Comprehending Performance from Real-World Execution Traces: A Device-Driver Case

Conference Paper *in* ACM SIGPLAN Notices · February 2014

DOI: 10.1145/2541940.2541968

4 authors, including:

Shi Han
Microsoft
33 PUBLICATIONS **642** CITATIONS
SEE PROFILE

Dongmei Zhang
Nantong University
188 PUBLICATIONS **3,324** CITATIONS
SEE PROFILE

Tao Xie
University of Illinois, Urbana-Champaign
290 PUBLICATIONS **10,448** CITATIONS
SEE PROFILE

Comprehending Performance from Real-World Execution Traces: A Device-Driver Case

Xiao Yu
North Carolina State University
xyu10@ncsu.edu

Shi Han Dongmei Zhang
Microsoft Research
{shihan, dongmeiz}@microsoft.com

Tao Xie
University of Illinois at Urbana-Champaign
taoxie@illinois.edu

Abstract

Real-world execution traces record performance problems that are likely perceived at deployment sites. However, those problems can be rooted subtly and deeply into system layers or other components far from the place where delays are initially observed. To tackle challenges of identifying deeply rooted problems, we propose a new trace-based approach consisting of two steps: impact analysis and causality analysis. The impact analysis measures performance impacts on a component basis, and the causality analysis discovers patterns of runtime behaviors that are likely to cause the measured impacts. The discovered patterns can help performance analysts quickly identify root causes of perceived performance problems. We instantiate our approach to study the performance of device drivers on over 19,500 real-world execution traces. The impact analysis shows that device drivers constitute a non-trivial part ($\approx 38\%$) in the overall system performance, and a big part ($\approx 26\%$) is due to interactions between drivers. The causality analysis effectively discovers highly suspicious and high-impact behavioral patterns in device drivers, examined and confirmed by our automated evaluation, developers, and performance analysts.

Categories and Subject Descriptors D.4.8 [*Operating Systems*]: Performance; D.2.5 [*Software Engineering*]: Testing and Debugging—Diagnostics, Tracing; D.2.8 [*Software Engineering*]: Metrics—Performance measures

Keywords Device Drivers, Execution Traces, Performance Analysis, Bottlenecks, Contrast Data Mining

ASPLOS '14, March 1–4, 2014, Salt Lake City, Utah, USA.

http://dx.doi.org/10.1145/2541940.2541968

1. Introduction

Real-world execution traces collected by mature infrastructures, such as the Event Tracing for Windows (ETW) [1] and DTrace [2] for Unix-like systems, offer tracing events with timestampes and callstacks for both user-mode applications and system layers. Such traces can record performance problems that are likely perceived at deployment sites and usually triggered by complex runtime environments and real-world usage scenarios. By analyzing large-scale and diverse execution traces collected from deployment-site computers, performance analysts and developers can gain useful knowledge for system design and optimization.

However, finding performance problems from real-world traces is challenging. The problems can be rooted subtly and deeply into system layers or other components far from the place where delays are initially observed. Performance analysts have to spend great efforts to confirm the existence of subtle problems, and to figure out how they were caused and spread. Given many real-world execution traces where bad performance may be amortized over various underlying problems, such performance-analysis activities become harder or even infeasible.

Based on our observation in real-world traces, we identify and generalize the concept of *cost propagation* in systems consisting of interacting components as an important underlying mechanism that introduces the subtleness of performance problems. In particular, the term "cost propagation" refers to a phenomenon that the cost of executing a component is propagated and accumulated to the execution of another component through cross-component interactions.

There are different kinds of interactions causing cost propagation. Call dependency is a well-known kind, which accumulates costs from callees to callers. Lock contention is a subtle kind, which allows a delay to be propagated to all other components waiting for the same lock. In complex software ecosystems (e.g., Windows systems), different kinds of interactions can be combined together to create performance bottlenecks involving multiple victims. For example, there can be a very long propagation path or multiple

propagation paths consisting of both call dependencies and contentions on different locks.

The cost propagation brings challenges to performance analysis of complex systems. Existing techniques, such as call-graph profiling [14] and lock-contention analysis [36], face two major limitations. First, these techniques cover a single aspect of underlying interactions only, which is either call dependency or lock contention. Their combinatorial effects are not generally considered or addressed. Second, these techniques usually work on a single program with a limited investigation scope, but cannot explain performance problems with complex cause-and-effect chains that are transparent to the program, but are grounded in the underlying system. To better reveal and explain these problems, a large number of system-level execution traces can be leveraged from real-world deployment sites. Handling these traces requires effective abstractions that can capture essential information, and classify similar and distinct runtime behaviors towards providing concise and actionable results.

To address these challenges, we propose a new approach consisting of two steps: impact analysis and causality analysis. The impact analysis extracts *Wait Graphs* [16] (Section 3.1) from execution traces, and measures performance impacts with respect to cost propagation on Wait Graphs. Performance analysts can narrow down the investigation scope by choosing highly suspicious components to measure, and decide whether the causality analysis is needed based on the significance of performance impacts.

The causality analysis discovers behavioral patterns of highly suspicious and high-impact components that are likely to cause perceived performance problems. To capture the essence of problematic behaviors with respect to cost propagation, we design the *Signature Set Tuple* (Section 4.1) as a pattern representation that presents abstracted and generalized behaviors to performance analysts. A Signature Set Tuple generalizes interacting behaviors among components in the form of signature sets (a set of function signatures extracted from callstacks). Such abstraction and generalization facilitate the comprehension of performance problems by narrowing down the investigation scope as well as preserving highly suspicious behaviors that may appear similarly in different traces.

The causality analysis is grounded in the technique of contrast data mining [7] to address challenges in discovering problematic behaviors. Such challenges are two-fold: bad performance can be amortized over multiple problematic behaviors, and there is no easy oracle for identifying problematic behaviors, especially for the expensive but necessary ones. Our technique exploits the fact that a large number of execution traces may contain various runtime behaviors with both good and bad performance. In particular, our technique defines two contrast classes over execution traces as a fast class and a slow class based on thresholds of execution time (usually specified by developers as the expectations on performance). The fast class contains expected runtime behaviors, whereas the slow class contains problems to be identified. Our technique applies two criteria to identify contrast patterns: (1) a pattern appears in the slow class but not in the fast class; (2) a pattern appears in both classes, but the pattern in the slow class shows a significant amount of execution time compared with the pattern in the fast class. The identified contrast patterns can reflect underlying key factors that discriminate the performance contrasts.

In this paper, we choose device drivers in Windows operating systems as a representative to demonstrate our approach and study performance problems. Device drivers constitute a majority part (about 70%) in the Linux code base [26], as well as in the Windows kernel [21]. Performance analysts in Microsoft product teams have noticed that a non-trivial number of response delays in user-mode applications were likely caused by device drivers. Therefore, it is interesting, as well as necessary, to study how device drivers affect overall system performance.

We notice that two key characteristics in device drivers can cause cost propagation. First, locks are widely used in device drivers to synchronize shared resources over the system kernel. Second, device drivers are organized in a hierarchical architecture (a.k.a driver stack) in which drivers can interact with each other via system services, e.g., the `IoCallDriver` routine in Windows. The form of hierarchical dependencies connecting multiple contention points of different locks can propagate a delay to all affected drivers and corresponding user-mode applications to create remarkable performance problems.

Main Contributions. In this paper, we make main contributions from two aspects. On performance analysis, we propose a practical two-step approach with effective data and pattern abstractions to measure performance impacts manifested through cost propagation, and discover behavioral patterns closely related to performance problems via contrast data mining. The effectiveness of the pattern discovery is examined and confirmed by our automated evaluation, driver developers, and performance analysts.

By applying the impact analysis on 19,500 real-world execution traces (339 hours of duration in total, collected from Windows machines), we show the empirical evidence of performance impacts posed by device drivers. Our findings show that device drivers constitute 36.4% on waiting time and 1.6% on running time in overall Windows performance, within which cost propagation causes 26.0% on waiting time. Moreover, by applying the causality analysis on some typical application scenarios, we show concrete examples of performance problems related to device drivers. These findings suggest that identifying and minimizing potential cost propagation in device drivers should be an important direction for designing a high-performance system, especially for the cases involving driver/driver interactions, which complement the discussion by Kadav *et al.* [21].

