Automatic Root-cause Diagnosis of Performance Anomalies in Production Software

Mona Attariyan, Michael Chow and Jason Flinn
University of Michigan
{monattar, mcchow, jflinn}@umich.edu

ABSTRACT

Troubleshooting the performance of complex production software is challenging. Most existing tools, such as profiling, tracing, and logging systems, reveal *what* events occurred during performance anomalies. However, the users of such tools must then infer *why* these events occurred during a particular execution; e.g., that their execution was due to a specific input request or configuration setting. Because manual root cause determination is time-consuming and difficult, this paper introduces *performance summarization*, a technique for automatically inferring the root cause of performance problems. Performance summarization first attributes performance costs to fine-grained events such as individual instructions and system calls. It then uses dynamic information flow to determine the probable root causes for the execution of each event. The cost of each event is assigned to root causes according to the relative probability that the causes led to the execution of that event. Finally, the total cost for each root cause is calculated by summing the per-cause costs of all events. This paper also describes a differential form of performance summarization that compares two activities. We have implemented a tool called X-ray that performs performance summarization. Our experimental results show that X-ray accurately diagnoses 14 performance issues in the Apache HTTP server, Postfix mail server and PostgreSQL database, while adding only 1–7% overhead to production systems.

1. INTRODUCTION

Understanding and troubleshooting performance problems in complex software systems is notoriously challenging. This challenge is compounded for software in production for several reasons. To avoid slowing down production systems, analysis and troubleshooting must incur minimal overhead. Further, performance issues in production can be both rare and non-deterministic, making the issues hard to reproduce.

However, we argue that the most important reason why troubleshooting performance in production systems is challenging is that current tools only solve half the problem. Troubleshooting a performance anomaly is essentially the process of determining *why* certain events, such as high latency or resource usage, happened in a system. Unfortunately, most current analysis tools, such as profilers and logging, only determine *what* events happened during a performance anomaly — they leave the more challenging question of why those events happened unanswered. Administrators and developers must manually infer the root cause of performance issue from the 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 of the tool must then infer that the routine is called more often due to a specific configuration setting.

In this paper, we introduce the technique of *performance summarization* which not only determines what events occurred during a performance anomaly but also determines why the anomaly occurred. Performance summarization first attributes performance costs such as latency and I/O utilization to fine-grained events (individual instructions and system calls). Then, it uses dynamic information flow analysis to associate each such event with a set of probable root causes such as configuration settings or specific data from input requests. The cost of each event is assigned to potential root causes weighted by the probability that the particular root cause led to the execution of that event. Finally, the per-cause costs for all events in the program execution are summed together. The end result is a list of root causes ordered by their performance costs. In the above example, the outcome of performance summarization would indicate that one specific configuration setting contributed the most to the performance slowdown. This output gives the system troubleshooter a direct indication of how to fix the problem, without the need for time-consuming manual analysis.

We also introduce *differential performance analysis* which is used to determine why the performance impact of two different events differed. For instance, differential performance analysis can be used to understand why two requests to a Web server took different amounts of time to complete. Differential performance analysis identifies branches where the execution paths of the two requests diverged. It assigns a performance cost to each path taken from the branch, then uses dynamic information flow analysis to determine why the two requests diverged at that point. It attributes the difference in performance costs between the two paths to the identified root causes according to the likelihood that they caused the branch condition to evaluate to different values during the two executions. The costs of all such divergences during are summed. The output shows the system troubleshooter a set of reasons why the performance costs of two requests differ, along with a specific performance impact for each reason. For example, the output for a Web server might show that 70% of the difference between two requests is due to the specific file requested, while 30% is due to a configuration parameter that caused extra initialization logic to be run in one request but not the other.

We have built a tool called X-ray that implements performance summarization. X-ray attributes latency, CPU utilization, disk usage, and network utilization to specific root causes. X-ray supports several different scopes of analysis: intervals of time, specific requests, or a differential analysis of pairs of requests. Thus, X-ray can answer performance questions such as:

- Why did a particular request take a long time to execute?
- Why is disk utilization high during a specific time period?
- Why did request R take longer to execute than request S?

Performance summarization is a high-overhead activity. In order to execute this analysis for production software, X-ray leverages prior work in deterministic replay to offload the heavyweight analysis from the production system. A deterministic replay system provides DVR-like functionality, in which an execution of a hardware or software system is recorded so that an identical execution can later be replayed on demand. Our use of deterministic replay to troubleshoot performance issues raised several new challenges. X-ray must split its functionality among the recorded and replayed executions; for example, timestamps must be captured during recording because the heavyweight analysis substantially perturbs timing. Further, because of the split analysis, the fidelity of the replay must be strict enough to guarantee that the two executions are identical at the granularity observed by X-ray. However, because the replayed execution includes analysis code that the recorded execution does not, the fidelity of the replay must be loose enough to allow the replayed execution to diverge enough to run the analysis. We show that all these goals can be achieved through careful co-design of the deterministic replay and analysis systems.

Thus, this paper contributes the following:

- The technique of performance summarization, which attributes performance costs to root causes.
- The technique of differential performance summarization for understanding why two similar events have different performance.
- Co-design methodologies for modifying deterministic replay systems to support heavyweight performance analysis tools such as binary instrumentation without imposing substantial overhead on a production system.
- Development and evaluation of the X-ray tool, which implements these techniques.

We evaluated X-ray using three applications: the Apache Web server, the Postfix mail server and the PostgreSQL database. We have reproduced and analyzed 14 performance issues reported for these applications. In 12 of 14 cases, X-ray identifies a correct root cause as the largest contributor to the performance problem; in the remaining two cases, X-ray identifies a correct root cause as the third largest contributor. Our evaluation also shows that X-ray adds a performance overhead of only 1–7% on the production system.

2. RELATED WORK

While many prior systems help troubleshoot performance issues, X-ray is the first such system that uses dynamic information flow analysis to automatically identify the root cause of production performance issues without the need for manual inference.

Profilers such as OProfile [23], VTune [35], DTrace [9], SystemTap [27], ETW [21], Debox [28], and Chopstix [6] allow the troubleshooter to instrument applications and/or the operating system and collect performance data. These tools reveal *what* events (e.g., functions) incur substantial performance costs. However, their users must manually infer *why* those events executed. In contrast, X-ray automatically identifies root causes.

Other tracing systems follow activities across multiple components or protocol layers, and use the causal relationships they observe to propagate and merge performance data. X-trace [17] observes network activities across protocols and layers in a distributed system. SNAP [40] profiles TCP-statistics and socket-call logs and correlates data across a data center. Aguilera *et al*. [1] use statistical analysis to infer causal paths between application components and attribute delays to specific nodes. Pinpoint [12] traces communication between middleware components to infer which components are responsible for causing faults. Follow-on work [11] adds the abstraction of causal *paths* that link black-box components. Like these tools, X-ray uses causality to propagate data across components when processes communicate (although propagation is currently limited to a single node by its replay system). Unlike these tools, X-ray analyzes causality *within* application components using dynamic binary instrumentation, so it can determine the specific relationship between component inputs and outputs.

Other performance troubleshooting tools build or use a model of application performance. Magpie [5] accurately extracts the control flow and resource consumption of each request to build a workload model that can be used for performance prediction. Magpie's per-request profiling can help troubleshooters diagnose potential performance problems. Even though Magpie provides detailed performance information that can be used 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 various components; X-ray currently uses a simpler method and thus could benefit from using Magpie's schemas for complicated request patterns.

