Reference-Driven Performance Anomaly Identification*

Kai Shen Christopher Stewart Chuanpeng Li Xin Li
University of Rochester, Rochester, NY, USA
{kshen@cs, stewart@cs, cli@cs, xinli@ece}.rochester.edu

ABSTRACT

Complex system software allows a variety of execution conditions on system configurations and workload properties. This paper explores a principled use of *reference executions*—those of similar execution conditions from the target—to help identify the symptoms and causes of performance anomalies. First, to identify anomaly symptoms, we construct change profiles that probabilistically characterize expected performance deviations between target and reference executions. By synthesizing several single-parameter change profiles, we can scalably identify anomalous reference-to-target changes in a complex system with multiple execution parameters. Second, to narrow the scope of anomaly root cause analysis, we filter anomaly-related low-level system metrics as those that manifest very differently between target and reference executions. Our anomaly identification approach requires little expert knowledge or detailed models on system internals and consequently it can be easily deployed. Using empirical case studies on the Linux I/O subsystem and a J2EE-based distributed online service, we demonstrate our approach's effectiveness in identifying performance anomalies over a wide range of execution conditions as well as multiple system software versions. In particular, we discovered five previously unknown performance anomaly causes in the Linux 2.6.23 kernel. Additionally, our preliminary results suggest that online anomaly detection and system reconfiguration may help evade performance anomalies in complex online systems.

Categories and Subject Descriptors

D.4.8 [**Performance**]: Measurements, Modeling and prediction

General Terms

Experimentation, Measurement, Performance, Reliability

Keywords

Performance anomaly, Operating system

*This work was supported in part by NSF CAREER Award CCF-0448413, grants CNS-0615045, CCF-0621472, and by an IBM Faculty Award.

SIGMETRICS/Performance'09, June 15–19, 2009, Seattle, WA, USA.

1. INTRODUCTION

Large system software such as operating systems and distributed system middleware are complex in terms of their implementation, their supported configuration settings, and their wide ranging workloads. Under certain execution conditions, these systems may deliver below-expectation *anomalous* performance. Besides causing performance degradation, such performance anomalies also compromise the predictability of system behaviors [11, 15, 17, 18], which is important for automatic system management.

Compared to program crashes and correctness violations, performance anomalies are more difficult to identify because *normal performance behaviors* are not always easily known or even clearly defined. Further, performance anomalies typically relate to high-level system semantics and they do not possess common source-level patterns such as accessing invalid pointers. Despite widespread anecdotal observations of performance problems, relatively few of them are clearly identified and understood. For instance, we examined CLOSED (*i.e.*, resolved and corrected) bugs concerning Linux 2.4/2.6 IO/storage, file system, and memory management in the Linux bug tracking system [9]. Among 219 reported bugs, only 9 were primarily performance-related (about 4%). Here, we determine a bug to be primarily performance-related if it causes significant performance degradation but it does not cause any incorrect behaviors like system crashes or deadlocks.

This paper explores principled, scalable techniques on using reference executions for performance anomaly analysis. A reference execution is very similar to the targeted anomalous execution in terms of system software, configuration settings, and workload properties (which, we collectively call execution conditions). Like a literary reference, it serves as a basis of comparison that helps understand the commonality and uniqueness of the target.

References can assist in identifying performance anomaly symptoms. We say that a target execution exhibits the symptoms of a performance anomaly if its performance is abnormally lower than that of a reference—*i.e.*, compared to the expected performance deviation between them. Our approach utilizes *change profiles* that probabilistically characterize expected performance deviations between target and reference executions. In a complex system with multiple execution parameters, we first use sampling to derive single-parameter change profiles for a target and its reference that differ only slightly in execution condition. By synthesizing multiple single-parameter change profiles using a generally applicable bounding analysis, we can scalably identify anomalous reference-to-target changes over wide ranges of execution conditions.

Identified reference-target anomaly symptoms can serve as the basis of root cause analysis. In particular, references may help the analysis by filtering anomaly-related system metrics from the large set of collectible metrics in today's systems. Our approach builds

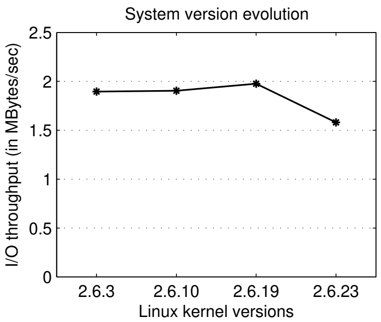

Figure 1: An example of system version evolution anomaly on SPECweb99 with an I/O-bound workload (19.5 GB total data size on a machine with 2 GB memory). The I/O throughput measures the speed of data access at the server application level (Apache 2.0.44). All Linux kernel versions are configured with the `anticipatory` **I/O scheduler.**

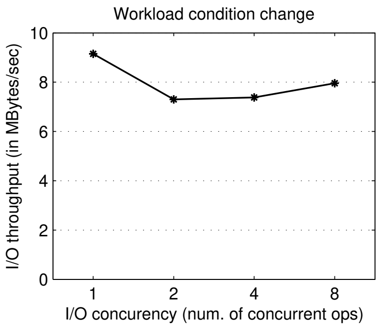

Figure 2: An example of workload condition change anomaly on an I/O microbenchmark that sequentially reads 256 KBytes from random locations in randomly chosen files of 4 MBytes large. The workload is I/O-bound (19.5 GB total data size on a machine with 2 GB memory) and the experiments run on the Linux 2.6.23 kernel configured with the `deadline` **I/O scheduler. We control the number of simultaneously running benchmark instances to adjust the I/O concurrency levels.**

on the intuition that system metrics that manifest very differently between the target anomalous execution and its normal reference are likely related to the performance anomaly. Further, our anomaly identification approach can assist the management of complex online systems. For instance, the online detection of anomaly symptoms coupled with adjustments of anomaly-inducing system configurations may help improve the performance (or evade performance anomalies).

Our reference-driven performance anomaly identification requires little knowledge or detailed models on internal system design and implementation. This allows easy deployment on complex systems supporting a variety of configuration settings and workload conditions. We are able to apply it on two large software systems: the Linux I/O subsystem and the JBoss J2EE distributed application server. Our empirical studies uncovered previously unknown anomaly symptoms and causes over wide ranges of system execution conditions (eight execution parameters for Linux and nine parameters for JBoss) as well as multiple system software versions.

2. ANOMALY SYMPTOM IDENTIFICATION

A performance anomaly arises when the system performance behavior deviates from the expectation. Performance expectations can be made in different ways, such as design-driven models, service-level agreements, or programmer specifications. Our objective is to derive performance expectations that match high-level design principles and that can be intuitively interpreted. For instance, the expected performance of an I/O scheduler is that intended by the high-level scheduling algorithm (*e.g.*, Cyclic-SCAN or anticipatory scheduling [5]). Its behavioral changes under different execution conditions can be intuitively explained according to the scheduling algorithm.

Intuitive interpretability is important for human understanding of the expected performance behavior and it also helps validate the derived expectation. With design-driven expectations, any performance anomaly indicates an implementation deviation from the high-level design. Such deviation may represent unintentional errors, but sometimes the implementation intentionally deviates from the design for simplification or due to a partially beneficial optimization that degrades performance in some cases.

2.1 Motivating Examples and Our Approach

The high-level guidance of reference-driven anomaly symptom identification can be explained in the following way. Given two executions T and R: if T delivers much worse performance than R against the expected performance deviation between them, then you identify the target execution T as anomalous in relation to the reference execution R.

We provide two illustrating examples. Figure 1 shows the performance of the SPECweb99 benchmark with an I/O-bound workload. The I/O throughput over multiple Linux kernel versions indicates anomalous performance degradation (around 20%) from Linux 2.6.19 to Linux 2.6.23. In this example, Linux 2.6.23 (released in October 2007) is anomalous in relation to any of the three earlier kernel versions (released in November 2006, December 2004, and February 2004, respectively) as a reference. In another example, Figure 2 shows the I/O throughput of a simple I/O microbenchmark. The measured throughput at different I/O concurrency levels indicates a suspicious performance drop between the concurrency levels of one and two. In this case, the execution at the concurrency of two is suspected to be anomalous in relation to the serial execution.

It is important to point out that the identified anomalies in the above two examples are not just due to large performance degradations between the target and reference, but that such degradations are against certain expectation. In the first example of the system evolution anomaly, the obvious expectation is that a system version upgrade should not lead to significant performance degradation. In the second example of the workload adjustment anomaly, the somewhat less obvious expectation is that the increase of I/O concurrency should not cause significant performance degradation.