2. Background and Motivating Examples

In this section, we first briefly introduce the basic notions about execution traces, and then use a real-world performance problem to show cost propagation in device drivers, and how our approach helps analysts find such problem.

2.1 Execution Traces

Execution traces are the data source of our trace-based performance analysis. To simplify the data representation while preserving essential information, we use an abstracted schema called *trace stream*. The schema is compatible with existing popular tracing infrastructures, e.g., ETW [1] and DTrace [2].

Trace Stream. A trace stream TS is a sequence of tracing events $e_0 e_1 e_2 \dots e_{L-1}$. Each event e falls into one of the following four types: (1) a *running* event represents CPU usage sampled in a constant interval, e.g., 1 millisecond in ETW and DTrace; (2) a *wait* event occurs when a thread enters the waiting state due to blocking operations, e.g., a thread tries to acquire a lock being held by others; (3) an *unwait* event occurs when a running thread signals another thread in the waiting state to continue execution, e.g., a thread releases a lock or sends a message; (4) a *hardware service* event is recorded with a start timestamp and duration in the period of a hardware operation.

Each event e is also related to a set of fields, which are callstack (denoted as $e.S$), timestamp ($e.T$), cost as time duration ($e.C$), the related thread ID ($e.TID$), and the ID of the other thread to be unwaited ($e.WTID$). These fields are sufficient for capturing and characterizing essential runtime behaviors under performance analysis from a cost-propagation perspective.

Scenario. Performance analysts usually start performance analysis with a scenario, e.g., *BrowserTabCreate* for creating a browser tab, to explore what run slow in that scenario. During a period of time, a system could have multiple ongoing scenarios simultaneously, since a user can browse websites while the system is performing other operations. Performance analysts have a set of predefined scenarios that are used to capture scenario-related execution traces.

Scenario Instance. A trace stream can contain events for multiple scenarios that were being performed by the system during the tracing period. Among these events, an event sequence representing the execution of a single scenario is called a *scenario instance*. A scenario instance typically starts and ends with the events from a single initiating thread, which initiates the execution of a particular scenario. For example, a browser UI thread that reacts to a user's request of creating a new browser tab is an initiating thread for *BrowserTabCreate*. Formally, a scenario instance *I* of a scenario *S* is a tuple $\langle TS, S, TID, t_0, t_1 \rangle$, indicating that the execution of the scenario *S* starting from a thread (with the identifier *TID*) within the time period between t_0 and t_1 is recorded in a trace stream *TS*.

Some events may overlap across multiple instances in the same trace stream. Such overlap indicates that other threads are suspended by the thread that triggers those overlapped events, so it is a typical manifestation of cost propagation. Such observation is the key to our approach to be effective in analyzing performance problems related to cost propagation.

2.2 A Real-World Case

We use a real-world case to show how cost propagation in device drivers is involved in producing a performance problem. Due to confidentiality, we anonymize the names of device drivers and relevant resources in this example.

An execution instance of the scenario *BrowserTabCreate* cost over 800 milliseconds to complete. From the user's perspective, the web browser fully displayed a new tab in over 800 milliseconds after the user had clicked "create a new tab". Such delay is perceivable on the user interface.

This problem involves three device drivers having lock contentions and hierarchical dependencies. First, File Virtualization filter driver *fv.sys* uses locks to synchronize queries on the entries of an internal *File Table* that maps some "virtual" files to their physical locations on the file system. Second, File System driver *fs.sys* acquires locks on *Meta Data Unit*s (*MDU*s) that contain file metadata when there are requests on reading or writing a file. Third, Storage Encryption driver *se.sys* performs computation-intensive encryption and decryption when data are being read and written on storage media. The three device drivers form a hierarchy in which the top-most driver *fv.sys* invokes *fs.sys*, whereas *fs.sys* invokes *se.sys*. To ease the illustration, we omit those less relevant drivers that may exist in the hierarchy on a real machine.

Figure 1 is a thread-level snapshot restructured from the trace stream to show the period of delay. The figure presents each thread by its corresponding callstack that contains ongoing operations. There were six threads executing the three device drivers during this period. We denote these threads by the notation $T_{X,Y}$, which means the thread Y of the process X. The dotted arrows in the figure represent the directions of cost propagation among threads.

Lock Contentions on the File Table. When the user clicked "create a new tab" on the browser, the browser UI thread $T_{B,UI}$ started to access some "virtual" files. So $T_{B,UI}$ executed *fv.sys* to query the *File Table*. However, two other worker threads $T_{B,W0}$ and $T_{B,W1}$ were performing file operations in the background, and happened to execute *fv.sys* at the same time. As a consequence, the three threads were contending a lock on some entries in the *File Table*, thereby forming a region of lock contentions in *fv.sys* (shown as the upper dashed box in Figure 1). Among these three threads, $T_{B,W1}$ first got the lock, so the other two threads had to wait.

Hierarchical Dependency between fv.sys and fs.sys. $T_{B,W1}$ proceeded to execute *fs.sys* by a function call initiated from *fv.sys*, thereby creating a hierarchical dependency between the two device drivers (shown as the arrow (4)

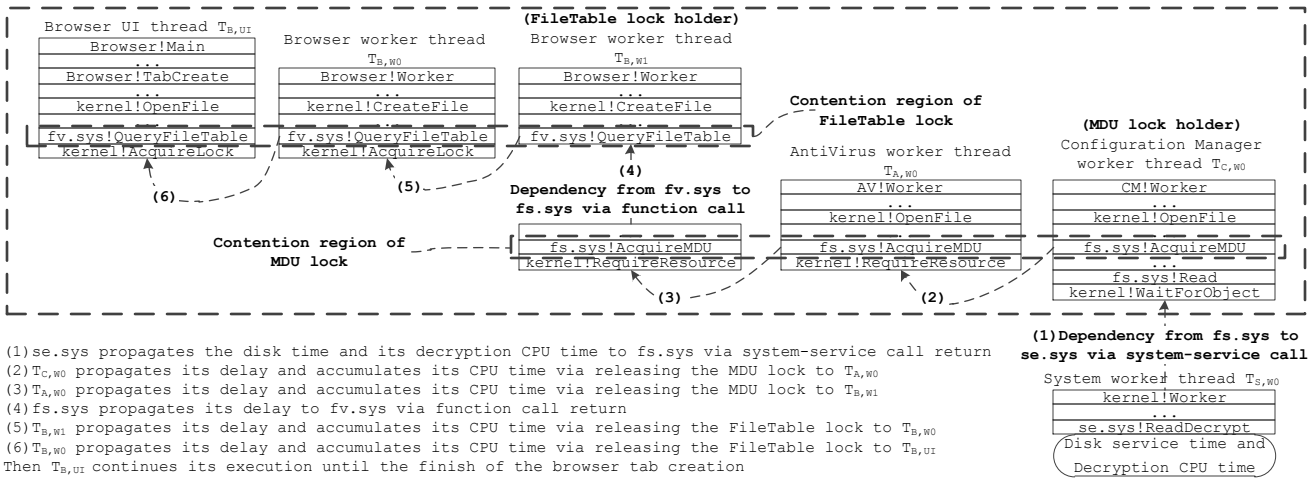

Figure 1. An illustration of cost propagation due to lock contentions and hierarchical dependencies among device drivers

in Figure 1). Thus, the continuation of executing *fv.sys* depended on the return of *fs.sys*.

Lock Contentions on Meta Data Units (MDUs). During the period that $T_{B,W1}$ was running, two threads $T_{A,W0}$ and $T_{C,W0}$ from two other applications (i.e., the AntiVirus and the Configuration Manager applications in Figure 1) were contending a lock on *MDU*s in *fs.sys*. Coincidentally, $T_{B,W1}$ joined to contend the same lock. Therefore, $T_{B,W1}$, $T_{A,W0}$, and $T_{C,W0}$ created another region of lock contentions in *fs.sys*, shown as the lower dashed box in Figure 1. $T_{C,W0}$ first got the lock, and proceeded.