Stewart *et al.* [31] extract resource usage from multi-component services to generate performance models for capacity planning and cost-effectiveness analysis. Urgaonkar *et al.* [4] use resource usage profiling to guide application placement in shared hosting platforms. Cohen *et al.* [14] use statistical learning techniques to automatically build system models. They identify a combination of system-level metrics and threshold values that correlate with high-level performance states. In contrast to X-ray, none of these systems tie performance to specific root causes such as configuration options.

Many research projects tune performance [15, 10, 41] by injecting artificial traffic and using machine learning to correlate performance with specific configuration options. Unlike X-ray, these tools limit the set of configuration options analyzed, and they must see controlled traffic in order to learn good configuration values.

Spectroscope [29] diagnoses performance changes by comparing request flows between two executions of the same workload. Kasick *et al*. [19] compare similar requests to diagnose performance bugs in parallel file systems. Unlike X-ray, these tools must see very similar requests in order to diagnose performance problems. In contrast, our results show that X-ray can correctly identify root causes even when requests are very dissimilar because it analyzes the control path of each request.

PeerPressure [36] and its predecessor, Strider [37], use statistical methods to compare configuration state in the Windows registry on different machines. These systems analyze static state rather than observe applications as they execute. Chronus [38] compares configuration states of the same computer across time using virtual machine checkpoint and rollback. AutoBash [32] helps users troubleshoot configuration by providing OS-level features such as causality analysis and speculative execution.

X-ray uses taint tracking [22] to identify the root cause for the execution of individual instructions and system calls. For this purpose, we use ConfAid [3]. ConfAid is a tool that we originally designed to debug program failures by helping users and administrators attribute those failures to erroneous configuration options. X-ray re-purposes ConfAid to tackle performance analysis.

Potentially, X-ray could instead have used other methods for inferring causality such as symbolic execution [8]. For instance, S2E [13] presented a case study in which symbolic execution was used to learn the relationship between inputs and low-level events

```
Allow domain.name (line 164) : 603 usecs
ServerRoot (line 29)         : 151 usecs
TypesConfig (line 298)       : 151 usecs
<IfModule(line 231)          : 75 usecs
alias\_module(line 231)      : 75 usecs
<Directory(line 162)         : 55 usecs
...
```

Figure 1: Example of X-ray output for Apache

such as page faults and instruction counts. While X-ray and S2E both use causality to explain low-level events, their focus is very different. X-ray starts from a specific performance anomaly seen in production and traces causality to find that event's root cause, whereas S2E starts from a specific set of inputs and explores multiple paths to infer the performance impact of different input choices.

X-ray uses deterministic record and replay. While many prior software systems provide this functionality [2, 16, 18, 24, 30, 34, 30], X-ray introduces new constraints that prior systems cannot satisfy. The fidelity of replay must be high enough to exactly reproduce application instructions and system calls, while still being loose enough to execution instrumentation during the replayed execution but not during the recorded execution. X-ray modified the replay system to compensate for the instrumentation code in order to achieve the needed fidelity.

3. X-RAY OVERVIEW

3.1 Troubleshooting with X-ray

X-ray pinpoints why performance anomalies, such as high request latencies or bottlenecks in resources, occurred on a production system. Our current system targets servers, though this is not fundamental to our design.

X-ray does not require application source code because its analysis operates entirely on application binaries and modifications are made using dynamic binary instrumentation. X-ray recognizes configuration tokens and other root causes through a limited form of binary symbolic execution. Thus, X-ray can be used on COTS (common off-the-shelf) applications, making the tool appropriate for system administrators as well as for developers.

The first step in using X-ray is to record an interval of software execution on a production system. As the results in Section 7 show, X-ray recording overhead is currently only 1–7%. Thus, a user can choose to leave X-ray running for long periods of time to capture rare and hard-to-reproduce performance issues. Alternatively, X-ray can be dynamically enabled only when specific performance issues are exhibited.

X-ray uses deterministic record and replay to offload analysis of the recorded execution from the production system. An X-ray user chooses which interval of execution to analyze. The user may select the entire execution, an interval of time, or a specific input request. X-ray produces a performance summary for the selected interval. The first two intervals are appropriate when the user notices degraded throughput over a period of time, whereas the latter is best when one or more requests take an unexpected amount of time to execute. Alternatively, a user may select two requests to compare, in which case X-ray does a differential performance summarization for the selected requests. Typically, a user would select two similar requests that differ substantially in service time, though our results show that X-ray will provide useful information even when the two selected requests are very dissimilar.

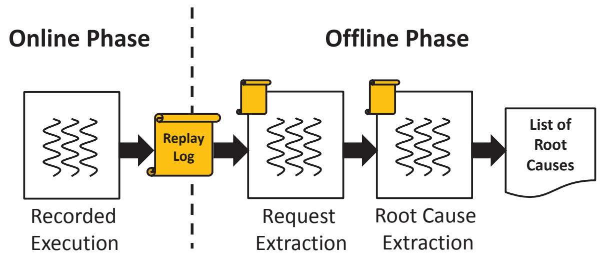


Figure 2: Overview of X-ray

The X-ray user next selects the set of performance statistics to summarize. Typically, we expect that a user will use basic performance analysis tools such as `top` and `iostat` to identify the bottleneck resource. X-ray provides a flexible framework for analyzing arbitrary statistics; our current implementation supports latency, CPU utilization, disk throughput and network bandwidth.

Figure 1 shows an example of X-ray output for Apache. The output shows the inferred root causes of a performance problem. X-ray associates a specific cost (in this case, latency) to each root cause and orders the list by that metric. In the figure, all root causes are from the `httpd.conf` configuration file. Based on X-ray output, users may identify configuration options that are inappropriate for their workload, they might choose a set of configuration options that offer a different tradeoff between performance or functionality, or they may re-provision their system to supply resources in quantities that match the features they desire.

Executions recorded by X-ray can be replayed multiple times. Therefore, X-ray users can perform many different analyses for the same recording. For instance, a user may change the scope of execution analyzed, choose different metrics to summarize, or switch between basic and differential performance summarization. This means that the X-ray user does not need to decide what type of analysis will be useful before a performance anomaly is recorded.

3.2 Mechanics of X-ray

Figure 2 shows an overview of how X-ray executes. To minimize production overhead, X-ray uses deterministic record and replay to divide its execution into two phases. In its online phase, X-ray observes the production system *in situ* and logs all non-deterministic inputs for the application so that it can subsequently replay that application elsewhere. X-ray also records timing information and other performance-specific data during the online phase because the subsequent, offline analysis perturbs the execution too much to accurately measure performance.

In its offline phase, X-ray performs two passes, each of which is a deterministic replay of the recorded execution. In the first pass, X-ray performs *request extraction*, in which it determines the specific intervals of execution (i.e., the basic blocks executed) during which each process is handling each input request to the recorded system. In the first pass, X-ray also assigns the recorded performance costs to each instruction and system call. In the second pass, X-ray completes performance summarization by using dynamic information flow to attribute events to root causes and by calculating the cost of each root cause. At the end of the second pass, X-ray outputs a list of root causes ordered by its user's chosen performance metric.

