it attributes amount of the work performed by read and write system calls to the size input parameter for each call. However, X-ray currently only supports a small number of system call parameters in this fashion. We hope to address these limitations more thoroughly, either by instrumenting the kernel or by creating more detailed performance models that observe system calls, parameters, and selected kernel events to infer such dependencies.

X-ray only considers configuration settings and program inputs as possible root causes. If the true root cause is a program bug or any other cause not considered by X-ray, X-ray cannot diagnose the problem. X-ray will produce an ordered list of possible causes, all of which will be incorrect. Thus, one potential improvement is to require a minimal level of confidence before X-ray adds a root cause to the ordered list—this may enable X-ray to better identify situations where the true root cause is not in its domain of observation.

While X-ray adds only a small overhead on the production system, its offline analysis runs 2–3 orders of magnitude slower than the original execution. Thus, while logging may be continuously enabled, we envision that only portions of the log will be analyzed offline.

7 Conclusion

Diagnosing performance problems is challenging. X-ray helps users and administrators by identifying the root cause of observed performance problems. Our results show that X-ray accurately identifies the root cause of many real-world performance problems, while imposing only 2.3% average overhead on a production system.

References

[1] AGUILERA, M. K., MOGUL, J. C., WIENER, J. L., REYNOLDS, P., AND MUTHITACHAROEN, A. Performance debugging for distributed systems of black boxes. In *Proc. SOSP* (October 2003), pp. 74–89.

[2] ALTEKAR, G., AND STOICA, I. ODR: Output-deterministic replay for multicore debugging. In *Proc. SOSP* (October 2009), pp. 193–206.

[3] Apache HTTP server version 2.4 documentation: Authentication, authorization, and access control. `http://httpd.apache.org/docs/2.2/howto/autho.html`.

[4] Apache HTTP server version 2.4 documentation: Apache performance tuning. `http://httpd.apache.org/docs/current/misc/perf-tuning.html`.

[5] Apache performance tuning. `http://perlcode.org/tutorials/apache/tuning.html`.

[6] ATTARIYAN, M., AND FLINN, J. Automating configuration troubleshooting with dynamic information flow analysis. In *Proc. OSDI* (October 2010).

[7] BARHAM, P., DONNELLY, A., ISAACS, R., AND MORTIER, R. Using Magpie for request extraction and workload modelling. In *Proc. OSDI* (December 2004), pp. 259–272.

[8] BHATIA, S., KUMAR, A., FIUCZYNSKI, M. E., AND PETERSON, L. Lightweight, high-resolution monitoring for troubleshooting production systems. In *Proc. OSDI* (December 2008), pp. 103–116.

[9] BRESSOUD, T. C., AND SCHNEIDER, F. B. Hypervisor-based fault tolerance. *ACM TOCS 14*, 1 (February 1996), 80–107.

[10] BROWN, A. B., AND PATTERSON, D. A. To err is human. In *DSN Workshop on Evaluating and Architecting System Dependability* (July 2001).

[11] BROWN, A. B., AND PATTERSON, D. A. Undo for operators: Building an undoable e-mail store. In *Proc. USENIX ATC* (June 2003).

[12] CADAR, C., DUNBAR, D., AND ENGLER, D. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. OSDI* (December 2008), pp. 209–224.

[13] CANTRILL, B. M., SHAPIRO, M. W., AND LEVENTHAL, A. H. Dynamic instrumentation of production systems. In *Proc. USENIX ATC* (June 2004), pp. 15–28.

[14] CHEN, H., JIANG, G., ZHANG, H., AND YOSHIHIRA, K. Boosting the performance of computing systems through adaptive configuration tuning. In *Proc. SAC* (March 2009), pp. 1045–1049.

[15] CHEN, M. Y., ACCARDI, A., KICIMAN, E., LLOYD, J., PATTERSON, D., FOX, A., AND BREWER, E. Path-based failure and evolution management. In *Proc. NSDI* (March 2004).

[16] CHEN, M. Y., KICIMAN, E., FRATKIN, E., FOX, A., AND BREWER, E. Pinpoint: Problem determination in large, dynamic Internet services. In *Proc. DSN* (June 2002), pp. 595–604.