Hierarchical Dependency between fs.sys and se.sys. $T_{C,W0}$ established a hierarchical dependency between *fs.sys* and *se.sys* by a system-service call that reads data from disk (shown as the arrow (1) in Figure 1). The system thread $T_{S,W0}$ was scheduled to serve the read request by fetching data from disk and executing *se.sys* to decrypt. To accomplish the task, $T_{S,W0}$ cost hundreds of milliseconds.

Summary: Manifestation of Cost Propagation. The three device drivers together created a bottleneck that consisted of two regions of lock contentions, and two hierarchical dependencies over the six threads (i.e., three browser threads, one AntiVirus thread, one Configuration-Manager thread, and one system thread). A delay occurring in $T_{S,W0}$ was propagated and accumulated through the path shown as the arrows from (1) to (6) in Figure 1 to the UI thread $T_{B,UI}$. Consequently, the user perceived an obvious delay when clicking a button on the browser to create a new tab. In addition, the other two applications in this case were also affected by such performance impact. Reducing the granularity of locks is a general principle to alleviate such problem.

2.3 How Our Approach Facilitates Performance Analysis

The preceding example shows the subtleness caused by cost propagation in real-world performance problems. In practice, a performance analyst has to start with the browser UI thread to examine many executed functions in many traces, since the costs may vary in different executions. Such work is usually tedious. Even if the analyst finds out that *fv.sys!QueryFileTable* could incur high execution cost in lock contentions among different browser threads under some circumstances, the analyst still needs to realize the relations across three drivers and many threads that constituted the whole performance. Realizing such relations is the key to future performance tuning.

By employing impact analysis, the analyst can first realize to which extent the performance of scenario *BrowserTabCreate* was affected by cost propagation. The analyst may conduct impact analysis on different scopes to realize performance impacts of different components. The results serve as the preliminary evidence pointing to some potentially hidden problems. Then the analyst can apply causality analysis on high-impact components, for instance, device drivers in this case. The causality analysis can suggest a list of contrast patterns sorted by their performance impacts. We pick the following one to explain in detail. The pattern is in the form of Signature Set Tuple, which is formally defined in Section 4.1.

$$wait\ signatures : \{fv.sys!QueryFileTable, fs.sys!AcquireMDU\}$$
$$unwait\ signatures : \{fv.sys!QueryFileTable, fs.sys!AcquireMDU\}$$
$$running\ signatures : \{se.sys!ReadDecrypt, DiskService\}$$

This pattern describes that the cost of the running signatures *se.sys!ReadDecrypt* and *Disk Service* can be propagated through the two unwait signatures *fv.sys!QueryFileTable* and *fs.sys!AcquireMDU* to the wait signatures, while functions represented by the wait signatures are invoked by the browser threads. The causality analysis can discover this pattern because in normal cases either such pattern does not appear, or it has much lower cost.

The discovered pattern can help an analyst from two aspects. First, it guides the analyst to realize the concrete performance incident by investigating a specific trace stream. The preceding example is actually restructured with the help of the discovered pattern. Second, the pattern as a generalized representation is a clue for similar cases. The analyst may prioritize the search of the three driver signatures in other cases to facilitate future analysis.

3. Impact Analysis

The goal of impact analysis is to scope and measure performance impacts for some chosen components, such as all device drivers. The impact analysis takes two inputs: (1) the instances of various scenarios over trace streams; (2) the component name(s) that are used to filter tracing events for the chosen components to be measured.

The impact analysis outputs three metric values: (1) the running percentage IA_{run}; (2) the wait percentage IA_{wait}; (3) IA_{opt}, the percentage of waiting time introduced by cost propagation.

The three metrics suggest three performance aspects of the chosen components. A large IA_{wait} indicates that the execution of the components is frequently blocked by others, whereas a large IA_{run} reflects a computation-intensive characteristic. IA_{opt} suggests how much extra cost is introduced by waiting on others. It can also serve as an upper bound for the optimization potential.

3.1 Data Abstraction

The impact analysis measures the running time and the waiting time for the chosen components, respectively. To serve this purpose, we use the Wait Graph structure from StackMine [16] to model scenario instances. A Wait Graph encodes *wait* and *unwait* events into wait chains, with *running* events as well. So it enables an easy way to measure both running and waiting time for impact analysis; otherwise, we would need to simultaneously keep track of each *wait* event to its corresponding *unwait* event.

DEFINITION 1. *A Wait Graph is a graph $WG = \langle V, E \rangle$. V is a set of nodes $V = \{e\}$, where e is a tracing event. E is a set of directed edges $E = \{e_i \rightarrow e_j\}$. For each edge $e_i \rightarrow e_j$, e_i must be a wait event, indicating that the thread triggering e_i keeps suspended before e_j occurs and completes, and e_j is triggered by another thread during the wait interval of e_i.*

The Wait Graph of a scenario instance is basically constructed by (1) pairing each *wait* event with its corresponding *unwait* event to restore wait chains among threads, (2) edging events as graph nodes based on the restored wait chains, and (3) restoring the duration of *wait* events from the timestamps on paired *wait/unwait* events. StackMine [16] contains an algorithm description for constructing a Wait Graph. We construct Wait Graphs based on that algorithm for our impact analysis and causality analysis with characteristics of Windows systems.

3.2 Impact Measurement

Basic Metrics. For every scenario instance, the impact analysis constructs a corresponding Wait Graph. To evaluate the values for IA_{run}, IA_{wait}, and IA_{opt}, the impact analysis introduces the following metrics that are calculated from all constructed Wait Graphs.

The total duration D_{scn} defines the aggregated execution time of all scenario instances in the input trace streams. Given the Wait Graph for each scenario instance, the impact analysis evaluates D_{scn} by adding up the time periods of top-level tracing events in the Wait Graph instance by instance.

The total wait duration D_{wait} defines the aggregated wait time cost by the chosen components in waiting for others. To determine which components should be counted, the impact analysis uses the names of the chosen components to filter the topmost signatures on the callstacks of tracing events. The impact analysis uses a breadth-first search on the Wait Graphs, and adds up the time periods of only top-level *wait* events of the chosen components to avoid counting child events that constitute the time cost already counted from their parent events.

The total running duration D_{run} defines the aggregated running time cost by the chosen components. Similar to D_{wait}, the impact analysis evaluates D_{run} by counting duration of the *running* events that contain signatures of the chosen components. Note that (1) some portions of periods in D_{run} are overlapped with D_{wait} because the *running* events are mostly the leaf nodes of some *wait* events; (2) D_{run} is an approximate value since ETW samples *running* events every millisecond and thus those running costs are at the millisecond granularity.

The total distinct-wait duration $D_{waitdist}$ defines the aggregated duration of *distinct wait* events of the chosen components over all scenario instances. A *wait* event may be involved in multiple Wait Graphs for multiple scenario instances, indicating that (1) the enclosing instances are captured in a common or close period of time from the same deployment site, and (2) the event has performance impact on multiple instances. The impact analysis measures $D_{waitdist}$ by excluding the duration of duplicate events across different Wait Graphs from D_{wait}.

Output-Metric Derivation. The impact analysis derives the values of output metrics IA_{run}, IA_{wait}, and IA_{opt} from the preceding metrics. In particular, $IA_{run} = D_{run}/D_{scn}$ represents the performance impact of running time, and $IA_{wait} = D_{wait}/D_{scn}$ represents the performance impact of wait time.

For deriving the severity of cost propagation across different scenario instances, there is $IA_{opt} = (D_{wait} - D_{waitdist})/D_{scn}$. IA_{opt} comes from the observation that any performance impact involving the chosen components

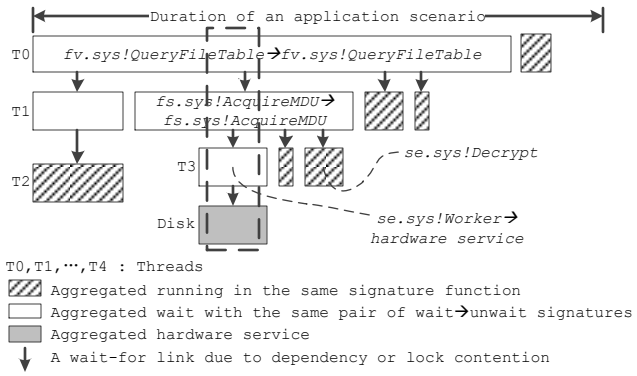

Figure 2. An illustration of an Aggregated Wait Graph for device drivers

can affect multiple different scenario instances. Namely, $D_{wait}/D_{waitdist} > 1$, which describes that the $D_{waitdist}$ of duration actually causes D_{wait} long wait among multiple scenario instances. The cost propagation is the underlying reason causing such performance impact. For instance, in the motivating example (Section 2.2), the delay initiated by $T_{S,W0}$ actually affected not only *BrowserTabCreate*, but also other scenario instances corresponding to two other applications along the propagation path. If we suppose the optimal case D^{opt}_{scn} as removing cost propagation between irrelevant scenarios, the extra percentage of cost introduced by cost propagation to the entire system would be $1 - D^{opt}_{scn}/D_{scn} = (D_{wait} - D_{waitdist})/D_{scn}$. This value is also an upper bound for the optimization of cost propagation. However, the actual optimization depends on the root causes of performance problems and other factors.