4. PERFORMANCE SUMMARIZATION

Performance summarization is the heart of X-ray. The goal is to attribute specific performance costs such as request latency, CPU

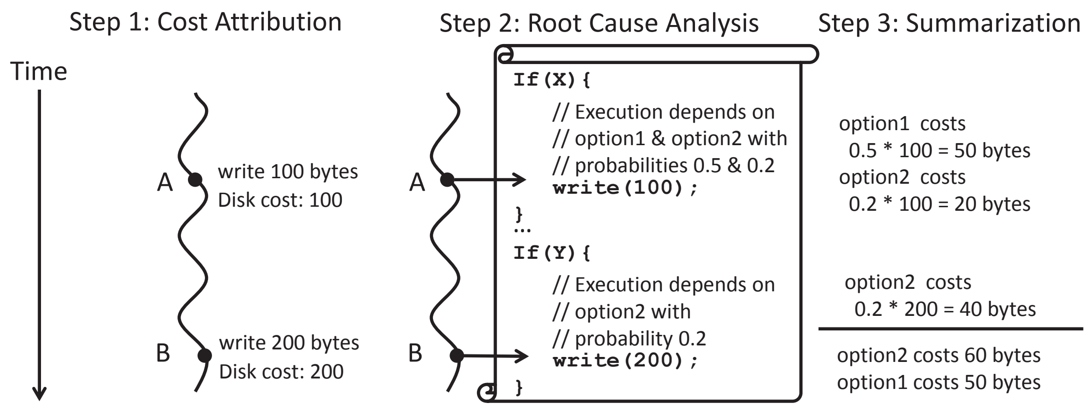


Figure 3: Example of performance summarization

usage, and I/O utilization to one or more root causes. X-ray considers any configuration option or any data received from an input request as a potential root cause.

4.1 Basic performance summarization

Performance summarization is akin to integration in calculus. X-ray individually analyzes the per-cause performance cost and root cause of each user-level instruction and system call (referred to as events in the discussion below), then adds together the per-event costs to calculate how much each root cause has reflected the performance of the application during the period of observation selected by the X-ray user.

Figure 3 shows an overview of how performance summarization works. In the first step, X-ray attributes performance metrics to each event executed by one or more processes comprising a server application; the figure assumes that the X-ray user has specified disk bandwidth as a metric. Some metrics such as disk bandwidth are associated only with system calls, while others such as latency are attributed to both system calls and user-level instructions.

In the next step, X-ray uses taint tracking to derive a set of possible root causes for the execution of each event. Essentially, this step answers the question: "how likely is it that changing a configuration option or receiving a different input would have prevented this event from executing?" Taints are tracked as floating-point values using algorithms that we developed in ConfAid [3], with the numerical taint value associated with each root cause reflecting the belief that the root cause is the reason why the event was executed.

In the last step, X-ray multiplies the performance metrics for each event by the per-cause taint values to derive the per-event performance cost for each root cause. X-ray sums these costs over all events that executed during the period selected by the user and outputs an ordered list of root causes.

4.2 Differential performance summarization

Differential performance summarization is a technique for comparing any two executions of an application activity, such as the processing of two different request by a Web server. Such activities have a common starting point (e.g., the receipt of a request) and termination point (e.g., the sending of a response), but the execution paths for different events may diverge due to differences in the input or specific configuration settings.

Figure 4 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. It then uses taint tracking to evaluate why each divergence occurred; this reason is given by the taint of the branch conditional at the divergence point. For each performance metric, X-ray calculates the cost of the divergence by subtracting the cost of all events on the divergent path taken by the first execution from the cost of all events on the path taken by the second execution. This cost is attributed to root causes by multiplying the metric values by the taint weight. X-ray sums the per-cause costs of all divergences and output a list of root causes ordered by the differential cost.

5. REPLAY FIDELITY

The dynamic instrumentation used by X-ray can slow down the execution of an application by several orders of magnitude; this overhead is too high to run X-ray directly on production software. Therefore, X-ray employs deterministic record and replay to offload time-consuming analysis from the production system to another computer. There exist many systems that provide deterministic replay by recording the initial state of an execution and logging all non-deterministic events that occur during the execution [7, 16, 30, 33]. With such systems, an execution can subsequently be reproduced on demand by restarting execution from the initial state and supplying the previously-recorded values for all non-deterministic events.

While deterministic replay is a well-studied technique, we encountered several new challenges in adapting the technique to work with X-ray. In particular, we found that we needed to carefully balance the *fidelity* of the record and replay and that we needed to *co-design* the deterministic replay system to work with the dynamic instrumentation employed by X-ray.

We define the fidelity of the replay to be the degree to which the replayed execution is guaranteed to match the recorded execution. For the purposes of X-ray, replay fidelity must be high enough to guarantee that the recording and replaying systems execute the same application instructions and system calls in the same order.

Since performance analysis via binary instrumentation can cause runtime overheads of several orders of magnitude, timing information gathered during an instrumented run is essentially useless for diagnosing most performance problems. In contrast, timing information gathered during the recorded run captures the exact performance experienced by the production system. X-ray therefore gathers timing data during recoding and explains the timing data by reasoning about the instructions and system calls executed during replayed executions. Thus, if the two sequences of instructions and

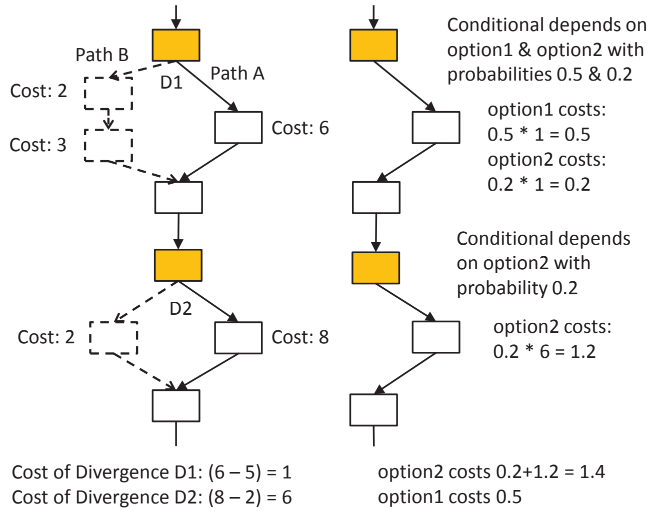

Path B
Path A
D1
Cost: 2
Cost: 3
Cost: 6
D2
Cost: 2
Cost: 8
Cost of Divergence D1: (6 – 5) = 1
Cost of Divergence D2: (8 – 2) = 6
Conditional depends on option1 & option2 with probabilities 0.5 & 0.2
option1 costs: 0.5 * 1 = 0.5
option2 costs: 0.2 * 1 = 0.2
Conditional depends on option2 with probability 0.2
option2 costs: 0.2 * 6 = 1.2
option2 costs 0.2+1.2 = 1.4
option1 costs 0.5

Figure 4: Example of differential performance summarization

system calls executed were allowed to differ, X-ray could provide incorrect root cause diagnoses.

On the other hand, the fidelity of replay must be low enough so that X-ray can execute application instructions and system calls without dynamic instrumentation during the recorded execution but execute *both* application and dynamic instrumentation instructions and system calls during replay. From the point of view of the replay system, the replayed execution will contain a large number of additional events that were not present during recording.

Thus, the design of X-ray walks a fine line. The fidelity of deterministic replay must guarantee that the same *application* instructions and system calls are executed in the same order in all executions, but also allow replays to execute additional *instrumentation* instructions and system calls. These requirements preclude off-the-shelf use of any existing deterministic replay system. Some systems do not guarantee the same sequence of application instructions [2, 24], while others do not allow recorded and replayed executions to diverge sufficiently to run instrumentation code in one execution but not the other [39] or have unacceptably high recording overhead [25].