So how to systematically derive these expectations? Our approach is to probabilistically infer expected performance deviations due to changing system execution conditions through random measurements on the real system. The assumption is that commonly observed performance behaviors in real systems likely match high-level design principles and often they can be intuitively explained.

We define the *change profile* for an execution condition change as the probabilistic distribution of resulted performance deviations. Different performance deviation metrics can be employed for the change profile representation. Here we choose one with good illustrating effects. From throughput measurements at old and new conditions (t_{old} and t_{new}), we define:

$$\text{deviation}(t_{\text{old}}, t_{\text{new}}) = \frac{t_{\text{new}} - t_{\text{old}}}{\max\{t_{\text{new}},\ t_{\text{old}}\}} \tag{1}$$

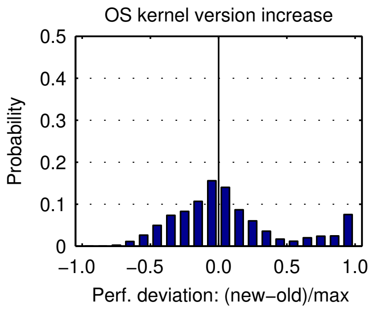

Figure 3: Change profile (in histogram) for OS version increase from Linux 2.6.3, 2.6.10, or 2.6.19 to 2.6.23.

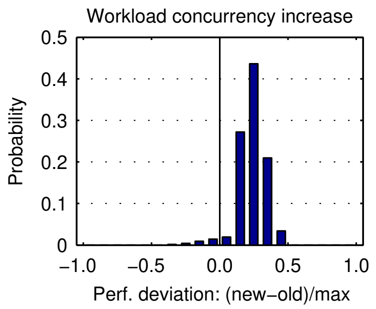

Figure 4: Change profile (in histogram) for workload concurrency increase by a factor of four.

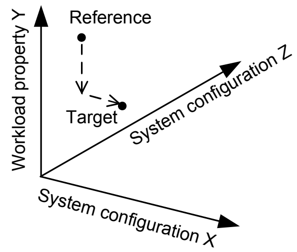

Figure 5: In a multi-parameter execution condition space, the condition change from reference to target is a combination of a single-parameter change on Y and a single-parameter change on X.

This performance deviation metric has a bounded range in $[-1.0, 1.0]$. Given $\text{deviation}(t_1, t_2) = -\text{deviation}(t_2, t_1)$, it also exhibits symmetry at opposite change directions. This allows us to simply revolve the distribution around the origin point when the change direction is reversed.

As an example, we can derive the change profile for OS kernel version increase by sampling the resulted performance deviations under many randomly chosen execution conditions (including system configuration parameters and workload properties, described in Section 4.1). Figure 3 illustrates the results for migrating from Linux 2.6.3, 2.6.10, or 2.6.19, to Linux 2.6.23. The results validate the intuitive expectation that large performance degradation is uncommon after system version increase but large performance improvement is quite possible.

Similarly, Figure 4 illustrates the change profile for the workload concurrency increase by a factor of four (the concurrency is limited so memory thrashing [7] does not occur). Results suggest that a workload concurrency increase tends to improve the I/O throughput slightly. The intuition is that at a higher concurrency, the I/O scheduler can choose from more concurrent requests and therefore achieve better seek reduction.

Given a probabilistic distribution of expected performance deviations with the probability density function of $\pi(\cdot)$. We can then quantify the anomaly of an observed performance degradation (δ) from a reference execution to the target. Specifically, we use the one-tailed p-value in statistics:

$$\text{pval}(\delta) = \int_{-1.0}^{\delta} \pi(x)\, \mathrm{d}x. \qquad (2)$$

It represents the cumulative probability for a random observation (drawn from the distribution with density function $\pi(\cdot)$) to be no more than δ. Intuitively, it is the probability to observe a performance degradation at least as extreme as δ. A lower p-value is more anomalous. The p-value concept makes no assumption on the underlying distribution π, making it easily applicable to our case.

2.2 Scalable Anomaly Quantification

In the previous subsection, we use sampling to construct change profiles for single-parameter adjustments (system version in Figure 3 or workload concurrency in Figure 4). They are able to help quantify the anomaly (in p-value) between a target and its reference execution that differ only in one condition parameter. A complex system, however, may have many execution condition parameters, including multiple configurable parameters in the system software and various properties for the hosted workloads. One may attempt to directly construct change profiles for multi-parameter reference-to-target condition changes (*e.g.*, differing in workload concurrency as well as the operating system I/O scheduler). However, there would be too many multi-parameter combinations for a comprehensive construction.

We consider each execution condition as a single point in the multi-dimensional space where each system configuration parameter and workload property represents a dimension. Figure 5 provides an illustration of the *execution condition space*. The multi-parameter change between two execution conditions can be considered as a series of single-parameter changes. To achieve scalability, we directly construct only single-parameter change profiles and then analytically synthesize them to identify anomalous performance deviations over multi-parameter execution condition changes. The intuition is that the performance ratio after a multi-parameter change is the product of the ratios of multiple single-parameter changes. The challenge, however, is the non-deterministic (or probabilistic) nature of our change profiles. Below we first describe a convolution-like synthesis that assumes independent performance deviations due to different parameter changes. We then present a generally applicable approach to conservatively bound the quantified anomaly for low false positives.

Our change profile representation was defined on the performance deviation metric (described in Equation 1) because it provides good illustrating effects (bounded range and symmetry at opposite change directions). However, for the simplicity of mathematical formulations in this section, we introduce a new change profile representation on the metric of performance ratio. The ratio is defined as $\frac{t_{\text{new}}}{t_{\text{old}}}$ where t_{old} and t_{new} are the system throughput measures before and after the execution condition change respectively. Specifically, the *performance ratio change profile* for an execution condition change is the probabilistic distribution of resulted performance change ratios. Note that the two change profile representations can be easily converted to each other. So our analytical results on performance ratio change profiles can be easily applied to the original change profile representation.

We introduce some notations. Let the target and reference executions differ on multiple condition parameters: $p_1, p_2, \cdots, p_n$. The single-parameter performance ratio change profile on p_i has a known probability density function of $\pi_{p_i}(\cdot)$. Let $\text{pval}_{p_i}(\cdot)$ be its one-tailed p-value function, or $\text{pval}_{p_i}(\delta) = \int_0^{\delta} \pi_{p_i}(x)\, \mathrm{d}x$.

Convolutional Synthesis of Independent Parameters. By assuming the independence across multiple parameter changes, we can probabilistically assemble the multi-parameter change profile from multiple single-parameter profiles. In particular, below we derive a two-parameter (p_1 and p_2) change profile using a variant of the convolution operator:

$$\pi_{p_1 p_2}(y) = \int_0^{\infty} \pi_{p_1}(x) \cdot \pi_{p_2}(\frac{y}{x})\,\mathrm{d}x. \quad (3)$$

Change profiles adjusting more than two parameters can be derived by repeatedly applying the above operator.

With the multi-parameter change profile, we can easily quantify the p-value anomaly of any observed performance degradation.

Bounding Analysis with General Applicability. Unfortunately, the assumption on independent performance deviations across multiple parameter changes may not be true in practice. As one example, different application I/O access patterns may lead to varying performance effects of aggressive I/O prefetching.

Here we introduce a more generally applicable problem model. Given the performance degradation (in the ratio δ) from a reference execution to the target, our p-value anomaly is the probability to observe a performance degradation at least as extreme as δ. We consider a randomly observed multi-parameter performance ratio change as an aggregate of multiple single-parameter changes—x_1, x_2, $\cdots$, x_n (where x_i is drawn from the density function $\pi_{p_i}(\cdot)$). We have:

$$\text{pval}_{p_1 \cdots p_n}(\delta) = \text{Prob}\left[\prod_{i=1}^{n} x_i \leq \delta\right]. \quad (4)$$

Without any knowledge of independence or correlation across different single-parameter performance changes (or x_i's), it is generally impossible to quantify the exact p-value anomaly in Equation 4. Here we derive a p-value upperbound which serves as a conservative anomaly estimation. We begin by introducing the following lemma.