4. Causality Analysis

If the results from impact analysis indicate that the chosen components are of high impact, performance analysts can use causality analysis to discover runtime behaviors that may cause the observed performance impacts.

We adapt contrast data mining [7] to achieve causality analysis. Contrast data mining is the mining of patterns contrasting two or more classes for knowledge discovery [7]. In the setting of performance analysis, there are few specifications that can be used to distinguish between normal and problematic behaviors affecting performance, but we do know the difference on performance measurements. Contrast data mining can thus serve as a link from effect to cause. Namely, it allows us to discover behaviors that may cause the observed difference on measurements.

4.1 Data Abstraction and Pattern Representation

To adapt contrast data mining in performance analysis, we introduce the Signature Set Tuple as the pattern representation, and derive the Aggregated Wait Graph as the data abstraction from Wait Graphs. So the contrast data mining can discover contrast patterns in the form of Signature Set Tuple among different Aggregated Wait Graphs that are of distinguishable performance measurements.

Aggregated Wait Graph. An Aggregated Wait Graph is essentially the abstraction and aggregation of runtime behaviors in a set of Wait Graphs belonging to the same scenario, based on the fact that runtime behaviors in the same scenario represent the similar tasks under investigation. The definition of an Aggregated Wait Graph is as follows, in which the term *signature* specially denotes the topmost signature related to the chosen components on the callstack *e.S* of a tracing event *e*, if there exists such signature on the callstack.

DEFINITION 2. *An Aggregated Wait Graph is a graph* $AWG = \langle V, E \rangle$. *V is a set of nodes* $V = \{v\}$, *in which each node represents the aggregated execution of a function signature in one of the following statuses: running, waiting, or hardware service. E is a set of directed edges* $E = \{v_i \rightarrow v_j\}$, *in which each edge must start from a waiting node, indicating that the pointed node performs its operation within the waiting cost of the starting node.*

In addition, we attach some extra properties to the nodes in an Aggregated Wait Graph to ease our analysis.

DEFINITION 3. *In an Aggregated Wait Graph, each running node has a signature v.r; each hardware-service node has a dummy signature v.h; and each waiting node has two signatures: a wait signature v.w and its paired unwait signature v.u. Each node has a performance metric v.C as its duration of execution time, and has an occurrence counter v.N indicating the number of the same nodes from the source Wait Graphs.*

Figure 2 shows a partial Aggregated Wait Graph for the motivating example in Section 2.2. The region highlighted with a dashed box shows an aggregated path that represents a group of paths of similar cost propagation from multiple scenario instances. The aggregated path captures cost propagation from a hardware service via the *se.sys* driver and then the *fs.sys* driver, finally to *fv.sys*.

Signature Set Tuple. Causality analysis presents mined patterns in the form of Signature Set Tuple. A Signature Set Tuple generalizes runtime interactions related to cost propagation into three signature sets: a wait-signature set, an unwait-signature set, and a running-signature set. The wait-signature set contains signatures that reside in *wait* events; namely, such signatures can cause the caller thread to be suspended. The unwait-signature set contains signatures from *unwait* events, which signal suspended threads to continue. The running-signature set contains signatures recorded in *running* events, or a dummy signature representing *hardware service* events. We extract Signature Set Tuples from path segments on Aggregated Wait Graphs. The definitions of the path segment and Signature Set Tuple are as follows.

DEFINITION 4. *A path segment S of an Aggregated Wait Graph AWG is a sequence of nodes* $S := \langle v_i \rangle : v_0 v_1 \dots v_{L-1}$,

where $v_i \in AWG.V$ and for $0 \leq i < L-1, v_i \rightarrow v_{i+1} \in AWG.E$. The performance metric of S is defined as the metric of its end node, i.e., $S.C \coloneqq v_{L-1}.C$, as well as the occurrence counter $S.N \coloneqq v_{L-1}.N$.

DEFINITION 5. *A Signature Set Tuple P from a path segment S is in the form of $P(S) \coloneqq \langle \bigcup v.w, \bigcup v.u, \bigcup v.r \rangle$, where $v \in S$. The performance metric of P(S) is defined as the metric of the path segment, i.e., $P(S).C \coloneqq S.C$, as well as the occurrence counter $P(S).N \coloneqq S.N$.*

We design such Signature Set Tuple based on two key rationales. First, the three signature sets appropriately generalize the phenomenon that the cost of some time-consuming operations (represented by the running-signature set) propagates to other parts in the system via interactions in the direction from functions represented by the unwait-signature set to those of the wait-signature set, regardless of the particular kinds of interactions. Second, signature sets can accommodate variations of the cost-propagation sequences. Consider a case of device drivers, in which two drivers contend a resource held by the third driver. Such case creates two possible execution sequences in reality, depending on which driver acquires the resource first. A pattern in the Signature Set Tuple can represent both possibilities.

4.2 Pattern Discovery

Given a set of instances of a particular scenario, and two performance thresholds T_{fast} and T_{slow} as inputs, the causality analysis adapts contrast data mining, and works in three steps: (1) classifying contrast classes, (2) constructing data abstractions, and (3) mining contrast patterns.

4.2.1 Classifying Contrast Classes

The causality analysis first classifies these instances into two contrast classes $\{I\}_{fast}$ and $\{I\}_{slow}$ by comparing their recorded execution time against performance thresholds T_{fast} and T_{slow}. In practice, developers need to explicitly specify the two thresholds for each application scenario as a part of performance specification. Specifically, T_{fast} is the upper bound reflecting normal performance, and T_{slow} is the lower bound of performance degradation. For example, the *BrowserTabCreate* scenario of a web browser should be completed within 300ms, and should not exceed 500ms, which typically makes users feel slow. In this case, $T_{fast} = 300ms$ and $T_{slow} = 500ms$, so the two classes would be *less-than-300ms* as the fast class and *more-than-500ms* as the slow class. The contrasts between the two classes are highly suspicious to be the constitution of performance problems. Typically we have $T_{slow} - T_{fast} \gg 0$, enabling little confusion between the two contrast classes.

4.2.2 Constructing Data Abstractions

Given two contrast classes of instances $\{I\}_{fast}$ and $\{I\}_{slow}$, causality analysis constructs two Aggregated Wait Graphs AWG_{fast} and AWG_{slow} for the two classes. The corresponding Wait Graphs for the scenario instances to be aggregated come from the previous impact analysis. The aggregation uses the algorithm presented in Algorithm 1.

Algorithm 1: The algorithm to aggregate Wait Graphs

```
   Input: a set of Wait Graphs {WG}, a set of component names {C}
   Output: an Aggregated Wait Graph AWG
1  InitializeAWG(AWG);
2  foreach WG in {WG} do
3      repeat
4          rootNodes ← GetRootNodes(WG);
5          foreach e in rootNodes do
6              if e.S does not contain signature from {C} then
7                  RemoveNode(WG, e);
8      until ∀e ∈ rootNodes, e.S contains signature from {C};
9      foreach e in WG do
10         if e.Type is u then
11             e' ← GetSourceNode(WG, e);
12             MergeWaitUnwaitPair(WG, e, e');
13     foreach path ⟨e_0, e_1, ..., e_n⟩ from roots to sinks in WG do
14         MergeWithCommonSigPrefix(AWG, ⟨e_0, e_1, ..., e_n⟩);
15 ReduceAWG(AWG);
16 return AWG;
```

Algorithm 1 works in four steps. For each Wait Graph, it first eliminates nodes that are irrelevant to the components to be analyzed from Wait Graphs (Lines 3 to 8). Then it merges paired wait/unwait nodes (Lines 9 to 12). After that it aggregates the processed Wait Graph to the Aggregated Wait Graph (Lines 13 to 14). Finally, we introduce a heuristic to reduce the size of the graph in Line 15. We describe the key points for each step as follows.