Our approach to solving this dilemma is co-design: we make the deterministic replay system *instrumentation-aware* so that it compensates for the specific divergences in replayed execution caused by the dynamic instrumentation. The X-ray replay system is designed to work with the Pin dynamic instrumentation tool; the replay code compensates for extra system calls made by Pin and the modifications to recorded system calls due to instrumentation. It also preallocates resources such as memory regions and signal handlers to avoid conflicts between the instrumentation and the replayed application. Instrumentation-awareness enables our replay system to provide the exact fidelity required by X-ray. Section 6.1.1 describes the detailed implementation. Since our current code only handles single-threaded applications, this section also explains how we are extending X-ray replay multi-threaded applications.

6. IMPLEMENTATION

We next describe the implementation of X-ray in detail.

6.1 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 two activities performed online are recording non-deterministic inputs and gathering performance information.

6.1.1 Deterministic record and replay

X-ray implements deterministic record and replay in the Linux kernel. The unit of replay can be either a single process or a group of communicating processes. Thus, X-ray records and replays one or more applications executing on the same computer.

X-ray currently uses a standard design to record and replay single-threaded processes. It takes a checkpoint (address space and registers) of the process or processes being recorded. For each such process, X-ray logs the data returned by all system calls the process executes on the production system. The logged values include addresses modified by the kernel within the process's address space. X-ray also records the value and timing of signals delivered to each process. When recorded processes spawn child processes, X-ray records the activities of the children — this is useful for servers that use children to handle incoming requests.

To replay a recorded execution, X-ray restarts the application from the checkpoint. When the application makes a system call, X-ray does not re-execute that call. Instead, it supplies the recorded values from the log of non-deterministic events. The exception to this rule is system calls such as `mmap` that change the address space of the application — such calls are executed by the replaying kernel in a manner that ensures that they produce an identical effect on the calling process's address space that was produced during recording. X-ray also delivers the same signals to each process at the point the original signal was received in the recorded execution. This guarantees high fidelity replay; i.e., that the recorded and replayed processes execute the same instructions and system calls in the same sequential order.

X-ray analysis tools use Pin [20] to monitor information flow and attribute performance costs to specific application activities. While Pin is designed to be invisible to the application being instrumented, it is *not* designed to be transparent to lower layers of the system such as the operating system. For instance, Pin adds and modifies system calls, modifies signal handlers, and reserves memory addresses in the application address space.

X-ray compensates for divergences in execution due to binary instrumentation. It allocates memory for use as a communication channel between the kernel replay system and the analysis tools run by Pin. An analysis tool uses this region to inform the kernel which system calls are initiated by the application (and hence should be replayed from the log) and which are initiated by Pin or the analysis tool (and should be executed normally). X-ray intercepts all system calls issued by the applications and sets a flag in this region prior to issuing the system call; it clears the flag when the system call ends. Thus, when the kernel sees a system call with the flag cleared, it knows that Pin or the analysis tool has issued the system call.

X-ray also compensates for interference between system calls made by the recorded application and system calls made by Pin or the analysis tool. For instance, we observed that Pin would sometimes ask the kernel to `mmap` a free region of memory and the kernel would return a region that would later be requested by the recorded application, leading to a conflict. We compensated for this by scanning the log to identify all regions that will be requested by the recorded application during the replay and reserving these regions so that Pin does not ask for them and the kernel does not return them. We made similar modifications to compensate for conflicting requests for signal handlers and other resources that could potentially be requested by both the application and by the dynamic instrumentation.

We are currently modifying X-ray to support multi-threaded applications. The biggest challenge has been supporting the needed fidelity of deterministic record and replay while adding low overhead to the production system. Several recent deterministic replay systems have lowered record overhead for multi-threaded processes running on multiprocessors by searching either online [34] or offline [2, 24] for a replayed execution that is equivalent only in external output to the recorded system. Like these prior systems, we plan to record system calls and user-level synchronization operations. During replay, we can enforce the same *happens-before* order among these operations that was observed during recording. In the absence of data races, this guarantees that the same sequence of instructions and system calls is executed by each pair of corresponding record/replay threads.

To deal with data races, we plan to run a dynamic data race detector during offline replay; we expect that the relative performance impact of this additional step will be small because we already execute high-overhead dynamic instrumentation during replay. During performance summarization, X-ray will assign lower confidence to values accumulated from regions of code in which the executing thread is racing with another thread. The range of the potential error can be estimated by sampling different interleavings of racing instructions during replay. X-ray users can either use the lower-confidence results, or they can add annotation or synchronization to the application to eliminate the data races.

6.1.2 Recording performance information

During the online phase, X-ray also records timing information. For each system call executed by the application, the X-ray kernel records the system time at kernel entry and exit. For simplicity, the kernel writes the timing information for each system call to the same log that it uses to store non-deterministic events. Analysis tools read the log directly to extract the timing information during replay. Other performance information, such as the number of bytes read or written during I/O system calls are already captured as a result of recording sources of non-determinism.

6.2 Offline phase

X-ray executes analysis in two passes. In the first pass, X-ray performs request extraction to determine when each application process is handling each request. It also identifies which basic blocks are executed within the analysis scope chosen by the user and attributes performance costs to those blocks. In the second pass, X-ray attributes basic block execution to specific root causes and summarizes the performance cost for each cause. Since X-ray operates on a previously-recorded execution, it is trivial to replay the execution multiple times so that different parts of the analysis can be executed sequentially (much like a multi-pass compiler).

6.2.1 Request extraction

During the request extraction phase, X-ray identifies the intervals of application execution during which each request was processed. For many types of analysis, X-ray must understand how an application processes one or more particular requests such as particular mail messages for the Postfix mail server or Web requests for Apache. Request extraction traces the causal path of each request from the point when the request is received by the application to the point when the request terminates (e.g., when a server sends the response). Often, requests traverse multiple processes, and different processes handle different requests at the same time.

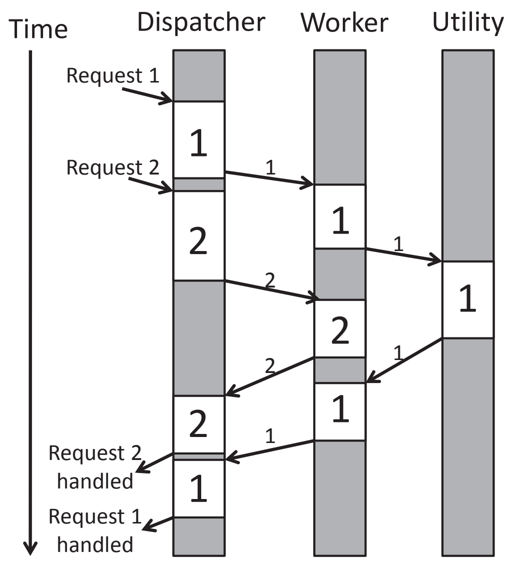


Figure 5: An example of X-ray request extraction. The intervals marked as 1 or 2 in each process correspond to the portions of process execution that X-ray associates with the first and second requests, respectively.

The notion of a request is application-dependent. Thus, X-ray requires a per-application filter that specifies the boundaries of incoming requests. The filter is simply a regular expression over incoming data. For instance, the Postfix filter looks for the string `HELO` to identify incoming mails. A filter only needs to be created once for each protocol (e.g., SMTP or HTTP).