[17] CHIPOUNOV, V., KUZNETSOV, V., AND CANDEA, G. S2E: A platform for in vivo multi-path analysis of software systems. In *Proc. ASPLOS* (March 2011).

[18] CHOW, J., GARFINKEL, T., AND CHEN, P. M. Decoupling dynamic program analysis from execution in virtual environments. In *Proc. USENIX ATC* (June 2008), pp. 1–14.

[19] COHEN, I., GOLDSZMIDT, M., KELLY, T., SYMONS, J., AND CHASE, J. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *Proc. OSDI* (December 2004), pp. 231–244.

[20] DIAO, Y., HELLERSTEIN, J. L., PAREKH, S., AND BIGUS, J. P. Managing Web Server Performance with AutoTune Agent. *IBM Systems Journal 42*, 1 (January 2003), 136–149.

[21] DUNLAP, G. W., KING, S. T., CINAR, S., BASRAI, M. A., AND CHEN, P. M. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proc. OSDI* (December 2002), pp. 211–224.

[22] FONSECA, R., PORTER, G., KATZ, R. H., SHENKER, S., AND STOICA, I. X-trace: A pervasive network tracing framework. In *Proc. NSDI* (April 2007), pp. 271–284.

[23] GEELS, D., ALTEKAR, G., SHENKER, S., AND STOICA, I. Replay debugging for distributed applications. In *Proc. USENIX ATC* (June 2006).

[24] GRAY, J. Why do computers stop and what can be done about it? In *Proc. Symp. Rel. Dist. Sofrware and DB Syst.* (1986).

[25] JUNQUEIRA, F., SONG, Y. J., AND REED, B. BFT for the skeptics. In *Proc. SOSP: WIP Session* (October 2009).

[26] KASICK, M. P., TAN, J., GANDHI, R., AND NARASIMHAN, P. Black-box problem diagnosis in parallel file systems. In *Proc. FAST* (February 2010).

[27] LAURIE, B., AND LAURIE, P. *Apache: The Definitive Guide, 3rd Edition*. O'Reilly Media, Inc., December 2002.

[28] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proc. PLDI* (June 2005), pp. 190–200.

[29] `http://msdn.microsoft.com/en-us/library/bb968803(v=VS.85).aspx`.

[30] MURPHY, B., AND GENT, T. Measuring system and software reliability using an automated data collection process. *Quality and Reliability Engineering International 11*, 5 (1995).

[31] MYERS, E. W. An O(ND) difference algorithm and its variations. *Alogorithmica 1*, 1–4 (1986), 251–266.

[32] NAGARAJA, K., OLIVERIA, F., BIANCHINI, R., MARTIN, R. P., AND NGUYEN, T. Understanding and dealing with operator mistakes in Internet services. In *Proc. OSDI* (December 2004), pp. 61–76.

[33] NEWSOME, J., AND SONG, D. Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software. In *Proc. NDSS* (February 2005).

[34] OPPENHEIMER, D., GANAPATHI, A., AND PATTERSON, D. A. Why do Internet services fail, and what can be done about it? In *Proc. USITS* (March 2003).

[35] `http://oprofile.sourceforge.net/`.

[36] PARK, S., ZHOU, Y., XIONG, W., YIN, Z., KAUSHIK, R., LEE, K. H., AND LU, S. PRES: Probabilistic replay with execution sketching on multiprocessors. In *Proc. SOSP* (October 2009), pp. 177–191.

[37] PATIL, H., PEREIRA, C., STALLCUP, M., LUECK, G., AND COWNIE, J. PinPlay: A framework for deterministric replay and reproducible analysis of parallel programs. In *Proc. CGO* (March 2010).

[38] `http://www.karoltomala.com/blog/?p=576`.

[39] Postfix stress-dependent configuration. `http://www.postfix.org/STRESS_README.html`.

[40] Postfix tuning guide. `http://www.postfix.org/TUNING_README.html`.

[41] POZNIANSKY, E., AND SCHUSTER, A. Efficient on-the-fly data race detection in multithreaded C++ programs. In *Proc. PPoPP* (June 2003), pp. 179–190.

[42] PRASAD, V., COHEN, W., EIGLER, F. C., HUNT, M., KENISTON, J., AND CHEN, B. Locating system problems using dynamic instrumentation. In *Proceedings of the Linux Symposium* (July 2005), pp. 49–64.