Eliminating Component-Irrelevant Nodes. A node is component-irrelevant if none of the signatures on the callstack belongs to the specified components $\{C\}$. From roots of the graph, the algorithm removes each component-irrelevant node, and promotes its child nodes as new roots. The algorithm repeats until all root nodes are component-relevant. The processed Wait Graph contains only behaviors initiated by the components in $\{C\}$.

Merging Wait/Unwait Nodes. The algorithm merges *wait* events with corresponding *unwait* events as well as their edges to create waiting nodes (`MergeWaitUnwaitPair` in Line 12). The result structure represents that the child nodes perform operations within the cost of the parent node. In fact, the Aggregated Wait Graph is essentially a forest in which all inner nodes must represent *wait/unwait* event pairs, whereas all leaf nodes must be of *running* events or *hardware service* events.

Aggregating Wait Graphs. The algorithm aggregates the processed Wait Graphs based on common prefixes of paths (`MergeWithCommonSigPrefix` in Line 14). Paths having common prefix indicate that they have the identical sequence of signatures on nodes along their prefixes. Common prefixes indicate that there are common behaviors starting at the beginning of executions, and subsequent behaviors may be diverse due to external or internal reasons,

such as various inputs from higher-level applications, or lock contentions and different dependencies.

Non-Opimtizable Portions. To help subsequent mining and eliminate data noises, the algorithm performs a reduction on Aggregated Wait Graphs (`ReduceAWG` in Line 15). Specifically, it prunes all structures in the pattern of a waiting node as a root pointing to a single hardware-service leaf. This pattern implies that the performance impact of hardware services is not propagated to other components. Developers usually do not have any chances to optimize such cases.

4.2.3 Mining Contrast Patterns

The algorithm to discover contrast patterns between the two Aggregated Wait Graphs works in the following three steps. To address challenges from both data complexity and computational complexity, we particularly introduce a step of meta-pattern enumeration.

Enumerating Meta-Patterns. Given the two Aggregated Wait Graphs for the two contrast classes, the causality analysis first enumerates meta-patterns in the form of Signature Set Tuples from path segments in the two graphs. Specifically, we introduce a constant k to bound the maximum length of path segments in order to improve efficiency. Given a certain value of k, the causality analysis iteratively enumerates all path segments that have the lengths from 1 to k, and collects meta-patterns from these segments. Given two path segments having a common meta-pattern P, the causality analysis aggregates their $P.C$ and $P.N$ values. After this step, we have two groups of meta-patterns for the fast and slow contrast classes.

We have three considerations of enumerating meta-patterns from a bounded number of path segments rather than from all possible path segments or full paths in the two Aggregated Wait Graphs. First, contrast mining on simple data abstractions such as sets is of high complexity [37]. For a complex data structure such as an Aggregated Wait Graph with a potentially large amount of data in it, the computation is challenging. Second, in fact meta-patterns collected from a bounded number of path segments are adequate without loss of patterns, compared with expensive all-segment enumeration. The causality analysis enumerates path segments from the smallest length; thereby, all other patterns could be the combinations of the smaller ones. Third, using patterns collected from complete paths is too specific for causality analysis. If a pattern of a complete path from one contrast class does not appear in another contrast class, we cannot claim that this path is a contrast, because its meta-patterns may appear frequently in both classes. So by conducting the segment enumeration, we can have these meta-patterns to exclude non-contrast paths.

Discovering Meta-Pattern Contrasts. The contrasts among meta-patterns in the two groups infer the behaviors having potential impacts to performance. There are two criteria to discover such contrasts. First, a meta-pattern appearing only in the slow class indicates that runtime behaviors represented by it may harm the whole performance to some extent, since such behaviors never happen in the cases of normal performance. The causality analysis selects such meta-pattern as a contrast. Second, a meta-pattern that is common in the two classes, but whose attached performance metrics are significantly higher in the slow class, also implies highly suspicious and high-impact runtime behaviors to the performance. In particular, for the common meta-pattern P_s in the slow class, and P_f in the fast class, whenever $\frac{P_s.C}{P_s.N} / \frac{P_f.C}{P_f.N} > \frac{T_{slow}}{T_{fast}}$, the causality analysis selects the meta-pattern as a contrast.

Discovering Contrast Patterns. Finally, the causality analysis discovers contrast patterns based on the contrast meta-patterns in two steps. First, the causality analysis uses contrast meta-patterns as clues to select contrast paths. It computes a pattern in the Signature Set Tuple for each full path in the Aggregated Wait Graph of the slow class. If the extracted pattern contains any contrast meta-patterns, the causality analysis selects it as a contrast pattern. Second, the causality analysis merges identical contrast patterns with their $P.C$ and $P.N$ counters. Recall that multiple paths representing the same kind of problem could share the same pattern, even when their cost-propagation sequences might be different.

These contrast patterns are the output of causality analysis. To further facilitate manual efforts on inspection, we rank these patterns by their performance impacts, with the highest impacts ranked the first. We define the impact of a contrast pattern by its average execution cost in performing its task, i.e., $P.C/P.N$.

5. Empirical Results and Evaluation on Device Drivers

We apply our approach to study the performance of device drivers in Windows. We first perform impact analysis on device drivers, and then present an evaluation of using causality analysis to reveal patterns of performance problems.

Our data set contains about 19,500 trace streams, consisting of about 505,500 scenario instances that fall into 1,364 usage scenarios of Windows and various applications. The total recorded execution time is approximately 339 hours. These trace streams are collected by the ETW infrastructure.

Our study and evaluation offer developers and performance analysts two main benefits. First, the study and evaluation uncover the impact and problems of device drivers in the real world. Second, the study and evaluation show that our approach can facilitate performance analysis by reducing manual efforts on the inspection of performance problems.

5.1 Results of Impact Analysis on Device Drivers

We set the impact analysis to analyze device drivers exclusively over all scenario instances in the data set, namely, by setting the names of components to include all device drivers

(matching the wildcard pattern "*.sys" in all function signatures).

The impact analysis shows that the wait percentage $IA_{wait} \approx 36.4\%$, and running percentage $IA_{run} \approx 1.6\%$, which are compelling metrics to inform the performance impact. IA_{wait} directly explains what percentage device drivers block application executions. We can consider it as the average percentage of performance impact introduced by device drivers in each instance. IA_{run} indicates what percentage device drivers consume CPU time. The small percentage of IA_{run} indicates that the running time of device drivers is not a dominating part in overall system performance.

The impact analysis also shows that the percentage of extra waiting time $IA_{opt} \approx 26\%$, indicating the severity of cost propagation in device drivers, and the fact that the total execution time D_{scn} could be reducible by optimizing device drivers and/or relevant components. In addition, the ratio of the total wait duration to the total distinct-wait duration $D_{wait}/D_{waitdist} \approx 3.5$ shows that any cost involving device drivers in one scenario instance is propagated to the other 2.5 instances on average, indicating the severity of cost propagation from a high-level of view.

In summary, the results indicate that device drivers are worth being further investigated. In particular, the results disclose three aspects. First, device drivers constitute a non-trivial part ($\approx 36.4\%$) in OS and application performance. Second, the percentage of time likely wasted due to cost propagation would be 26%, which is also an upper bound for the optimization of cost propagation. Third, CPU consumption of device drivers has limited impact ($\approx 1.6\%$). It is consistent with the expectation that drivers do little computation [27], and thus their computation cost is not our focus of analysis.

5.2 Evaluation of Causality Analysis

Performance analysts can manually inspect output patterns to determine whether these patterns represent high-impact performance problems and need further actions towards optimization. To show the effectiveness of causality analysis, and how causality analysis can improve the efficiency of manual inspection, we address the following three research questions.

RQ1: To what extent would the discovered patterns explain the bad performance in device drivers?