Request extraction runs as a Pin tool. The tool examines values returned from all system calls that provide external input such as those that receive data from the network. When the data returned from such system calls match the specified filter, X-ray tags the receiving process with a unique request identifier to show that it is handling the request in question.

As shown in Figure 5, X-ray propagates request tags among processes as they communicate. It currently assumes that each process handles a single request at a time, but it allows multiple processes to concurrently handle different requests (for instance, the dispatcher handles request 2 while a worker handles request 1 in the figure). When a message with a new tag is received by a process, X-ray assumes that it ceases to handle the old request and starts to handle the new one. This assumption is valid for the server applications we use in the evaluation.

Note that since these processes are being replayed, the X-ray kernel does not actually send and receive data when they execute system calls. Therefore, request extraction cannot use existing communication channels to propagate request tags. Instead, X-ray modifies the application binaries to establish and use special *side channels* (replay-specific TCP connections) for communicating request tags with each other. Since side channels are established by instrumentation and not by the application, the kernel executes side channel system calls during replay. During replay, when the instrumentation sees that one recorded process communicated with another, it uses the side channel to transmit any current request tag from the sending process to the process that received the data during recording. The receiving process blocks until information is available on the side channel. This means that the replayed processes obey the same causal order of execution that they followed during recording.

Although most popular servers such as Apache, Postfix or PostgreSQL handle a single request per thread of execution, event-based servers may handle many requests simultaneously using a single thread. Since X-ray already tracks application data flow, we plan to extend X-ray to handle such servers via fine-grained information flow analysis (i.e., taint tracking). Essentially, we can identify the memory addresses associated with each request and use that information to identify the code intervals in which a thread or process is handling a particular request. Alternatively, we could use per-application schemas as is done during Magpie request extraction [5].

As the replayed application processes execute, the request extraction Pin tool tags each basic block with a request identifier if it believes the process is handling a request at that time. The final output of the request-extraction instrumentation is a per-request list of <process,basic block> tuples in the order that the basic blocks were executed.

6.2.2 *Identifying basic blocks*

The first step in performance summarization is to map the scope of the analysis specified by the user to a set of basic blocks. If the user specifies the scope as a time interval, X-ray includes all basic blocks executed by any process within that interval. Identification is somewhat imprecise because X-ray only records timestamps at the entry and exit of system calls. The analyzed scope is from the exit of the last system call executed before the specified interval to the entry of the first system call executed after the specified interval. If the analysis scope is a time interval, X-ray omits request extraction because it is not needed.

If the user specifies a particular request as the scope of analysis, X-ray uses the request extraction results that identify the set of basic blocks for that request. If the user specifies two requests to compare using differential performance analysis, X-ray uses the request extraction results for both requests.

6.2.3 *Attributing performance costs*

X-ray next attributes specific performance costs to events (application instructions and system calls executed). As a performance optimization, X-ray considers all events in the same basic block together since they have the same set of root causes (in other words, if one event is executed, they all must be executed).

Currently, users may choose one or more of the following metrics: latency, CPU utilization, disk bandwidth, and network throughput. During recording, X-ray records the start and end time of every system call in the log of non-deterministic events. When it encounters the same system call during replay, the Pin tool reads the log and subtracts the two values to determine the system call latency. The latency is then attributed to the basic block that invoked the system call.

X-ray next considers latency not attributable to system calls. It currently uses a simple method that attributes latency in proportion to the number of user-level instructions executed. X-ray then takes the total process execution time, subtracts the time spent in system calls, and divides the remaining time by the number of instructions. The result is the latency per instruction. Multiplying this value by the number of instructions in a basic block and adding in any system call latency for that block gives the block's total latency.

To calculate CPU utilization, X-ray simply counts the number of instructions executed by each basic block. To calculate disk and network usage, it inspects the replay log as it is replayed to identify file descriptors associated with the resource being analyzed. When a system call reads or writes data for these descriptors, X-ray attributes the total number of bytes processed to the basic block that invoked the system call.

```
/* a, b, c and d are read from the config file*/
if (c == 0) { /* c set to 0 in config file */
    x = a;
} else {
    y = b;
}
z = d;
```

Figure 6: Example to illustrate data and control flow tracking

6.2.4 *Information flow analysis*

X-ray next determines why each basic block executed. X-ray uses taint tracking [22], a form of dynamic information flow analysis, to associate each block with a set of root causes. More specifically, it uses our tool ConfAid [3] to generate a set of probable root causes for each block. We next provide some background on ConfAid.

ConfAid assigns a unique taint identifier to registers and memory addresses when data is read into the program from configuration files and incoming request sockets. It identifies specific configuration tokens through a simple form of symbolic execution. For instance, if data read from a known configuration file is compared to the string "FOO", then ConfAid marks that data as associated 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. Rather than track taint as a binary value, it associates a weight with each taint identifier that represents the strength of the causal relationship between the tainted value and the root cause. X-ray builds on ConfAid by also assigning a weighted set of taint values to each basic block that is executed; membership in this set indicates that the block's execution depends on the specified root cause, and the associated weight indicates the strength of the dependency.

ConfAid specifies the taint of each variable as a set of configuration options. For instance, if the taint set of a variable is $\{\texttt{FOO}, \texttt{BAR}\}$, ConfAid believes that the value of that variable could change if the user were to modify either the `FOO` or `BAR` tokens in the configuration file.

When a monitored process executes an instruction that modifies a memory address, register, or CPU flag, the taint set of each modified location is set to the union of the taint sets of the values read by the instruction. For example, consider the instruction $x = y + z$ where the taint set of x becomes the union of taint sets of y and z. Intuitively, the value of x might change if a configuration token were to cause y or z to change prior to the execution of this instruction.

ConfAid tracks control flow dependencies as well since they propagate the majority of configuration-derived taint. To do so, ConfAid takes into account the basic block structure of an application. Consider the example in Figure 6. Assume a, b, c, and d were read from a configuration source and have taint sets assigned to them. The taint of variable x not only includes the taint of a via data flow, but also the taint of condition c, since the value of condition c could affect the execution path and consequently the value of variable x. The taint of variable z, however, only includes the taint of d and not c, since the execution of $z = d$ statement happens regardless of value of c.

ConfAid also tracks implicit control flow dependencies. In Figure 6, the values of *x* and *y* both depend on *c* since the occurrence of their assignments to *a* and *b* depend on whether or not the branch is taken. Note that *y* is still dependent on *c* even though the `else` path is not taken by the execution since the value of *y* might change if a configuration token is modified such that the condition evaluates differently.

In contrast to prior taint tracking tools, ConfAid tracks taint as a floating-point weight ranging in value between zero and one. ConfAid uses two heuristics. First, it assumes that data flow dependencies are more likely to lead to the root cause than control flow dependencies. Second, it assumes that control flow dependencies are more likely to lead to the root cause if they occur closer to the basic block being executed.

We modified ConfAid to better suit the needs of X-ray. Our first modification was to broaden the source of tainted data. X-ray not only tracks data read from configuration sources; it also tracks data read from input requests. X-ray uses the same filter that it uses during request extraction to determine when the application is reading data from a request. The taint identifier in this case indicates the particular request on which a memory address or register depends.

ConfAid originally transmitted taint values over the same channels that are used to send the tainted data between processes. However, these channels do not exist during replay since the kernel does not re-execute recorded system calls for inter-process communication. We therefore modified ConfAid to create and use side channels to transmit taint values, as described in the previous section on request extraction.