[43] QIAN, F., QUAH, K. S., HUANG, J., ERMAN, J., GERBER, A., MAO, Z. M., SEN, S., AND SPATSCHECK, O. Web caching on smartphones: Ideal vs. reality. In *Proc. MobiSys* (June 2012).

[44] RONSSE, M., AND DE BOSSCHERE, K. RecPlay: A fully integrated practical record/replay system. *ACM TOCS 17*, 2 (May 1999), 133–152.

[45] RUAN, Y., AND PAI, V. Making the "box" transparent: System call performance as a first-class result. In *Proc. USENIX ATC* (June 2004), pp. 1–14.

[46] SAMBASIVAN, R. R., ZHENG, A. X., ROSA, M. D., KREVAT, E., WHITMAN, S., STROUCKEN, M., WANG, W., XU, L., AND GANGER, G. R. Diagnosing performance changes by comparing request flows. In *Proc. NSDI* (March 2011), pp. 43–56.

[47] SMITH, G. *PostgreSQL 9.0 High Performance*. October 2010.

[48] SMITH, G., TREAT, R., AND BROWNE, C. Tuning your postgresql server. `http://wiki.postgresql.org/wiki/Tuning_Your_PostgreSQL_Server`.

[49] SRINIVASAN, S., ANDREWS, C., KANDULA, S., AND ZHOU, Y. Flashback: A light-weight extension for rollback and deterministic replay for software debugging. In *Proc. USENIX ATC* (June 2004), pp. 29–44.

[50] SU, Y.-Y., ATTARIYAN, M., AND FLINN, J. AutoBash: Improving configuration management with operating system causality analysis. In *Proc. SOSP* (October 2007), pp. 237–250.

[51] VEERARAGHAVAN, K., CHEN, P. M., FLINN, J., AND NARAYANASAMY, S. Detecting and surviving data races using complementary schedules. In *Proc. SOSP* (October 2011).

[52] VEERARAGHAVAN, K., LEE, D., WESTER, B., OUYANG, J., CHEN, P. M., FLINN, J., AND NARAYANASAMY, S. DoublePlay: Parallelizing sequential logging and replay. In *Proc. ASPLOS* (March 2011).

[53] `http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/`.

[54] WANG, H. J., PLATT, J. C., CHEN, Y., ZHANG, R., AND WANG, Y.-M. Automatic misconfiguration troubleshooting with PeerPressure. In *Proc. OSDI* (December 2004), pp. 245–257.

[55] WANG, Y.-M., VERBOWSKI, C., DUNAGAN, J., CHEN, Y., WANG, H. J., YUAN, C., AND ZHANG, Z. STRIDER: A black-box, state-based approach to change and configuration management and support. In *Proc. LISA* (October 2003), pp. 159–172.

[56] WHITAKER, A., COX, R. S., AND GRIBBLE, S. D. Configuration debugging as search: Finding the needle in the haystack. In *Proc. OSDI* (December 2004), pp. 77–90.

[57] XU, M., MALYUGIN, V., SHELDON, J., VENKITACHALAM, G., AND WEISSMAN, B. ReTrace: Collecting execution trace with virtual machine deterministic replay. In *Proc. MoBS* (June 2007).

[58] YIN, Z., MA, X., ZHENG, J., ZHOU, Y., BAIRAVASUNDARAM, L., AND PASUPATHY, S. An emprirical study on configuration errors in commerical and open source systems. In *Proc. SOSP* (October 2011).

[59] YU, M., GREENBERG, A., MALTZ, D., REXFORD, J., YUAN, L., KANDULA, S., AND KIM, C. Profiling network performance for multi-tier data center applications. In *Proc. NSDI* (March 2011), pp. 57–70.

[60] ZHENG, W., BIANCHINI, R., JANAKIRAMAN, G. J., SANTOS, J. R., AND TURNER, Y. JustRunIt: Experiment-based management of virtualized data centers. In *Proc. USENIX ATC* (June 2009).

[61] ZHENG, W., BIANCHINI, R., AND NGUYEN, T. D. Automatic configuration of Internet services. In *Proc. EuroSys* (March 2007), pp. 219–229.