RQ2: How does the ranking strategy applied on discovered patterns help improve efficiency of inspecting performance problems?

RQ3: What insights can the discovered patterns offer to performance analysts in identifying real performance problems?

5.2.1 Evaluation Setup

We apply the causality analysis on 8 selected out of all 1,364 scenarios. There are 17,612 instances in the selected scenarios. Table 1 shows these scenarios. We choose these scenarios for two reasons: (1) the selected scenarios represent a set of typical applications (e.g., web browsers) that need to interact with multiple device drivers, and (2) the number of instances in the selected scenarios are relatively large to avoid data noises and biases affecting contrast data mining. There are 2,201 instances for each selected scenario on average, compared with the overall average of 370 instances per scenario in our data set.

Scenario	#Instances	in $\{I\}_{fast}$	in $\{I\}_{slow}$
AppAccessControl	1547	598	772
AppNonResponsive	631	164	392
BrowserFrameCreate	1304	437	707
BrowserTabClose	989	134	678
BrowserTabCreate	2491	597	1601
BrowserTabSwitch	2182	1122	914
MenuDisplay	743	171	499
WebPageNavigation	7725	4203	1175
Total	17612	7426	6738

Table 1. Selected Scenarios

Table 1 shows the selected scenarios, the number of scenario instances, and the numbers of instances in the two contrast classes. For confidentiality, we anonymize the real names of the selected scenarios, and use T_{fast} and T_{slow} to denote the two thresholds. These two thresholds are determined by application vendors, and may vary in different scenarios. We set the component names $\{C\}$ to be all device drivers, and the maximum length of segment enumeration to be 5 in all experiments to focus on the propagation of performance impact up to that length.

To address RQ1, we first use an automated rule to determine whether the discovered patterns represent high-impact driver behaviors. In particular, we determine that a contrast pattern is of high impact if at least one of its executions in trace streams exceeds T_{slow}. Such high-impact pattern indicates that the involved drivers must be able to constitute the bad performance as a significant factor. Otherwise, the pattern's impact to the bad performance is relatively low. After classifying the discovered patterns by the automated rule, we show two execution-time coverages for the classified patterns over the total time cost of device drivers in the slow class. In particular, the impactful-time coverage (ITC) is the sum of the time cost $P.C$ for the high-impact patterns over the total time cost of device drivers, and the total-time coverage (TTC) is the sum of the time cost $P.C$ for all patterns over the total time cost of device drivers. The ITC suggests the scope where there *must* be high-impact driver behaviors causing the bad performance, and the TTC suggests the scope where there *may* be such behaviors. The ITC and TTC can show the effectiveness of causality analysis, because high coverages indicate that the discovered patterns can interpret the bad performance to a great extent, which offers a higher chance for performance analysts to find high-impact performance problems.

For RQ2, we show an execution-time coverage of top $n\%$ patterns based on the ranking strategy. In practice, a perfor-

Scenario (T_{slow})	Driver Cost	ITC	TTC
AppAccessControl	66.4%	18.9%	35.5%
AppNonResponsive	64.6%	41.0%	48.7%
BrowserFrameCreate	76.5%	24.1%	35.4%
BrowserTabClose	21.9%	27.1%	38.0%
BrowserTabCreate	51.3%	23.1%	35.3%
BrowserTabSwitch	41.0%	7.8%	17.5%
MenuDisplay	77.0%	39.2%	49.2%
WebPageNavigation	34.7%	18.4%	28.5%
Average	54.2%	24.9%	36.0%

Table 2. Impactful-Time and Total-Time Coverages

Scenario (T_{slow})	#Patterns	10%	20%	30%
AppAccessControl	4875	55.3%	91.1%	98.3%
AppNonResponsive	1158	29.6%	39.2%	95.1%
BrowserFrameCreate	1933	51.6%	92.0%	96.8%
BrowserTabClose	1075	55.1%	90.0%	93.5%
BrowserTabCreate	5045	49.0%	87.5%	97.0%
BrowserTabSwitch	1514	42.3%	64.9%	98.0%
MenuDisplay	1855	64.5%	86.5%	91.9%
WebPageNavigation	5122	35.6%	89.3%	96.5%
Average	2822	47.9%	80.1%	95.9%

Table 3. Coverages by Ranking

mance analyst needs to manually investigate the discovered patterns in trace streams to confirm the root causes, and prioritize the investigation efforts for high-impact patterns. So a relatively small $n\%$ with high coverage indicates that the ranking strategy prioritizes high-impact patterns properly, and can substantially improve the inspection efficiency.

For RQ3, we first present what kinds of drivers are involved in the top-10 patterns from each selected scenario (80 patterns in total). Then we describe three major observations that we can make from the relations between scenarios and drivers. We also use some representative cases to illustrate how causality analysis can help performance analysts identify real-world problems. Note that the analysis in this paper was conducted independently by the authors, but all cases described in detail have been reviewed and confirmed by performance analysts and developers at Microsoft. We are continuing the confirmation process for other cases.

5.2.2 RQ1: Effectiveness

We show the effectiveness of causality analysis by presenting the impactful-time and total-time coverages for the discovered patterns. Table 2 shows the results of the two coverages. The column "Driver Cost" presents total execution time of device drivers in each scenario. The columns "ITC" and "TTC" present the impactful-time and the total-time coverages on the total driver time.

Overall, the total driver cost in the slow class can be characterized by three portions. The first portion is represented by the driver behaviors that may account for the bad performance. The second one is represented by the driver behaviors that are likely of normal performance. The two portions do not have an easily distinguishable boundary. The causality analysis can help performance analysts identify the boundary. In particular, the contrast patterns that are counted in the ITC must fall into the first portion, whereas the rest of the contrast patterns may cross the boundary of the two portions, but still likely cause the bad performance. Let us take the *AppNonResponsive* scenario as an example. The total driver cost covers 64.6% of the total execution time. The ITC is 41.0%, and the difference between the ITC and the TTC is 7.7%. The contrast patterns counted in the ITC should be the first priority for performance analysts to inspect, and the rest of the patterns falling between the ITC and the TTC reflect a non-negligible amount of execution time that performance analysts should inspect to confirm potential problems.

On the other hand, the third portion is represented by the driver behaviors that directly interact with hardware without cost propagation. These behaviors are usually of high impact, but non-optimizable, since the real cost depends on external hardware. During the construction of an Aggregated Wait Graph, the reduction of non-optimizable portions would remove such behaviors, even though some of them could account for long-time executions. Since device drivers frequently interact with slow hardware, the non-optimizable portions are very common. For example, in the *BrowserTabSwitch* scenario, 66.6% of the total driver cost is manifested in direct hardware services without cost propagation, thus being removed from the Aggregated Driver Graph. The resulting graph represents the remaining 33.4%, and more than half of the remaining portions (17.5%) are represented by contrast patterns, and classified into the total-time coverage.

5.2.3 RQ2: Efficiency

We calculate another execution-time coverage to show the efficiency improvement brought by the causality analysis on manually confirming potential performance problems. Based on the ranking strategy used in the pattern discovery, we select the top 10%, 20%, and 30% contrast patterns, and measure their execution-time coverages over all discovered patterns. Table 3 shows the results, in which the column "#Patterns" presents the number of contrast patterns in each scenario, and the last three columns show the coverage for the top 10%, 20%, and 30% contrast patterns.

The numbers of the output patterns range from 1,075 to 5,122 across different scenarios. The thousand magnitude of patterns came from the scale of input traces and the high complexity of the OS ecosystem, especially the complexity of interactions and dependencies in device drivers. As shown in Table 3, the inspection of the top 10% patterns would involve 107 to 512 patterns and achieve 30% to 65% execution-time coverage. According to the evaluations provided for StackMine [16], a performance analyst could inspect the top 400 stack-trace patterns of an analysis within 8 hours to achieve 60% execution-time coverage in all analyzed regions of scenario instances, with over 90% inspection effort saved as an estimation. Such comparable experi-

ences indicate the affordability and efficiency regarding inspection efforts required by the causality analysis as well.

5.2.4 RQ3: Real Cases

Table 4 shows that in each scenario what kinds of drivers are involved in top-10 patterns. Each cell shows the number of patterns containing the corresponding type of drivers. From Table 4, we have three major observations regarding the relations between scenarios and drivers. These observations can suggest proper starting points for the performance analysis on similar cases.