LEMMA 1. *For any set of single-parameter performance ratio changes $\hat{x_1}$, $\hat{x_2}$, $\cdots$, $\hat{x_n}$ (all greater than 0) that exactly aggregate to the overall change of δ (in other words, $\prod_{i=1}^{n} \hat{x_i} = \delta$), we can show that $pval_{p_1 \cdots p_n}(\delta) \leq \sum_{i=1}^{n} pval_{p_i}(\hat{x_i})$.*

Proof: For any set of x_1, x_2, $\cdots$, x_n where $x_i > \hat{x_i}$ for all i's, we know that $\prod_{i=1}^{n} x_i > \prod_{i=1}^{n} \hat{x_i} = \delta$. Conversely, if $\prod_{i=1}^{n} x_i \leq \delta$, then $x_i \leq \hat{x_i}$ holds for at least one i. Therefore:

$$\begin{aligned} \text{pval}_{p_1 \cdots p_n}(\delta) &= \text{Prob}\left[\prod_{i=1}^{n} x_i \leq \delta\right] \\ &\leq \text{Prob}\left[(x_1 \leq \hat{x_1}) \text{ or } \cdots \text{ or } (x_n \leq \hat{x_n})\right] \\ &\leq \sum_{i=1}^{n} \text{Prob}\left[x_i \leq \hat{x_i}\right] \\ &= \sum_{i=1}^{n} \text{pval}_{p_i}(\hat{x_i}). \quad \blacksquare \end{aligned} \quad (5)$$

Lemma 1 can derive a p-value upperbound anomaly for any set of single-parameter performance ratio changes that exactly aggregate to the overall change of δ. Since a tighter bound is more desirable, we want to identify the one yielding the smallest upperbound. Formally, our problem is to:

$$\text{Minimize} \sum_{i=1}^{n} \text{pval}_{p_i}(\hat{x_i}), \quad \text{subject to} \quad \prod_{i=1}^{n} \hat{x_i} = \delta. \quad (6)$$

The optimal solution to the above problem can be derived using Lagrange multipliers as follows. As there is just a single constraint, we use only one multiplier λ to combine the constraint and the optimization goal together into the Lagrangian function:

$$\Lambda(\hat{x_1}, \hat{x_2}, \cdots, \hat{x_n}, \lambda) = \sum_{i=1}^{n} \text{pval}_{p_i}(\hat{x_i}) + \lambda \cdot (\prod_{i=1}^{n} \hat{x_i} - \delta) \quad (7)$$

The critical values of Λ are achieved only when its gradient is zero. In other words, for each k ($1 \leq k \leq n$), we have:

$$\frac{\partial \Lambda}{\partial \hat{x_k}} = \frac{\partial \text{pval}_{p_k}(\hat{x_k})}{\partial \hat{x_k}} + \lambda \cdot \prod_{i=1 \to n,\, i \neq k} \hat{x_i} = \pi_{p_k}(\hat{x_k}) + \lambda \cdot \frac{\delta}{\hat{x_k}} = 0. \quad (8)$$

Therefore, we know that the optimal solution must satisfy:

$$\hat{x_1} \cdot \pi_{p_1}(\hat{x_1}) = \hat{x_2} \cdot \pi_{p_2}(\hat{x_2}) = \cdots = \hat{x_n} \cdot \pi_{p_n}(\hat{x_n}). \quad (9)$$

Combining Condition 9 with the constraint $\prod_{i=1}^{n} \hat{x_i} = \delta$, we can compute a numerical solution using the iterative secant method (a variant of the Newton's method).

In practice, the calculation of the above optimal solution requires accurate distribution density values $\pi_{p_i}(\cdot)$'s, which places a high burden on the construction of single-parameter change profiles. As a simpler alternative, one can use a Monte Carlo approximation—to sample a large number of randomly chosen sets of $\hat{x_i}$'s (where $\prod_{i=1}^{n} \hat{x_i} = \delta$) and pick the one producing the minimal $\sum_{i=1}^{n} \text{pval}_{p_i}(\hat{x_i})$.

By providing a conservative anomaly estimation without requiring the independent parameter assumption, our bounding analysis yields high confidence in identified anomaly symptoms (low false positives). However, it may allow some anomalous reference-target pairs to escape identification. This is acceptable if our goal is to identify some but not all anomaly symptoms in the system.

2.3 Additional Discussions

We may need to construct multiple single-parameter change profiles on one parameter due to multiple possible change-from and change-to parameter settings. For categorical parameters with more than two settings (like the Linux I/O scheduler with `noop`, `deadline`, and `anticipatory` settings), we would need to construct multiple change profiles (one for each two-setting pair). For quantitative parameters such as the workload concurrency, we may need to construct change profiles for different concurrency adjustment magnitude. Alternatively, to save the overhead of direct construction, we may only directly construct change profiles for small-magnitude setting changes. We then consider a large-magnitude setting change as an aggregate of multiple small-magnitude changes and use our multi-step synthesis in Section 2.2 to derive anomaly quantification. Note, again, that our generally applicable bounding analysis makes no assumption on how the overall performance deviation is distributed across multiple small-magnitude parameter changes.

The p-value calculation provides a way to quantitatively rank suspected anomaly symptoms. However, there is a lack of well-founded threshold below which a p-value would indicate a true anomaly. In practice, 0.05 is a commonly used threshold to signify statistical significance, which can be traced to Fisher's early work on agriculture experiments [3].

3. UTILIZATIONS

Our approach identifies anomalous reference-to-target performance degradations in a complex system with a large execution condition space. This is an otherwise challenging task due to the difficulty in constructing comprehensive, accurate performance models for complex systems [11, 15, 17, 18]. Our anomaly symptom identification can be utilized in a number of ways. We can measure the system performance at some sample execution conditions (with good coverage of the whole space) and use them to serve as references to each other for identifying anomalous execution conditions. Anomalous symptoms can further serve as the basis for the root cause analysis. During online operations, the performance anomaly detection on the current execution condition can also enable more dependable system management.

3.1 Anomaly Cause Analysis

Given a pair of reference-target executions with anomalous performance degradation, we want to discover the anomaly cause in the system implementation. Corrections to anomaly causes would improve the system performance, and more importantly, they would maintain predictable performance behavior patterns for the system. The root cause discovery also helps validate that the anomaly symptom identifications are correct—*i.e.*, they indeed correspond to implementation deviations from the high-level design.

The anomaly cause analysis is challenging due to the implementation complexity in large systems. Fortunately, there is an abundance of collectible metrics in today's systems and low-level system metrics may shed lights on internal system behaviors. Specifically for the case of the Linux I/O subsystem, example metrics include the I/O request granularity and workload concurrency at different levels of the system, the frequency of various system events (like system calls), as well as the latency of many concerned system functions. It is likely that some collected metrics are related to the anomaly cause. Such anomaly-related metrics, if discovered, may help significantly narrow the scope of root cause analysis.

A scalable anomaly cause analysis needs to prune the vast majority of anomaly-unrelated metrics so that the manual examination is only required in a very limited scope. Our approach is driven by the intuition that system metrics that do not differ significantly between the target anomalous execution and its normal reference are not likely related to the performance anomaly. Conversely, those metrics with very different anomaly-reference manifestations may be anomaly-related. This would be particularly so if the target and reference execution conditions are very alike (or if a very recent past version serves as the reference to a target system version).

Typically, a performance-oriented system metric manifests as a set of varying quantitative sample measurements in a system execution. In general, we represent the manifestation of each system metric as a probability distribution of measured sample values. We then quantify how each metric's manifestation differs between the target anomalous execution and its normal reference. As an example, Figure 6 illustrates the probabilistic manifestations of one metric in target and reference executions.

Motivated by Joukov *et al.*'s differencing of latency distributions [6], we utilize the earth mover's distance [14] as the difference measure between two distributions. Consider the process of moving some probability density mass of one distribution's probability density plot to become another distribution's probability density plot, the earth mover's distance indicates the minimum amount of work (probability density mass times the moving distance) required for such move. Note that different metrics may have different units of measure or scales. To allow their difference measures to be directly comparable, we scale all sample values for each metric so that the larger of the two distribution means equals 1.0.

The result of our approach is a list of system metrics ranked on their manifestation differences between the target and reference executions. Although a large difference indicates a high likelihood that the metric is anomaly-related, the difference may also be due to the natural variation between reference and target executions. Further, it may not be trivial to pinpoint the exact anomaly cause

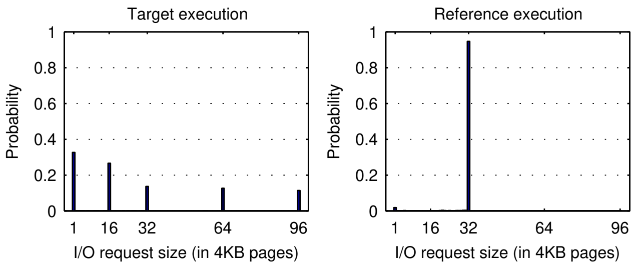

Figure 6: A real example of metric manifestations for target and reference executions. The concerned metric is the granularity of read I/O requests at the device level (sent from file system to the SCSI device driver). Probabilistic distributions are computed from over 10,000 request size measurements in each execution.

from an anomaly-related system metric. In some cases, a metric-to-anomaly correlation does not necessarily mean causality or dependence (*i.e.*, the differing metric manifestation may not directly cause the performance anomaly). We acknowledge that the final metric screening and anomaly-cause reasoning still require significant human efforts. However, our reference-driven metric filtering can significantly narrow the scope of such manual work.