Finally, we modified how ConfAid uses taint values. The original ConfAid implementation only outputs taint values when it encounters an application failure. However, X-ray is interested in the taint values of all instructions and system calls executed within the scope of analysis. During execution, our modified version of ConfAid generates a taint set that contains root causes and associated weights for every basic block that has been marked as being within the scope of analysis.

As an example, our modified version of ConfAid might emit the following taint set for a basic block: {`FOO:1.0`, `BAR:0.5`}. This represents the belief that the basic block would definitely not have been executed if root cause `FOO` were different and the belief that the block is 50% likely not to have been executed if root cause `BAR` were different. Note that these are two independent probabilities: potentially changing either of the two options might cause the basic block to not have been executed. Thus, the values in a taint set need not sum to one.

6.2.5 *Integration*

Next, X-ray attributes the performance cost of executing each basic block according to specific root causes. For each root cause in the block's taint set, X-ray multiplies the per-block cost by the weight associated with the root cause. Each process maintains a running sum of the costs associated with each root cause as it is replayed. The final cost for each root cause is determined by adding together the sums from all replayed processes. At the end of analysis, X-ray prints out a list of root causes and shows the estimated performance cost for each. X-ray can simultaneously analyze multiple performance metrics.

6.2.6 *Differential performance summarization*

X-ray uses a different method to compare the performance of two requests. It first identifies the points where the execution paths diverged from one another. It uses the results of request extraction to output each path as a sequence of basic blocks executed by the request. Each path may span multiple processes. X-ray then uses the `diff` tool to compare the two paths and understand where they diverged from one another and where the divergence ended as the paths merged back together.

X-ray then determines the root cause of each divergence. It attributes the cost of the divergence to the conditional that immediately preceded the divergence. It calculates a performance cost for the divergence by first summing the performance costs of all basic blocks along the divergent path for one request and then subtracting the sum of the performance costs of all basic blocks along the divergent path for the other request. It attributes the divergence to root causes by multiplying the cost of the divergence by the weights in the taint set for the conditional that caused the divergence.

7. EVALUATION

Our evaluation of X-ray answers the following questions:

- How accurately does X-ray identify the root cause of performance problems?
- How fast can X-ray troubleshoot a performance problem?
- How much overhead does X-ray add to a production system?

7.1 Experimental Setup

We used X-ray to diagnose performance problems in three applications: the Apache Web server version 2.2.14, the Postfix mail server version 2.7 and the PostgreSQL database version 9.0.4. In Apache, each request is handled by one process. Postfix has multiple utility processes, each of which is responsible for handling a certain part of a request. On average, a Postfix request is handled by 5 different processes. In PostgreSQL, each request is handled by one process. However, PostgreSQL has multiple time-based utility processes such as a write-ahead log writer and an auto-vacuum that handle requests in batches. 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.

7.2 Root cause identification

We evaluated X-ray by recreating known performance issues reported in application performance tuning and troubleshooting Web pages, forums, and blog posts. To recreate each issue, we either modified configuration settings or sent a problematic sequence of requests to the server. In total, we recreated the 14 problems described in Table 1 (7 for Apache, 3 for Postfix, and 4 for PostgreSQL).

For each test case, we recorded server execution while we sent several application requests. We used standard lightweight performance monitoring tools such as top, iostat, netstat and log files to identify the bottleneck resource and identify requests during which resource usage was high. Later, we executed X-ray offline analysis of the recorded runs to explain the performance anomalies.

For each test case, Table 2 shows the scope and metric we used for X-ray analysis. The next column shows the top three root causes identified by X-ray, along with X-ray's analysis of how much the cause contributed to the performance metric under observation. The correct answers for each test case is shown in bold. The last column shows how long X-ray offline analysis took.

7.2.1 *Apache*

In the first Apache test case, the threshold for the number of requests that can reuse the same TCP connection is set too low. Reestablishing a connection causes some requests to exhibit higher latency than others. To exhibit this problem, we sent 100 various

App	#	Description of performance test cases
Apache	1,2	Apache sets a threshold for the number of requests that are handled in one TCP connection using the KeepAlive and MaxKeepAliveRequests setting. A low threshold causes Apache to shut down and rebuild the connections too often, causing a significant delay in handling some requests.
	3	In Apache, access to various directories can be controlled in the config file based on the domain name of the client sending the request. This setting causes extra DNS calls for verifying the domains and leads to high latency in handling the requests.
	4	Apache can be configured to log the host names of clients sending requests to specific directories for administrative purposes. This setting causes extra DNS calls and leads to high latencies in handling requests for those directories.
	5	Apache can be configured to require authentication for some directories. Authentication causes high CPU usage peaks.
	6	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.
	7	By default, Apache sends eTags in the header of HTTP responses. The eTags can be used by the client in future requests for the same file to only receive the file if its contents have changed.
Postfix	1	Postfix can be enabled to log more information for a list of specific hosts, using debug_peer_list option. The extra logging causes excessive disk activity.
	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.
	3	Postfix can be configured to reject requests that are sent from blacklisted domains. Postfix uses DNS mechanism to query blacklist operators to determine if the message should be rejected. Based on the number of operators specified, Postfix performs extra DNS calls, which significantly increases the latency of the handled message.
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 5x.
	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.
	3	The frequency of taking checkpoints from the write-ahead log can be configured in the PostgreSQL configuration file. Having more frequent checkpoints decreases crash recovery time but significantly increases disk activity for busy databases.
	4	The delay between the activity rounds of the write-ahead log write process can be configured in PostgreSQL configuration file. Setting this delay higher causes potential loss of transactions. However, lower delays cause extra CPU usage.

Table 1: Description of the Apache, Postfix and PostgreSQL performance test cases

requests to the Apache server using the *ab* Apache benchmarking tool. The requests used different HTTP methods (GET and POST) and asked for files with different sizes.

We first used X-ray to perform a differential performance summarization of two similar requests (HTTP GETs of small files), one of which had a small latency and one of which had a high latency. X-ray correctly identified the `MaxKeepAliveRequests` and `KeepAlive On` tokens as the largest root causes. As with many issues we examined, there are multiple ways to fix the problem: in this case, changing either token value removes the performance anomaly.

Next, we explored how sensitive X-ray is to the similarity of the compared requests (Apache test case 2). We compared two very dissimilar requests using differential performance summarization: a small HTTP POST and a large HTTP GET. As would be expected, X-ray reported that the largest cause of the divergence in processing time was due to the input data from the requests. The `DocumentRoot` parameter is also reported as a large cause of the divergence because the root is appended to the input file name. However, X-ray still reported that the `MaxKeepAliveRequests` is a substantial reason for divergence. Further, the estimated performance impact of `MaxKeepAliveRequests` is not affected much by the similarity of the requests.

This test case highlights the power of differential performance summarization. X-ray does not require two requests to be substantially similar in order to identify performance anomalies. Because it analyzes program control flow, X-ray can correctly differentiate performance differences due to diverging input from those due to other root causes such as configuration options.

The remaining Apache test cases use both basic and differential performance summarization for a variety of metrics (latency, CPU usage, and network throughput). In every case, X-ray identified the correct root cause (or causes) of the performance problem. While the root cause of the first six performance problems were configuration setting, the high network usage in the last test case was due to one client's failure to use the eTag header. This last case shows that X-ray's analysis of server performance can sometimes identify inefficient client behavior.