First, most patterns contain both file-system drivers and filter drivers, especially in the *AppAccessControl* scenario. In fact, most of those filter drivers belong to security software. Security software uses system-wide filter drivers to intercept various application requests, but usually uses a single process and database for security inspection. Such architecture makes the performance of security inspection very critical. Once the workload of the system increases, it is easy to form bottlenecks starting from those filter drivers. If the system also enables storage encryption, the situation could become worse, as the encryption drivers could induce extra overheads. The example described in Section 2.2 is a typical case of how filter drivers combined with encryption drivers slow down the system. Developers should estimate the granularity and potential contentions of locks in filter drivers to alleviate overheads.

Second, some scenarios are especially vulnerable to particular types of drivers. In addition to the *AppAccessControl* scenario, which suffers from file-system drivers and filter drivers, we can observe that network drivers take a big part in the scenario of *MenuDisplay* (7 out of 10). It is understandable that network drivers can be delayed by unstable bandwidth. So if a menu needs to display items from remote servers, developers should take into account such instability, and use an asynchronous mechanism or prefetched cache to prevent network delays from being propagated to user interfaces.

Third, hard faults can establish more subtler interactions between drivers, and cause severe performance degradation. In the *AppNonResponsive* scenario, an interesting pattern shows that a graphics driver *graphics.sys* appears with a file system driver *fs.sys* and a storage encryption driver *se.sys*. From driver signatures we can infer that *graphics.sys* should not have interactions with the other two in normal cases, because it does not need to access files on disk. Based on our knowledge, it is highly suspicious that this pattern hints a hard fault occurring in *graphics.sys*.

From trace streams, we find a corresponding instance that costs about 4.73 seconds to finish the execution. To simply describe this case, we next describe only three threads that are most relevant. The initiating one was a UI thread $T_{U,UI}$. $T_{U,UI}$ was executing *graphics.sys*, and the driver tried to acquire resources of a Graphics Processing Unit (GPU). At the same time, there was a system worker thread $T_{S,W0}$ in which *graphics.sys* was executing a routine in response to a system event. Since $T_{S,W0}$ had acquired GPU resources, $T_{U,UI}$ had to wait. At this point, a hard fault occurred in $T_{S,W0}$ when *graphics.sys* initialized an internal structure. To solve the hard fault, the system scheduled another worker thread $T_{S,W1}$ to perform a page read. $T_{S,W1}$ was actually executing *se.sys* because the system was storage-encrypted. Finally, the system spent about 4.7 seconds to complete the page read. This significant performance degradation was then spread to $T_{U,UI}$, and made the user interface unresponsive.

The preceding case indicates that solving hard faults could be very time-consuming, so device drivers should be developed to minimize the usage of paged memory in order to avoid expensive disk I/O and the consequence of potential cost propagation.

5.2.5 Validity and False Positives in Analysis Results

To validate the analysis supported by our approach, we sent the case described in Section 2.2, a representative case of security software, and the hard fault in *graphics.sys* as high-impact problems to relevant performance analysts and developers at Microsoft, and received their confirmations. The high impact was partially confirmed after their frequent observation of such problems, reflecting the limitations of the conventional performance analysis. One developer from a Windows product team for driver quality provided his enthusiastically positive feedback: "*...It is very impressive to see how performance issues of device drivers can be root caused in such a detail... yes, this is the technique we are interested in w.r.t. driver qualities...*".

Except those high-impact real cases, we can observe some false positives. In some special circumstances, our approach fails to distinguish between by-design behaviors and problematic behaviors. For instance, in Table 4 there is a type of drivers called "Disk Protection", which can halt hard drives to prevent damages when a computer is in motion. Drivers of this type are designed to block all disk reads and writes when necessary, and meanwhile performance can be compromised. The appearance of such driver patterns suggests that we need to incorporate such knowledge to filter out some known and exceptional cases.

6. Related Work

Performance Analysis. The high-level goal of our work is to help with the performance analysis of software ecosystems, particularly the evaluation of impacts and identification of root causes for performance degradation, by acquiring the knowledge from a large number of execution traces. We organize the discussion of related work by three major aspects, specifically, (1) handling cross-component interactions, (2) leveraging multiple executions, and (3) oracles for finding root causes.

Cross-component interactions may work concurrently and cause the effect of cost propagation. The potential transparency of these interactions to individual components

Scenario	FileSystem, General Storage	FileSystem Filter	Network	Storage Encryption	Disk Protection	Graphics	Storage Backup	IO Cache	Mouse	ACPI
AppAccessControl	9	9	–	–	–	–	–	1	–	–
AppNonResponsive	6	2	1	2	1	1	–	–	–	1
BrowserFrameCreate	7	4	2	–	1	–	–	–	–	–
BrowserTabClose	5	6	–	2	–	–	2	–	–	–
BrowserTabCreate	5	6	3	2	–	1	–	–	1	–
BrowserTabSwitch	6	5	3	1	–	–	–	–	–	–
MenuDisplay	2	3	7	–	2	–	–	–	–	–
WebPageNavigation	7	3	3	1	1	–	–	–	–	–

Table 4. Top-10 Patterns Categorized by Driver Types

makes the diagnosis of performance to be even harder. Some existing work [4, 35] identified the idleness as a performance characteristic of multithreaded and multi-component programs. Tallent *et al.* [35, 36] analyzed lock-based interactions as the source of such idleness. However, their work is limited to isolate the effect of a single lock only. In contrast, our work reveals the combinatorial effect of multiple locks connected by hierarchical dependencies, which commonly exist in large and complex systems.

Multiple executions of a system can provide diverse performance information. StackMine [16] and Magpie [10] also work on multiple ETW traces. Our previous StackMine work discovers callstack patterns via costly-pattern mining, resulting in patterns capturing within-thread behaviors. Our current work complements it with the mining of contrast patterns characterizing cross-thread behaviors. Magpie correlates workloads with events synthesized from traces to build performance models that support performance prediction and debugging. Our work can further help Magpie diagnose root causes by using workload-based scenarios. Jovic *et al.* [19] targeted at finding and prioritizing performance bugs in GUI applications from multiple profiles in the wild. However, this work relies on the manual selection of landmark methods to measure latency, and has the limitation of not being able to handle concurrency.

Using proper oracles is helpful to find performance problems and root causes effectively. Rule-based problem identification [4, 5, 18, 38] is limited due to finding only problems defined by existing rules. Some work performs thresholding and filtering on performance measurements [32] combined with frequencies of observed system behaviors [3]. However, in complex systems, no clear-cut criteria can be used to attribute bad performance to behaviors that may be expensive in nature and vary in different usage scenarios. Our work leverages information from the labeled normal executions as the criteria addressing such variations. Some work [6, 15, 30] proposed similar ideas exploiting contrasts between different executions to identify performance problems. Compared to such previous work, our work does not require concrete inputs of systems. These inputs are not always available especially when performance data are collected from client sites. Furthermore, our work provides useful data abstractions that group similar behaviors in a large volume of data, and provides a pattern representation to represent actionable patterns to further narrow down the scope of the root-cause localization.

Quality Assurance for Device Drivers. Quality of device drivers includes mainly reliability, security, and performance. Most existing research work is to improve driver reliability [8, 9, 12, 13, 20, 22, 24, 26–29, 31, 33, 34]. While certain kinds of reliability issues are handled, e.g., memory corruption, the security is also improved. In addition, isolation-based approaches [11, 31] were also introduced to improve driver security. In contrast, little work has been proposed on driver performance. In fact, there is a trade-off between reliability (together with security) and performance. Leslie *et al.* [23] and Ganapathy *et al.* [13] demonstrated that it is possible to build user-level drivers to improve reliability without significant performance overheads. Menon *et al.* [25] explored the opportunity to balance reliability, security, and performance with a framework allowing to create safe and efficient hypervisor drivers. Huang and Chen [17] developed an approach to find performance bottlenecks of TCP/IP protocol and the corresponding drivers. Such work usually addresses driver performance by a specific analysis or from a specific perspective, but not from a general perspective of real-world production ecosystems.