Our method can discover anomaly-related metrics on execution states (*e.g.*, the number of outstanding block-level I/O requests) as well as control flows (*e.g.*, the frequency of reaching a particular I/O exception condition). More precise control flow tracking for anomaly analysis can be done at function or basic block levels, such as in Triage [20] and DARC [19]. Note that our analysis target (execution of the full software system) is significantly broader than that of Triage (execution of an application program) and DARC (latency peak of a particular system operation). A good overall approach may be to first narrow down the analysis target using our reference-driven metric filtering, and then apply more precise control flow tracking to further reduce the human analysis efforts.

3.2 Online Anomaly Detection and System Management

It is hard to detect anomaly symptoms in complex systems. The dynamic and wide ranging workloads of such systems can mask small performance degradations. Further, such systems have many complicated system parameters with hard-to-understand performance effects. Our reference-driven anomaly identification is useful in this scenario, because it uses easily-built change profiles (rather than expert knowledge or detailed models on system internals) to capture design-intended performance. Further, our approach can be applied online to detect anomalies as they happen.

Our reference-driven approach to online anomaly detection begins with the offline construction of single-parameter change profiles. Then during online execution, we monitor performance, workload conditions, and system configurations. We periodically compile monitoring data for the most recent executed condition. This condition serves as the target in our anomaly identification. For references, we use conditions encountered in previous online executions or during the change profile construction.

Online anomaly detection has practical uses in system management. It can alert system administrators of costly performance degradations. With online anomaly detection, management software may automatically increase logging activity to support later analysis of anomaly root causes. And in autonomic systems, the management software may avoid anomalous execution conditions by reconfiguring the system to normal alternatives.

4. CASE STUDY ON LINUX I/O SYSTEM

In this section, we explore performance anomalies in the Linux I/O subsystem. We consider an execution condition space that spans eight system parameters and four Linux versions. This case study demonstrates our reference-driven anomaly symptom identification and root cause analysis on a real system implementation.

4.1 Empirical Setup

We empirically examined performance anomalies over a multi-dimensional execution condition space, including the following workload properties related to I/O system performance:

1. *Number of concurrent I/O operations*: 1, 2, 4, 8, 16, 32, 64, 128, or 256.
2. *Average length of sequential access streams*: 64 KB, 128 KB, 256 KB, 512 KB, 1 MB, or 2 MB.
3. *Portions of sequential streams accessed without interleaving I/O from other streams*: 16 KB, 32 KB, 64 KB, $\cdots$, up to the length of sequential access streams. We also call this parameter *sequential access run length*.
4. *Whether each stream start at file beginning*: true or false.
5. *Average application thinktime per megabyte of data access*: 0 ms, 1 ms, 2 ms, 4 ms, or 8 ms.

To create workloads with arbitrary properties, we developed an adjustable microbenchmark that can exhibit any combination of workload property settings. It reads randomly selected files from a dataset of 5,000 4 MBytes files (19.5 GB in total).

Further in the execution condition space, we considered three system configuration parameters related to I/O system performance:

6. *File system caching*: enabled or disabled.
7. *File system prefetching depth*: 64 KB, 128 KB, 256 KB, or 512 KB.
8. *Linux I/O scheduling*: noop, deadline, or anticipatory.

We augmented the operating system to allow these different configurations. The prefetching depth and I/O scheduler can be adjusted by setting appropriate operating system configuration variables. To disable the file system caching, one can discard the cached pages encountered during I/O processing.

We experimented with four Linux versions released in the span of about four years (2.6.23, October 2007; 2.6.19, November 2006; 2.6.10, December 2004; and 2.6.3, February 2004). Our measurements used a server equipped with dual 2.0 GHz Xeon processors and 2 GB memory. The data is hosted on an IBM 10 KRPM SCSI drive with raw seek time in the range of 1.3–9.5 milliseconds (depending on the seek distance) and raw sequential transfer rate in the range of 33.8–66.0 MBytes/second (depending on the disk zone where the data is located).

4.2 Anomaly Symptom Identification

As the basis for our anomaly symptom identification, we created single-parameter change profiles for all system parameters. Each probabilistically characterizes the expected performance deviations when a single execution condition parameter changes (while all other parameter settings remain unaltered). We produced the probabilistic distribution by sampling the resulted performance deviations under at least 288 randomly chosen settings of other parameters. At each execution condition, we measured the system throughput by averaging the results of three 100-second runs. Our measurements are stable. Excluding the very low-throughput (2 MBytes/second or less) execution conditions, the standard deviation of every condition's three-run throughputs is less than 10% of the corresponding average.

Figure 7 shows some produced single-parameter change profiles on execution condition adjustments. For quantitative parameters like prefetching depth, workload concurrency, and the lengths of sequential access stream/run, the provided change profiles are for parameter value increases by a factor of four. Following our discussion at Section 2.3, our anomaly symptom identification approach considers larger-magnitude setting changes on these parameters as an aggregate of multiple small-magnitude changes. Figure 8 shows change profiles for the Linux kernel version evolution.

A key advantage of our approach is that the probabilistically derived change profiles (in Figures 7 and 8) match high-level system design principles and often they can be intuitively interpreted. We provided such understanding for two single-parameter change profiles in Section 2.1. Here we briefly explain several others. Caching in memory improves the system performance, while the specific improvement depends largely on the data-to-memory ratio (19.5 GB data on 2 GB memory for our case). The anticipatory scheduler may improve the concurrent I/O performance in cases of deceptive idleness [5]. With respect to sequential I/O accesses, it is intuitive that workloads with more sequential patterns tend to deliver higher I/O throughput. More aggressive prefetching would take advantage of sequential accesses to deliver better performance.

We identified symptoms of execution condition change anomalies and system version evolution anomalies separately. To explore execution condition change anomalies, we chose a number of sample conditions from the execution condition space. We measured I/O throughput at all chosen conditions and used them as references to each other for identifying anomalous execution conditions. For reference-to-target execution condition changes that are not directly characterized by existing single-parameter change profiles, we used our generally applicable bounding analysis in Section 2.2 to produce anomaly measures. To explore system evolution anomalies, we measured I/O throughput of multiple system versions over the same set of sample execution conditions. We then used an earlier system version as the reference to identify performance anomaly symptoms in the more recent system version.

We explored execution condition change anomalies on both Linux 2.6.23 and Linux 2.6.10. As multiple nearby sampled execution conditions may serve as references for a target condition, we chose the one yielding the smallest p-value (or the highest anomaly likelihood) to be its reference. The corresponding p-value is the anomaly measure for the target condition. We also examined evolution anomalies between the two system versions. As inputs to our studies, we chose 300 sample conditions from the execution condition space in a uniformly random fashion and measured the system throughput at these 300 conditions for both kernel versions.

Validation. We validated whether the identified anomaly symptoms (reference-target pairs with anomalous performance degradation) indeed correspond to implementation deviations from the high-level design. As discussed in Section 2.3, we focused on the symptoms with p-value measure of 0.05 or less. This indicates a 5%-or-less probability for observing performance degradations (in the real system) at least as extreme as these symptoms. Ideally, for each symptom, we need to find the real cause for the performance difference first and then judge whether it corresponds to a departure from the system design intention. In particular, our validation utilized the discovered anomaly causes presented later in Section 4.3. If an anomaly symptom can be explained by a valid anomaly cause, we consider it a true anomaly. On the other hand, a symptom that cannot be explained by any known anomaly cause is not necessarily a false positive—it may be due to a not-yet-discovered anomaly cause. As an indication of high confidence, all our suspected false