X-ray analysis time for the 7 test cases varies between 2 and 3 minutes. This is very reasonable considering that analysis is performed offline and does not affect the online production software.

7.2.2 Postfix

The first Postfix test case reproduces a problem reported in a

App	#	Analysis scope	Analysis Metric	Results : Expected contribution	Exec. time
Apache	1	Differential	Latency	**MaxKeepAliveRequests**: 17.2 usecs. **KeepAlive On**: 8.6 usecs. <Directory: 4.7 usecs.	2m 40s
	2	Differential	Latency	**User's request**: 311.6 usecs. **DocumentRoot**: 311.5 usecs. **MaxKeepAliveRequests**: 16.8 usecs.	2m 41s
	3	Differential	Latency	**Allow domain.com**: 603 usecs. ServerRoot: 151 usecs. TypesConfig : 151uses	2m 14s
	4	Differential	Latency	**HostNameLookups On**: 254 usecs. <Directory: 127 usecs. **HostNameLookups**: 127 usecs.	2m 4s
	5	Request	CPU	**AuthUserFile**: 9M instrs. **User's request**: 600K instrs. Listen: 80K instrs.	2m 6s
	6	Differential	CPU	**ContentDigest On**: 217K instrs. **ContentDigest**: 108K instrs. <Directory: 108K instrs.	2m 6s
	7	Differential	Network	**User's request**: 35 KB **DocumentRoot**: 35 KB <Listen: 4 KB	2m 4s
Postfix	1	Request	Disk	**User's request**: 100 KB **debug_peer_list**: 28 KB queue_directory: 5 KB	1m 18s
	2	Request	CPU	**body_checks**: 1M instrs. **User's request**: 900K instrs. myhostname: 300K instrs.	2m 49s
	3	Request	Latency	**reject_rbl_client**: 3.5 secs. **reject_rbl_client**: 1.9 secs. **smtpd_client_restrictions**: 0.9 secs.	1m 24s
PostgreSQL	1	Time interval	CPU	**timezone**: 28M instrs. default_text_search_config: 11M instrs. datestyle: 11M instrs.	15+m
	2	Request	Latency	shared_buffers: 0.42 secs. max_connections: 0.26 secs. **wal_sync_method**: 0.26 secs.	2m 50s
	3	Time interval	Disk	**checkpoint_timeout**: 16 KB shared_buffers: 11 KB max_connections: 11 KB	4m 48s
	4	Time interval	CPU	shared_buffers: 2.6M instrs. max_connections: 2M instrs. **wal_writer_delay**: 1.4M instrs.	5m 27s

Table 2: The results for our performance test cases.

Postfix user's blog [26]. The user noticed that emails with attachments sent from his account transferred very slowly, while everything else, including the mail received by IMAP services, had no performance issues.

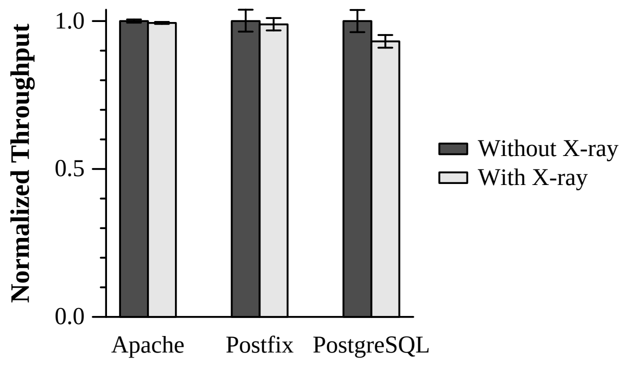
The user employed `iotop` to monitor the Postfix server, and observed that one child process was generating a lot of disk activity. He poured through the server logs and realized that the child process was logging large amounts of data. Finally, he ran through his configuration file, and eventually found out that the `debug_peer_list`, which specifies a list of hosts that triggered the logging, included his own IP address.

Our results show that X-ray can make this diagnosis automatically. We simply analyzed a specific request that was associated with a period of high disk usage. X-ray identifies both the request (since it contains the IP address that caused excessive logging) and the erroneous parameter as the top two root causes, pinpointing the specific reasons for the high disk activity. Note that we did not have to identify which child process was responsible for the logging, nor did we have to read any log files. Since X-ray produced these results in a little over a minute, our tool could have saved the blogger considerable time.

The remaining two Postfix test cases reproduce CPU and latency problems. X-ray identifies the correct root cause for each problem in only a few minutes.

7.2.3 PostgreSQL

The first PostgreSQL case study is based on our own experience. Our evaluation started and stopped PostgreSQL many times. We noticed that our scripts were running slowly due to application start-up delay, and decided to try to use X-ray to improve perfor-



This figure compares server throughput with and without X-ray recording. Results are normalized to the number of requests per second without X-ray. Higher values are better. Each result is the mean of 10 trials; error bars are 95% confidence intervals.

Figure 7: X-ray online overhead

mance. Since `top` showed 100% CPU usage, we performed a X-ray CPU analysis during the interval before PostgreSQL received the first request.

Unexpectedly, X-ray identified the `timezone` configuration option as the top root cause. In the configuration file, we had set the `timezone` option to `unknown`. This caused PostgreSQL to expend a surprising amount of effort to attempt to identify the correct time zone. We updated the configuration to specify our time zone, and were pleased to see that the application startup time decreased by over 80%. While this problem is admittedly esoteric since most PostgreSQL users will not start and stop the application several times in succession, we were happy to see that X-ray could help identify performance issues that we did not specifically inject into the application.

Since PostgreSQL utility processes are mostly asynchronous (they sleep for a while and then wake up to perform tasks such as flushing write-ahead log to disk, taking checkpoints, or vacuuming the database) time interval analysis is the best fit for this application. When we examined three other performance issues that affect PostgreSQL throughput, X-ray identified the correct root cause in one instance and ranked the correct cause third in the other two cases. The `shared_buffers` and `max_connections` parameters appear to taint many branches during PostgreSQL execution causing them to always rank as top causes of resource usage.

X-ray analysis time is currently capped at 15 minutes; analysis of the first test case hit this limit but still returned meaningful results since the analysis executed almost all the code used during startup. The remaining PostgreSQL issues required 2–5 minutes to analyze. We have not yet put much effort into optimizing X-ray analysis performance, since these times are still substantially faster than manual performance debugging.

7.3 X-ray online overhead

We measured online overhead by comparing the throughput and the latency of our three applications when they are recorded by X-ray to results running the applications on the default Linux kernel without recording.

Figure 7 shows that X-ray adds a 1–7% throughput overhead for the three applications. For Apache, 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. X-ray recording reduced throughput by 0.6%. Per-request latency increased by 0.6%. The recording log size for this experiment was 7 MB, containing 115K system calls.

For Postfix, we used the `smtp-source` tool to send 10000 mail messages of size 1 KB from another machine on the isolated network. Postfix processing is asynchronous, so there is no meaningful latency measure. X-ray recording reduced server throughput by 1.1%. The log size was 453 MB, containing 6 million system calls.

We benchmarked PostgreSQL using `pgbench`. We measured the number of transactions completed in 60 seconds with concurrency of 10 transactions sent at a time. Each transaction involves one `SELECT`, three `UPDATE`s, and one `INSERT` command. X-ray recording reduced throughput by 7% and increased per-request latency by 7%. The log size was 820 MB, containing 17 million system calls. We conjecture that the higher overhead for PostgreSQL was mostly due to the increased log size and larger number of system calls.