7. Conclusion

We have proposed a performance-analysis approach on real-world execution traces to address the challenges posed by cost propagation in complex systems. We have applied impact analysis to study the performance of device drivers on 19,500 execution traces. The results show that device drivers constitute approximately 38% of the overall system performance, and a big part ($\approx 26\%$) is introduced by cost propagation. To understand how device drivers affect system performance, we have conducted causality analysis to discover patterns of problematic behaviors. The discovered patterns can help identify high-impact performance problems.

Acknowledgments

We would like to thank the conference reviewers and shepherds for their feedback in finalizing this paper. We would also like to thank the performance analysts and the developers at Microsoft who helped review and verify the studied cases.

References

[1] http://msdn.microsoft.com/en-us/library/windows/desktop/bb968803(v=vs.85).aspx.

[2] http://en.wikipedia.org/wiki/DTrace.

[3] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance Debugging for Distributed Systems of Black Boxes. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 74–89, New York, NY, USA, 2003. ACM.

[4] E. Altman, M. Arnold, S. Fink, and N. Mitchell. Performance Analysis of Idle Programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 739–753, New York, NY, USA, 2010. ACM.

[5] G. Ammons, J.-D. Choi, M. Gupta, and N. Swamy. Finding and Removing Performance Bottlenecks in Large Systems. In *Proceedings of ECOOP 2004 - Object-Oriented Programming*, volume 3086 of *Lecture Notes in Computer Science*, pages 172–196. Springer Berlin Heidelberg, 2004.

[6] M. Attariyan, M. Chow, and J. Flinn. X-ray: Automating Root-Cause Diagnosis of Performance Anomalies in Production Software. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, pages 307–320, Berkeley, CA, USA, 2012. USENIX Association.

[7] J. Bailey and N. V. Laboratory. Contrast Data Mining: Methods and Applications. *Tutorial at ICDM*, 2007.

[8] G. Balakrishnan and T. Reps. Analyzing Stripped Device-Driver Executables. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08, pages 124–140, Berlin, Heidelberg, 2008. Springer-Verlag.

[9] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough Static Analysis of Device Drivers. *SIGOPS Oper. Syst. Rev.*, 40(4):73–85, Apr. 2006.

[10] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for Request Extraction and Workload Modelling. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 18–18, Berkeley, CA, USA, 2004. USENIX Association.

[11] S. Boyd-Wickizer and N. Zeldovich. Tolerating Malicious Device Drivers in Linux. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'10, pages 9–9, Berkeley, CA, USA, 2010. USENIX Association.

[12] V. Ganapathy, A. Balakrishnan, M. M. Swift, and S. Jha. Microdrivers: A New Architecture for Device Drivers. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems*, HOTOS'07, pages 15:1–15:6, Berkeley, CA, USA, 2007. USENIX Association.

[13] V. Ganapathy, M. J. Renzelmann, A. Balakrishnan, M. M. Swift, and S. Jha. The Design and Implementation of Microdrivers. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 168–178, New York, NY, USA, 2008. ACM.

[14] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A Call Graph Execution Profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, CC'82, pages 120–126, New York, NY, USA, 1982. ACM.

[15] M. Grechanik, C. Fu, and Q. Xie. Automatically Finding Performance Problems with Feedback-Directed Learning Software Testing. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE'12, pages 156–166, Piscataway, NJ, USA, 2012. IEEE Press.

[16] S. Han, Y. Dang, S. Ge, D. Zhang, and T. Xie. Performance Debugging in the Large via Mining Millions of Stack Traces. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE'12, pages 145–155, Piscataway, NJ, USA, 2012. IEEE Press.

[17] J. H. Huang and C.-W. Chen. On Performance Measurements of TCP/IP and Its Device Driver. In *Proc. 17th Conference on Local Computer Networks*, pages 568–575, Sept. 1992.

[18] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and Detecting Real-World Performance Bugs. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI'12, pages 77–88, New York, NY, USA, 2012. ACM.

[19] M. Jovic, A. Adamoli, and M. Hauswirth. Catch Me If You Can: Performance Bug Detection in the Wild. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA'11, pages 155–170, New York, NY, USA, 2011. ACM.

[20] A. Kadav, M. J. Renzelmann, and M. M. Swift. Tolerating Hardware Device Failures in Software. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP'09, pages 59–72, New York, NY, USA, 2009. ACM.

[21] A. Kadav and M. M. Swift. Understanding Modern Device Drivers. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 87–98, New York, NY, USA, 2012. ACM.

[22] V. Kuznetsov, V. Chipounov, and G. Candea. Testing Closed-Source Binary Device Drivers with DDT. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 12–12, Berkeley, CA, USA, 2010. USENIX Association.

[23] B. Leslie, P. Chubb, N. Fitzroy-Dale, S. Gtz, C. Gray, L. Macpherson, D. Potts, Y.-T. Shen, K. Elphinstone, and G. Heiser. User-Level Device Drivers: Achieved Performance. *Journal of Computer Science and Technology*, 20(5):654–664, 2005.

[24] J. Li, F. Xie, T. Ball, V. Levin, and C. McGarvey. An Automata-Theoretic Approach to Hardware/Software Co-Verification. In *Proceedings of the 13th International Conference on Fundamental Approaches to Software Engineering*, FASE'10, pages 248–262, Berlin, Heidelberg, 2010. Springer-Verlag.

[25] A. Menon, S. Schubert, and W. Zwaenepoel. TwinDrivers: Semi-Automatic Derivation of Fast and Safe Hypervisor Network Drivers from Guest OS Drivers. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 301–312, New York, NY, USA, 2009. ACM.

[26] L. Ryzhyk, P. Chubb, I. Kuz, and G. Heiser. Dingo: Taming Device Drivers. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys'09, pages 275–288, New York, NY, USA, 2009. ACM.

[27] L. Ryzhyk, P. Chubb, I. Kuz, E. Le Sueur, and G. Heiser. Automatic Device Driver Synthesis with Termite. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP'09, pages 73–86, New York, NY, USA, 2009. ACM.

[28] L. Ryzhyk, J. Keys, B. Mirla, A. Raghunath, M. Vij, and G. Heiser. Improved Device Driver Reliability through Hardware Verification Reuse. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 133–144, New York, NY, USA, 2011. ACM.

[29] L. Ryzhyk, Y. Zhu, and G. Heiser. The Case for Active Device Drivers. In *Proceedings of the 1st ACM Asia-Pacific Workshop on Systems*, APSys'10, pages 25–30, New York, NY, USA, 2010. ACM.

[30] K. Shen, C. Stewart, C. Li, and X. Li. Reference-Driven Performance Anomaly Identification. In *Proceedings of the 11th International Joint Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS'09, pages 85–96, New York, NY, USA, 2009. ACM.

[31] M. F. Spear, T. Roeder, O. Hodson, G. C. Hunt, and S. Levi. Solving the Starting Problem: Device Drivers as Self-Describing Artifacts. *SIGOPS Oper. Syst. Rev.*, 40(4):45–57, Apr. 2006.

[32] K. Srinivas and H. Srinivasan. Summarizing Application Performance from a Components Perspective. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE'13, pages 136–145, New York, NY, USA, 2005. ACM.

[33] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering Device Drivers. *ACM Trans. Comput. Syst.*, 24(4):333–360, Nov. 2006.

[34] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the Reliability of Commodity Operating Systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, SOSP'03, pages 207–222, New York, NY, USA, 2003. ACM.

[35] N. R. Tallent and J. M. Mellor-Crummey. Effective Performance Measurement and Analysis of Multithreaded Applications. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP'09, pages 229–240, New York, NY, USA, 2009. ACM.

[36] N. R. Tallent, J. M. Mellor-Crummey, and A. Porterfield. Analyzing Lock Contention in Multithreaded Applications. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP'10, pages 269–280, New York, NY, USA, 2010. ACM.

[37] L. Wang, H. Zhao, G. Dong, and J. Li. On the Complexity of Finding Emerging Patterns. *Theor. Comput. Sci.*, 335(1):15–27, May 2005.

[38] A. Wert, J. Happe, and L. Happe. Supporting Swift Reaction: Automatically Uncovering Performance Problems by Systematic Experiments. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE'13, pages 552–561, Piscataway, NJ, USA, 2013. IEEE Press.