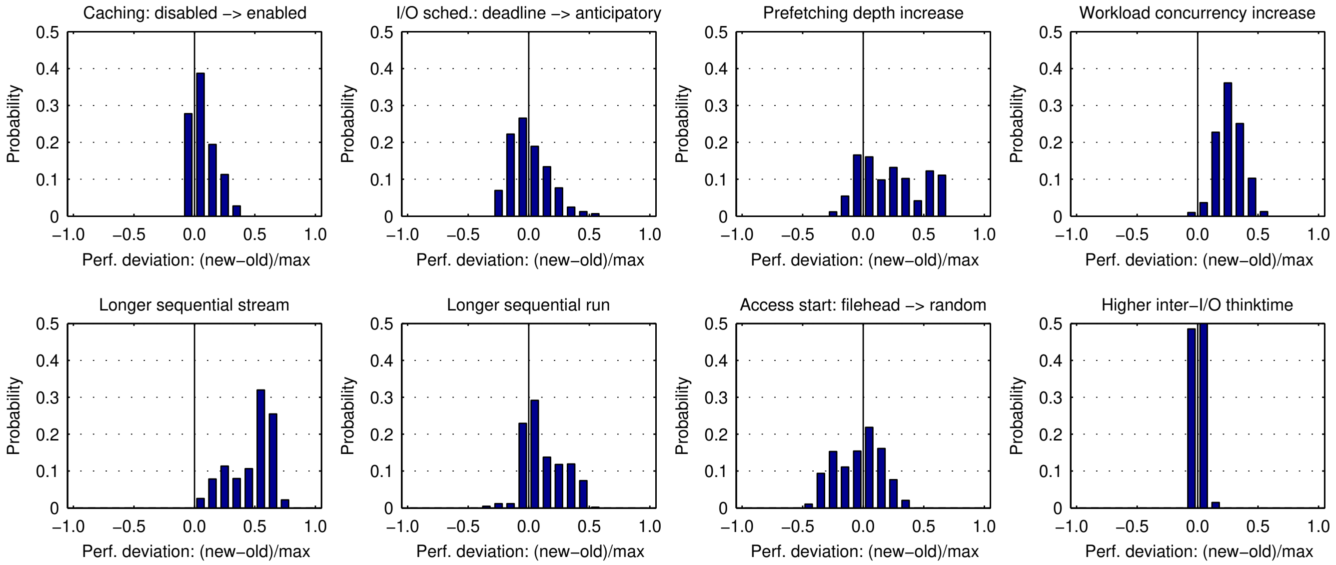

Figure 7: Single-parameter change profiles (in histograms) for adjustments of various execution condition parameters. The performance (I/O throughput) deviation is defined earlier in Equation 1.

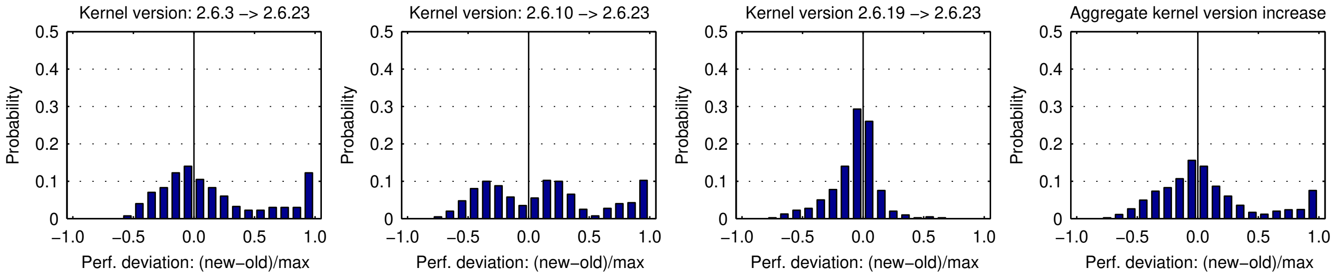

Figure 8: Single-parameter change profiles (in histograms) for OS kernel version increases.

positives can be explained by some normal system design intentions. We call them *probable false positives*.

We provide the validation results to demonstrate the low false positive rate of our approach. Below are results for the three performance anomaly explorations. The cited anomaly cause numbers point to detailed descriptions in Section 4.3:

- We identified 35 top anomaly symptoms on Linux 2.6.10 execution condition changes. Within these, 32 are due to cause #1, one is due to cause #3, and one is due to cause #6. The remaining one is a probable false positive—the reference-to-target performance degradation can be explained by the anticipatory scheduler (of the reference execution)'s expected high throughput for concurrent I/O.
- The exploration on Linux 2.6.23 execution condition changes identified 12 top anomaly symptoms. Within these, four are due to cause #2, one is due to #4, one is due to #5, and three are due to a combination of causes #6 and #7. The remaining three are probable false positives.
- We identified 15 top symptoms on Linux 2.6.10 to 2.6.23 evolution anomalies. Within these, 14 are due to cause #5 and one is due to cause #4. There is no false positive.

Comparison. For complex systems with multi-parameter execution conditions, our bounding analysis in Section 2.2 provides a conservative estimation of the anomaly measure which leads to low false positives in identified anomaly symptoms. We compared this approach to the convolutional synthesis that assumes independent parameters (also presented in Section 2.2). We further compared against another approach, called *raw-difference*, that quantifies anomalous reference-to-target performance degradations based on the raw degradation ratio. To minimize false positives in raw-difference, we only consider references that are very similar to the target (differing on no more than two execution condition parameters with small-magnitude differences on quantitative parameters).

Figure 9 shows the anomaly symptom identification results in the Linux 2.6.23 execution condition exploration. Results suggest that the convolutional synthesis approach may identify more anomalies than the bounding analysis, but it does so at the cost of a much higher false positive ratio. This is the result of potential overestimation of the anomaly measure (or under-calculation of the p-value) due to the convolutional synthesis's independent parameter assumption. Figure 10 shows its quantified p-values in relation to those under the more generally applicable bounding analysis.

Finally, Figure 9 also shows that the raw-difference approach is poor in identifying performance anomalies. Most of the significant reference-to-target performance degradations can be explained by normal system design intentions.

4.3 Anomaly Cause Discovery

We attempted to discover the causes of the identified anomaly symptoms using our reference-driven system metric filtering (de-

Category	*Event types*
Process management	Kernel thread creation, process fork or clone, process exit, process wait, process signal, process wakeup, CPU context switch
System call	System call enter, system call exit (distinct event type for each system call number)
Memory system	Page allocation, page free, page swapin, page swapout
File system	File execution, file open, file close, file read, file write, file seek, file ioctl, file prefetch, start waiting for a data buffer, finish waiting for a data buffer
I/O scheduling	Request arrival, request re-queue, request dispatch, request removal, request completion
Anticipatory scheduling (when enabled)	Request arrival, request dispatch, request completion, anticipation timeout, request deadline triggering immediate service, anticipation stop, various reasons to stop anticipation
SCSI device	SCSI device read request, SCSI device write request
Interrupt	Enter interrupt handler, exit interrupt handler (distinct event type for each interrupt identifier)
Network socket	Socket call, socket send, socket receive, socket creation

Table 1: Traced Linux event types. The total number of event types is up to 624 for Linux 2.6.10 and up to 703 for Linux 2.6.23. The difference in event type number is mainly due to additional system calls in newer kernels.

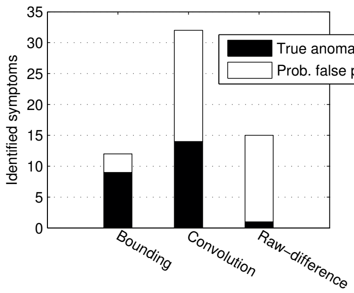

Figure 9: Numbers of identified true anomalies and probable false positives (in Linux 2.6.23) out of the total 300 samples. For raw-difference, the identified anomalies are those with the highest 5% reference-to-target performance degradation ratio. This threshold is comparable to the 0.05 p-value threshold used in the other two approaches.

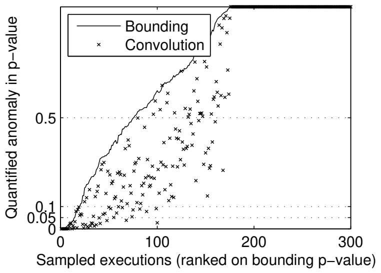

Figure 10: Quantified p-values under different approaches for 300 sampled execution conditions (in Linux 2.6.23).

scribed in Section 3.1). This evaluation demonstrates the effectiveness of our proposed approach. The discovered causes also help validate the identified anomaly symptoms by matching them with real causes in the implementation, as explained in Section 4.2.

Traced Events and Derived System Metrics. We first describe traced events and derived system metrics for our reference-driven metric filtering. Table 1 lists the specific Linux event types we traced by instrumenting the operating system. We chose them for their easy traceability and their perceived relevance to the I/O performance. From these traced events, we derived the following system metrics for performance anomaly analysis:

- *Delay of adjacent arrivals for each type of events.*
- *Delay between causal events of different types.* Examples include the delay between a system call enter and exit, the delay between file system buffer wait start and end, the delay between a block-level I/O request arrival and its dispatch, and the delay between a request dispatch and its completion.
- *Event parameters.* Examples include the file prefetch size, SCSI I/O request size, and file offset of each I/O operation to block device.
- *I/O concurrency* (number of outstanding requests). Examples include the I/O concurrency at the system call level, the block level, and the SCSI device level.