8. CONCLUSION

Diagnosing performance problems in production systems is challenging. X-ray helps system administrators by identifying the root cause of observed performance problems. X-ray first records the execution of the production system and collects performance information. In an offline phase, X-ray deterministically replays the recorded execution and performs heavyweight analysis. X-ray uses dynamic information flow analysis to attribute the recorded performance information to root causes that include configuration options and request inputs. Our results show that X-ray accurately identifies the root cause of several real-world performance problems, while imposing only 1–7% overhead on a production system.

Acknowledgments

We thank the anonymous reviewers of this report. This research was supported by NSF award CNS-1017148. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NSF, the University of Michigan, or the U.S. government.

9. 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 *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, October 2003), pp. 74–89.

[2] ALTEKAR, G., AND STOICA, I. ODR: Output-deterministic replay for multicore debugging. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles* (October 2009), pp. 193–206.

[3] ATTARIYAN, M., AND FLINN, J. Automating configuration troubleshooting with dynamic information flow analysis. In *Proceedings of the 9th OSDI* (Vancouver, BC, October 2010).

[4] B. URGAONKAR, P. S., AND ROSCOE, T. Resource overbooking and application profiling in shared hosting platforms. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation* (Boston, MA, December 2002), pp. 239–254.

[5] BARHAM, P., DONNELLY, A., ISAACS, R., AND MORTIER, R. Using Magpie for request extraction and workload modelling. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation* (San Francisco, CA, December 2004), pp. 259–272.

[6] BHATIA, S., KUMAR, A., FIUCZYNSKI, M. E., AND PETERSON, L. Lightweight, high-resolution monitoring for troubleshooting production systems. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation* (San Diego, CA, December 2008), pp. 103–116.

[7] BRESSOUD, T. C., AND SCHNEIDER, F. B. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems 14*, 1 (February 1996), 80–107.

[8] Cadar, C., Dunbar, D., and Engler, D. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Usenix Symposium on Operating System Design and Implementation (OSDI)* (December 2008), pp. 209–224.

[9] Cantrill, B. M., Shapiro, M. W., and Leventhal, A. H. Dynamic instrumentation of production systems. In *Proceedings of the USENIX Annual Technical Conference* (Boston, MA, June 2004), pp. 15–28.

[10] Chen, H., Jiang, G., Zhang, H., and Yoshihira, K. Boosting the performance of computing systems through adaptive configuration tuning. In *Proceedings of the 2009 ACM symposium on Applied Computing* (Honolulu, Hawaii, March 2009), pp. 1045–1049.

[11] Chen, M. Y., Accardi, A., Kiciman, E., Lloyd, J., Patterson, D., Fox, A., and Brewer, E. Path-based failure and evolution management. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI)* (San Francisco, CA, March 2004).

[12] Chen, M. Y., Kiciman, E., Fratkin, E., Fox, A., and Brewer, E. Pinpoint: Problem determination in large, dynamic Internet services. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)* (Bethesda, MD, June 2002), pp. 595–604.

[13] Chipounov, V., Kuznetsov, V., and Candea, G. S2e: A platform for in vivo multi-path analysis of software systems. In *ASPLOS* (March 2011).

[14] 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 *Proceedings of the 6th Symposium on Operating Systems Design and Implementation* (San Francisco, CA, December 2004), pp. 231–244.

[15] 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.

[16] 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 *Proceedings of the 5th Symposium on Operating Systems Design and Implementation* (Boston, MA, December 2002), pp. 211–224.

[17] Fonseca, R., Porter, G., Katz, R. H., Shenker, S., and Stoica, I. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th NSDI* (Cambridge, MA, April 2007), pp. 271–284.

[18] Geels, D., Altekar, G., Shenker, S., and Stoica, I. Replay debugging for distributed applications. In *Proceedings of the USENIX 2006 Annual Technical Conference* (Boston, MA, June 2006).

[19] Kasick, M. P., Tan, J., Gandhi, R., and Narasimhan, P. Black-box problem diagnosis in parallel file systems. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies* (San Jose, CA, February 2010).

[20] 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 *Programming Language Design and Implementation* (Chicago, IL, June 2005), pp. 190–200.

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

[22] Newsome, J., and Song, D. Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software. In *In Proceedings of the 12th Network and Distributed Systems Security Symposium* (February 2005).

[23] http://oprofile.sourceforge.net/.

[24] 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 *Proceedings of the 22nd SOSP* (October 2009), pp. 177–191.

[25] Patil, H., Pereira, C., Stallcup, M., Lueck, G., and Cownie, J. PinPlay: A framework for determinisrtic replay and reproducible analysis of parallel programs. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (March 2010).

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

[27] 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* (Ottawa, ON, Canada, July 2005), pp. 49–64.

[28] Ruan, Y., and Pai, V. Making the "box" transparent: System call performance as a first-class result. In *Proceedings of the USENIX Annual Technical Conference* (Boston, MA, June 2004), pp. 1–14.

[29] 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 *Proceedings of the 8th NSDI* (Boston, MA, March 2011), pp. 43–56.

[30] Srinivasan, S., Andrews, C., Kandula, S., and Zhou, Y. Flashback: A light-weight extension for rollback and deterministic replay for software debugging. In *Proceedings of the 2004 USENIX Technical Conference* (Boston, MA, June 2004), pp. 29–44.

[31] Stewart, C., and Shen, K. Performance modeling and system management for multi-component online services. In *Proceedings of the Second Symposium on Networked Systems Design and Implementation (NSDI)* (Boston, MA, May 2005).

[32] Su, Y.-Y., Attariyan, M., and Flinn, J. AutoBash: Improving configuration management with operating system causality analysis. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles* (Stevenson, WA, October 2007), pp. 237–250.

[33] Veeraraghavan, K., Flinn, J., Nightingale, E. B., and Noble, B. quFiles: The right file at the right time. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies* (San Jose, CA, February 2010), pp. 1–14.

[34] Veeraraghavan, K., Lee, D., Wester, B., Ouyang, J., Chen, P. M., Flinn, J., and Narayanasamy, S. DoublePlay: Parallelizing sequential logging and replay. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems* (Long Beach, CA, March 2011).

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

[36] Wang, H. J., Platt, J. C., Chen, Y., Zhang, R., and Wang, Y.-M. Automatic misconfiguration troubleshooting with PeerPressure. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation* (San Francisco, CA, December 2004), pp. 245–257.

[37] 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 *Proceedings of the USENIX Large Installation Systems Administration Conference* (October 2003), pp. 159–172.

[38] Whitaker, A., Cox, R. S., and Gribble, S. D. Configuration debugging as search: Finding the needle in the haystack. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation* (San Francisco, CA, December 2004), pp. 77–90.

[39] Xu, M., Malyugin, V., Sheldon, J., Venkitachalam, G., and Weissman, B. ReTrace: Collecting execution trace with virtual machine deterministic replay. In *Proceedings of the 2007 Workshop on Modeling, Benchmarking and Simulation (MoBS)* (June 2007).

[40] 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 *Proceedings of the 8th NSDI* (Boston, MA, March 2011), pp. 57–70.

[41] Zheng, W., Bianchini, R., and Nguyen, T. D. Automatic configuration of Internet services. In *Proceedings of the European Conference on Computer Systems* (Lisbon, Portugal, March 2007), pp. 219–229.