The total number of derived system metrics is up to 1,210 for Linux 2.6.10 and up to 1,361 for Linux 2.6.23. Note that some metrics may not manifest or manifest very rarely in specific execution conditions.

Discovered Anomaly Causes. For each pair of reference and target executions with anomalous performance degradation, we ranked the reference-target manifestation differences of all derived system metrics. As a larger difference infers a higher likelihood of anomaly correlation, such ranking helps us to narrow the scope of root cause analysis significantly. Note that even with the hints of anomaly-related system metrics, the final root cause analysis still involves non-trivial human effort. In our evaluation, such human analysis took about one or two days for each anomaly cause discovery. Our explorations on Linux 2.6.10 and 2.6.23 discovered seven anomaly causes (two of which matched known anomalies in our earlier work [15]). Below we describe the discovered causes, preceded by the metric filtering results that helped our discovery.

Anomaly cause #1 (known [15]), afflicting 2.6.10: The ranked metric differences between reference and target executions are shown in Figure 11(A). The two most differing metrics are the decreases of prefetching operation frequency and of the device-level I/O request size. These hints led us to the following anomaly cause. Linux 2.6.10 marks the disk as congested when there are a large number (113 and above) of requests in the device queue. During disk congestion, the OS cancels all prefetching operations and reverts to mostly single-page granularity I/O accesses. This strategy is probably due to the intuition that at high load, prefetching is not likely to complete on time for application access and thus it is not useful. However, this may also lead to extremely poor I/O throughput due to frequent disk seeking.

Anomaly cause #2 (new), afflicting 2.6.23: The fourth and fifth highest ranked metrics in Figure 11(B), prefetching operation frequency and sizes, helped us to make the following

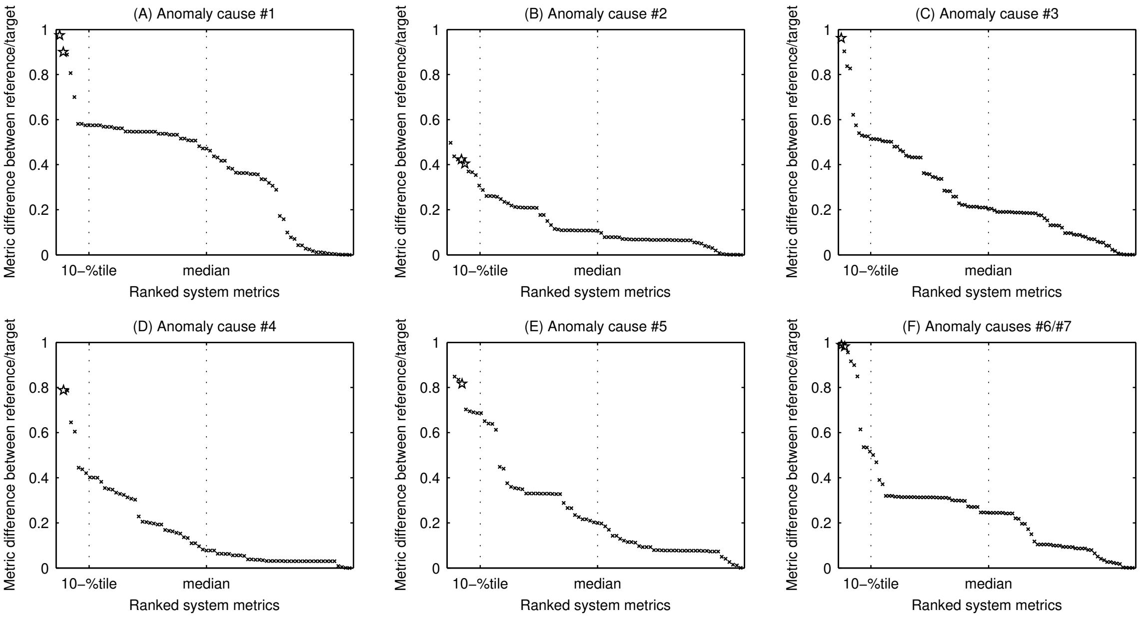

Figure 11: Metric difference (in the earth mover's distance [14]) between the target anomalous execution and its normal reference for anomaly cause discovery. Each marker represents a metric (metrics with too few samples in the executions are not shown). We single out the most helpful system metric(s) for each anomaly cause discovery and mark them as five-point star(s) in the plot.

discovery. Anomaly cause #1 appears to have been corrected since Linux 2.6.11, by only canceling asynchronous prefetching operations at disk congestions (*i.e.*, synchronous prefetching is allowed to proceed). However, the somewhat arbitrary disk congestion threshold at 113 queued I/O requests still remains and it causes anomalous performance deviations when such a threshold is crossed. Specifically, sometimes the throughput is dramatically improved at slightly higher workload intensity due to the cancellation of asynchronous prefetching.

Anomaly cause #3 (known [15]), afflicting 2.6.10/2.6.23: The most differing metric in Figure 11(C) is the frequency of anticipation stops due to the arrival of a new I/O request with a shorter estimated disk seek cost. This directly led to the anomaly cause that Linux inaccurately estimates that the seek cost is proportional to the seek distance (which is inaccurate due to disk head acceleration and settle-down cost).

Anomaly cause #4 (new), afflicting 2.6.23: The second highest ranked metric in Figure 11(D), the I/O operation file offset, helped us to discover the following anomaly cause. The OS employs a slow-start phase in file prefetching—prefetching takes place with a relatively small initial depth, and then the depths increase for later prefetching upon detection of a sequential access pattern. Compared to the earlier version of Linux 2.6.10, Linux 2.6.23 employs a more conservative slow-start—smaller initial depth and slower increases. This may cause substantial increases in disk seek frequency under concurrent I/O workloads. The intention of such change was likely to reduce wasted prefetching on unneeded data. However, it might be a wrong tradeoff given that the existing I/O prefetching depth is already far below a balanced level [8].

Anomaly cause #5 (new), afflicting 2.6.23: The fourth highest ranked metric in Figure 11(E), the device-level I/O request size, was helpful. Its manifestation distributions in target/reference executions were shown as an example in Section 3.1 (Figure 6). When a file access does not start from the file beginning, Linux 2.6.23 considers it a random access and does no prefetching at all, which typically results in inefficient small-granularity I/O. As in the case of cause #4, the intention would likely be to reduce wasted prefetching on unneeded data, but the potential cost of frequent disk seeking may outweigh the reduced waste on I/O bandwidth.

Anomaly cause #6 (new), afflicting 2.6.10/2.6.23: The most differing metric in Figure 11(F), the frequency of anticipation stops due to timeouts, led us to the following anomaly cause. The anticipatory scheduler only allows outstanding read operations from one process at a time. This restriction, however, does not prevent multiple outstanding device-level I/O requests—in the case of splitting a large file system-level operation into multiple device-level requests to satisfy the size limit, and in the case of concurrent asynchronous prefetching and synchronous I/O from a single process. In such cases, the anticipation timer is started after the first device-level request returns, causing premature timeout and issuance of other processes' requests (often before all outstanding requests from the current process return).

Anomaly cause #7 (new), afflicting 2.6.23: Correction to the anomaly cause #6 does not completely compensate the anomalous performance degradation between the reference and target executions. A closer look at the two highest ranked metrics, anticipation stops due to timeouts and excessive inter-I/O thinktime (in relation to timeout), led us to the following additional anomaly cause. Considering the following code in anticipatory I/O timeout setup:

```
/* max time we may wait to anticipate a read
   (default around 6ms) */
#define default_antic_expire ((HZ/150)?HZ/150:1)
```

With 1 KHz kernel ticks (`HZ=1000`), this code calculates correct timeout value of six ticks. However, very recent Linux kernels (including 2.6.23) employ 250 Hz kernel ticks on default. Consequently the above code calculates a timeout value of one tick. Effectively, the anticipation timeout can

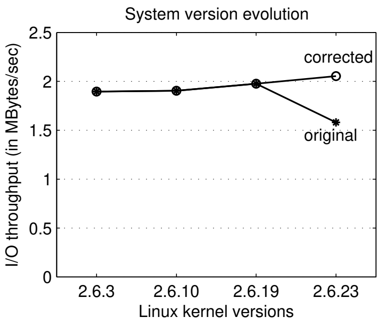

Figure 12: Effect of kernel correction (for anomaly cause #4) on I/O-bounded SPECweb99. All workload setup is the same as the experiment described in Figure 1.

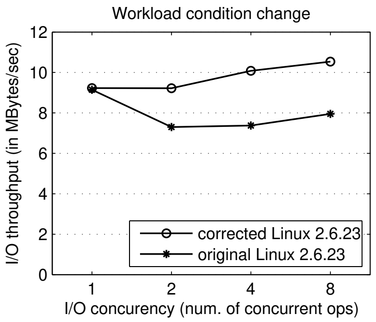

Figure 13: Effect of kernel corrections (for anomaly causes #6/#7) on an I/O microbenchmark. All workload setup is the same as the experiment described in Figure 2.

occur anywhere from 0 to 4 ms and we observe timeouts as short as 100 μs in practice. After our discovery and report, this performance anomaly has been acknowledged by the Linux kernel development team [10].

The discovered anomaly causes are of different natures. We believe only #1, #6, and #7 are unqualified bugs that should be corrected without question. Causes #4 and #5 are better described as partially beneficial optimizations that may degrade performance in some workload conditions. Corrections to these anomaly causes should be made in special-purpose systems (*e.g.*, machines dedicated to run a particular web server workload) that are most susceptible to the anomalous conditions. Finally, #2 and #3 are most likely intentional simplifications for which perfections (*i.e.*, optimal threshold setting and accurate disk seek time estimation) are difficult to realize.

Effects of Corrections. To demonstrate the effects of anomaly corrections, we show the I/O performance on corrected Linux 2.6.23 kernel for the two performance anomaly examples provided in Section 2.1. Results in Figure 12 indicate that our anomaly correction improves the SPECweb99 throughput by 30% on Linux 2.6.23. Figure 13 demonstrates that our corrections improve the system I/O throughput by 26–37% during concurrent executions. More importantly, our corrections lead to predictable performance behavior patterns during system version evolution and execution condition changes.

5. CASE STUDY ON A DISTRIBUTED ONLINE SERVICE

This section presents a preliminary case study on anomaly detection and system management for a J2EE-based distributed online service. First, we show that our change profiles capture intuitive performance expectations across system parameters. Built on our reference-driven anomaly identification, we further show that system reconfiguration can be a promising technique to evade anomalous performance degradations (leading to performance improvements of up to 69%).

5.1 Empirical Setup

We studied performance anomalies for the RUBiS online auction benchmark [13]. RUBiS is a multi-tier Internet service comprising a web server, a database, and nine middle-tier Enterprise Java Bean (EJB) components. The database manages persistent information about users, items, and ongoing auctions. EJB components query the database in order to compute business logic, such as the winning bids or items for sale in various geographic regions. The web server converts results from the middle-tier into presentable HTML format for end users. RUBiS ran on top of the JBoss Application Server, a middleware platform that provides procedures for caching remote data, communicating across cluster nodes, and managing cluster resources. In our setup, the RUBiS application components were distributed across three cluster nodes: our Tomcat web server ran on a front-end server, our MySQL database ran on a separate back-end server, and the RUBiS EJB components were distributed across all three for high performance [17].

Reference and target executions for this study are similar in terms of their JBoss configuration and supported RUBiS workload. An anomalous execution pair is one in which the request throughput, *i.e.*, the number of user requests successfully completed, is unexpectedly lower in the target execution compared to the reference. We explored 5 JBoss configurations and 4 properties of the RUBiS workload, which together represent over one million potential execution conditions. The considered JBoss configurations are (* indicates default settings):

1. *EJB-component cache coherence*: no cache*, assuming exclusive-access, or verifying content before use.
2. *Component invocation protocol*: Java RMI or JBoss-specific*.
3. *Invocation retry policy*: never retry* or retry once.
4. *Database driver*: version 3.1 or version 5.0*.
5. *Maximum concurrency (thread count)*: low (10), medium (128)*, medium high (512), or high (2048).

We also considered several RUBiS workload properties:

6. *HTTP session type*: HTTP 1.0, HTTP 1.1, or SSL.
7. *Database access frequency in the request mix*: 0%, 25%, 50%, 75%, or 100%.
8. *State maintenance method for EJB components*: bean managed state persistence, container managed state persistence, session state maintenance, or stateless Servlets only.
9. *Request arrival rate*: up to 180 requests per second.

We developed a custom workload generator that could toggle between various settings of HTTP session types, request mixes, and request arrival rates. The RUBiS benchmark can be configured to use different state maintenance policies. The dataset for our database was sized according to published dumps at the RUBiS web site [13]. Each node in our cluster was equipped with two 1.266 GHz Intel Xeon processors, 2 GB memory, and connected to 1 Gbps Ethernet.

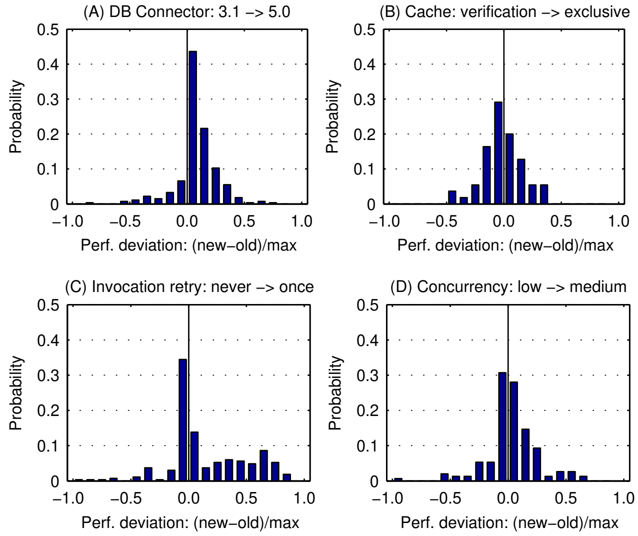

Figure 14: A partial set of single-parameter change profiles (in histograms) for JBoss execution condition adjustments. We perform pairwise tests in which each pair differs on only one execution condition parameter and their performance deviation then contributes to the change profile of the said parameter. Each change profile reflects performance deviations between about 260 pairs of sampled conditions.

5.2 Anomaly Symptom Identification and Online Management

Figure 14 plots four single-parameter change profiles for the JBoss system (a partial set due to space limitation). Our profiles capture intuitive performance variation patterns that reflect design-intended effects of each execution condition change. Specifically, we provide the following intuitive understanding on the single-parameter change profiles of Figure 14:

A) Updates in version 5.0 of the database driver, MySQL ConnectorJ [2], included a performance improving capability called resultset streaming.
B) The cache coherence policy has little impact on performance since EJB cache misses are rare in our RUBiS setup.
C) Invocation retries may yield better performance since they mask transient failures and sometimes salvage user requests that would otherwise be unsuccessful.
D) Low concurrency limits the utilization of available physical resources and therefore the medium concurrency may deliver better performance.

Single-parameter change profiles are the basis for our reference-driven approach to online anomaly detection. We deployed RUBiS under the default JBoss configuration, and executed a realistic sequence of user requests for over 4 hours. The trace that we used is derived from realistic workload patterns of a large e-commerce retailer and a global enterprise application at HP. Requests for each service were mapped to RUBiS requests to mimic their realistic nonstationary workloads [16]. The workload trace is publicly available [1]. During our tests, request arrival rates varied by two orders of magnitude and fluctuated in a sinusoidal fashion. Every request mix setting was encountered.

We followed the general approach described in Section 3.2. The execution was divided into 5-minute intervals. Figure 15 shows the throughput of the execution condition at each interval (ranked on the observed throughput). Our approach identified 7 anomalous execution conditions (p-value of 0.10 or less in this evaluation), including the conditions with 7^{th} and 11^{th} highest throughput. Anomalies with such subtle symptoms, *i.e.*, anomalous targets exhibiting decent absolute performance, are hard to detect. However, Figure 15 also shows the opportunistic cost represented by such anomalies. By reconfiguring the JBoss system parameters of the two anomalies, we were able to achieve respective performance improvements of 25% and 69% under the same workload conditions. This result suggests that system reconfiguration might be a valuable tool in evading performance anomalies.

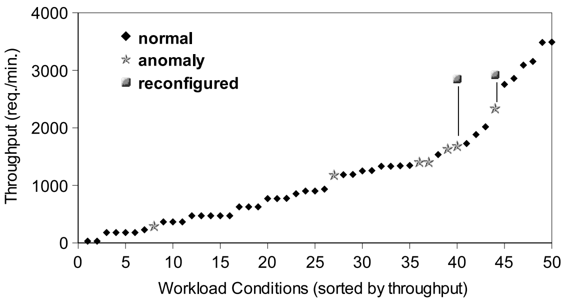

Figure 15: Throughput under each 5-minute interval in our trace. Each point on the x-axis reflects a unique combination of request rate and request mix. Diamonds and five-point stars reflect throughput under default configuration settings. Stars are conditions identified as anomalous. Squares show the throughput under alternative JBoss configuration settings. We found the alternative settings by randomly testing 20 nearby configuration settings.

6. RELATED WORK

Recent research has tackled performance anomaly detection and diagnosis for complex systems. Reynolds *et al.* [12] proposed an infrastructure, called `Pip`, to expose system performance problems by comparing system metric manifestation and programmer-specified expectation. Joukov *et al.* [6] uncovered operating system performance problems by analyzing the latency distribution of system execution. Without systematic understanding of expected performance, these anomaly identification approaches rely on programmer-specified expectations or they target specific system properties with known normal behaviors. By using references, our approach requires little knowledge on the target system design/implementation and it can identify anomalies over wide ranges of system execution conditions.

Besides our reference-driven approach, performance expectations can also be derived through absolute performance models (typically driven by design specifications) [15,17,18]. In particular, IRONModel [18] characterizes performance anomalies as observed deviations from such model expectations. However, it is difficult to construct comprehensive, design-driven performance models for complex systems with varying execution conditions (including system configuration parameters and workload properties). Expectations and corresponding anomalies can also be derived through machine learning techniques [11]. Most of the learning techniques, however, do not produce easily interpretable results. Consequently they cannot directly help discover anomaly causes and derived anomalies are often hard to validate.

The use of correct peer systems to aid problem diagnosis is not new. Wang *et al.* [21] discovered erroneous Windows registry configurations by matching with a set of known correct configurations. Triage [20] and delta debugging [22] proposed to isolate problem sources by comparing against successful program runs with slightly different inputs. These studies focus on diagnosing non-performance problems for which anomaly symptoms (crashes or program failures) are obvious. However, the normal peer executions are not easily identifiable for performance anomaly analysis. Further, performance analysis must handle quantitative system metrics (*e.g.*, the latency of the `read` system call or the number of outstanding I/O requests) that typically manifest as a set of varying sample measurements. This increases the challenge in understanding the difference between the anomalous target and its reference.

Our search of performance anomalies in a multi-parameter system condition space is reminiscent of the software testing problem of designing efficient test cases with good coverage over a complex system. Grindal *et al.*'s survey [4] summarized a number of combinatorial strategies to choose values for individual input parameters and combine them into complete test cases. In the context of software testing, it is assumed that the success or failure at a system condition can be easily determined after testing. However, identifying performance anomaly at a tested condition is fundamentally more challenging. This paper proposes to identify performance anomalies by checking unexpected performance degradations from reference to target conditions. This approach is based on a scalable technique to construct probabilistic characterizations of expected performance deviations over a large execution condition space.

7. CONCLUSION

This paper makes several contributions to reference-driven performance anomaly identification. First, we present a scalable approach to produce probabilistic expectations on performance deviations due to execution condition changes. This approach allows us to identify anomalous performance degradations between reference and target executions in complex systems with wide ranges of execution conditions. Second, we propose a reference-driven approach to filter anomaly-related performance metrics from many varieties of collectible metrics in today's systems. Such filtering can help narrow the scope of anomaly root cause analysis.

We apply our techniques to identify anomaly symptoms and causes in real system software including the Linux I/O subsystem and a J2EE-based distributed online service. In particular, we have discovered five previously unknown performance anomaly causes in the recent Linux 2.6.23 kernel, including one that has been acknowledged by the Linux kernel development team [10]. Corrections to these anomalies can significantly improve the system performance. But more importantly, they lead to predictable performance behavior patterns during system version evolution and execution condition changes. Such predictability [11, 15, 17, 18] is an essential foundation for automatic system management like resource provisioning and capacity planning.

Finally, our work has uncovered interesting characteristics of real performance anomalies. For instance, less than half of our discovered Linux performance anomaly causes are unqualified bugs. The rest are better described either as partially beneficial optimizations that may degrade performance in some workload conditions, or as intentional simplifications for which perfect implementations are difficult to realize. Also, we demonstrated the potential for reference-driven anomaly detection in a realistic distributed online system. In doing so, we discovered that subtle anomalies that do not exhibit poor absolute performance can represent significant opportunistic cost. System reconfiguration may be a promising technique to avoid such lost performance.

Acknowledgment

We thank the anonymous SIGMETRICS reviewers and our shepherd Erez Zadok for comments that helped improve this paper.

8. REFERENCES

[1] Realistic nonstationary online workloads. http://www.cs.rochester.edu/u/stewart/models.html.

[2] MySQL JDBC driver. http://www.mysql.com/products/connector.

[3] R. A. Fisher. The arrangement of field experiments. *J. of the Ministry of Agriculture of Great Britain*, 33:503–513, 1926.

[4] M. Grindal, J. Offutt, and S. F. Andler. Combination testing strategies: A survey. *Software Testing, Verification and Reliability*, 15(3):167–199, Mar. 2005.

[5] S. Iyer and P. Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *18th ACM Symp. on Operating Systems Principles*, pages 117–130, Banff, Canada, Oct. 2001.

[6] N. Joukov, A. Traeger, R. Iyer, C. P. Wright, and E. Zadok. Operating system profiling via latency analysis. In *7th USENIX Symp. on Operating Systems Design and Implementation*, pages 89–102, Seattle, WA, Nov. 2006.

[7] C. Li and K. Shen. Managing prefetch memory for data-intensive online servers. In *4th USENIX Conf. on File and Storage Technologies*, pages 253–266, Dec. 2005.

[8] C. Li, K. Shen, and A. Papathanasiou. Competitive prefetching for concurrent sequential I/O. In *Second EuroSys Conf.*, pages 189–202, Lisbon, Portugal, Mar. 2007.

[9] Linux kernel bug tracker. http://bugzilla.kernel.org/.

[10] Linux kernel bug tracker on "many pre-mature anticipation timeouts in anticipatory I/O scheduler". http://bugzilla.kernel.org/show_bug.cgi?id=10756.

[11] M. P. Mesnier, M. Wachs, R. R. Sambasivan, A. X. Zheng, and G. R. Ganger. Modeling the relative fitness of storage. In *ACM SIGMETRICS*, pages 37–48, San Diego, CA, June 2007.

[12] P. Reynolds, C. Killian, J. Wiener, J. Mogul, M. Shah, and A. Vahdat. Pip: Detecting the unexpected in distributed systems. In *Third USENIX Symp. on Networked Systems Design and Implementation*, San Jose, CA, May 2006.

[13] RUBiS: Rice University bidding system. http://rubis.objectweb.org.

[14] Y. Rubner, C. Tomasi, and L. J. Guibas. The earth mover's distance as a metric for image retrieval. *Int'l J. of Computer Vision*, 40(2):99–121, 2000.

[15] K. Shen, M. Zhong, and C. Li. I/O system performance debugging using model-driven anomaly characterization. In *4th USENIX Conf. on File and Storage Technologies*, pages 309–322, San Francisco, CA, Dec. 2005.

[16] C. Stewart, T. Kelly, and A. Zhang. Exploiting nonstationarity for performance prediction. In *Second EuroSys Conf.*, pages 31–44, Lisbon, Portugal, Mar. 2007.

[17] C. Stewart and K. Shen. Performance modeling and system management for multi-component online services. In *Second USENIX Symp. on Networked Systems Design and Implementation*, pages 71–84, Boston, MA, May 2005.

[18] E. Thereska and G. R. Ganger. IRONModel: Robust performance models in the wild. In *ACM SIGMETRICS*, pages 253–264, Annapolis, MD, June 2008.

[19] A. Traeger, I. Deras, and E. Zadok. DARC: Dynamic analysis of root causes of latency distributions. In *ACM SIGMETRICS*, pages 277–288, Annapolis, MD, June 2008.

[20] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou. Triage: Diagnosing production run failures at the user's site. In *21th ACM Symp. on Operating Systems Principles*, pages 131–144, Stevenson, WA, Oct. 2007.

[21] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang. Automatic misconfiguration troubleshooting with PeerPressure. In *6th USENIX Symp. on Operating Systems Design and Implementation*, pages 245–258, San Francisco, CA, Dec. 2004.

[22] A. Zeller. Isolating cause-effect chains from computer programs. In *10th ACM Symp. on Foundations of Software Engineering*, pages 1–10, Charleston, SC, Nov. 2002.