IntroPerf: Transparent Context-Sensitive Multi-Layer Performance Inference using System Stack Traces

Chung Hwan Kim[†*], Junghwan Rhee[‡], Hui Zhang[‡], Nipun Arora[‡],
Guofei Jiang[‡], Xiangyu Zhang[†], Dongyan Xu[†]
[†]Purdue University and CERIAS, [‡]NEC Laboratories America
[†]{chungkim,xyzhang,dxu}@cs.purdue.edu, [‡]{rhee,huizhang,nipun,gfj}@nec-labs.com

ABSTRACT

Performance bugs are frequently observed in commodity software. While profilers or source code-based tools can be used at development stage where a program is diagnosed in a well-defined environment, many performance bugs survive such a stage and affect production runs. OS kernel-level tracers are commonly used in post-development diagnosis due to their independence from programs and libraries; however, they lack detailed program-specific metrics to reason about performance problems such as function latencies and program contexts. In this paper, we propose a novel performance inference system, called IntroPerf, that generates fine-grained performance information – like that from application profiling tools – transparently by leveraging OS tracers that are widely available in most commodity operating systems. With system stack traces as input, IntroPerf enables transparent context-sensitive performance inference, and diagnoses application performance in a multi-layered scope ranging from user functions to the kernel. Evaluated with various performance bugs in multiple open source software projects, IntroPerf automatically ranks potential internal and external root causes of performance bugs with high accuracy without any prior knowledge about or instrumentation on the subject software. Our results show IntroPerf's effectiveness as a lightweight performance introspection tool for post-development diagnosis.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Measurement techniques

Keywords

Performance inference; stack trace analysis; context-sensitive performance analysis

*Work done during an internship at NEC Laboratories America, Princeton.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGMETRICS'14, June 16–20, 2014, Austin, Texas, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2789-3/14/06 ...$15.00.
http://dx.doi.org/10.1145/2591971.2592008 .

1. INTRODUCTION

Performance diagnosis and optimization are essential to software development life cycle for quality assurance. Existing performance tools such as profilers [3, 6, 7, 19] and compiler-driven systems [17, 22, 34, 33] are extensively used in application development and testing stages to identify inefficient code and diagnose performance problems at fine granularity. Despite these efforts, performance bugs may still escape the development stage, and incur costs and frustration to software users [21].

In a post-development setting, software users investigating performance issues are usually not the developers, who have source code and can debug line by line. Therefore, it is desirable for those users to have a diagnosis tool that will work transparently at the binary level, look into all components in the vertical software layers with a system-wide scope, and pinpoint the component(s) responsible for a performance issue. Furthermore, detailed and context-rich diagnosis reports are always helpful to such users so that they can provide meaningful feedback to the developers hence speeding up the resolution of performance issues.

Commodity software is commonly built on top of many other software components. For instance, the Apache HTTP server in Ubuntu has recursive dependencies on over two hundred packages for execution and over $8,000$ packages to build. Unfortunately, the maintenance of such diverse, inter-dependent components is usually not well coordinated. Various components of different versions are distributed via multiple vendors, and they are integrated and updated by individual users. Such complexity in the maintenance of software components makes the localization of performance anomaly challenging due to the increased chances of unexpected behavior.

OS tracers [1, 2, 15, 27] are commonly used as "Swiss Army Knives" in modern operating systems to diagnose application performance problems. These tools enable deep performance inspection across multiple system layers and allow users to spot the "root cause" software component. Given diverse applications and their complex dependencies on libraries and lower layer components, compatibility with all layers is an important requirement. OS tracers are very effective in this aspect because they are free from the dependencies by operating in the OS kernel – below any application software or library.

On the other hand, the tracers' location in the OS kernel makes them lose track of details of program internals such as function latencies and program contexts, which are very

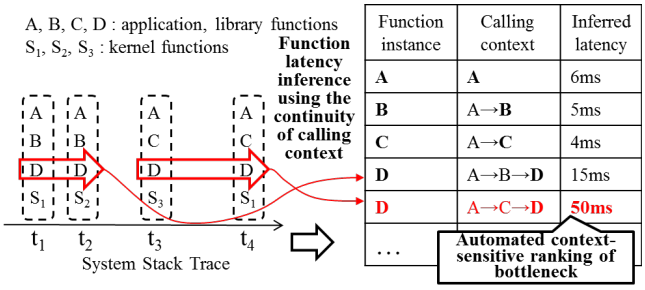

Figure 1: Main idea of IntroPerf: context-sensitive performance diagnosis using inferred latency from system stack traces.

useful to localize root cause functions. These tools collect information on coarse-grained kernel-level events; they do not precisely capture the calls and returns of application functions, the tracing of which is often performed by application profilers or dynamic analysis engines.

Recently, OS tracers provide stack traces generated on OS kernel events [1, 15, 27] to improve the visibility into programs. They cover all system layers from applications to the kernel as shown on the left of Figure 1; therefore, we call the traces *system stack traces*. While they provide improved software performance views, their usage is mostly within event-based performance analysis. For example, Windows Performance Analyzer provides a summary of system performance computed with the weights of program functions' appearances in the system stack traces. An important performance feature missing for performance analysis is *the measurement of application-level function latencies*. Since the system stack events are generated not on the boundaries of function calls but on OS kernel events (e.g., system calls), the timestamps of the events do not accurately reflect how long each function executes.

We propose a novel performance inference system, called IntroPerf, that offers fine-grained performance diagnosis data like those from application profiling tools. IntroPerf works transparently by leveraging the OS tracers widely available in most operating systems. Figure 1 illustrates the main idea. With system stack traces as input, IntroPerf transparently infers context-sensitive performance data of the traced software by measuring the *continuity of calling context*[1] – the continuous period of a function in a stack with the same calling context. The usage of stack traces commonly available from production OS tracers [1] allows IntroPerf to avoid the requirement of source code or modification to application programs while it analyzes detailed performance data of the traced application across all layers. Furthermore, it provides context-driven performance analysis for automating the diagnosis process.

Contributions: The contributions of this paper are summarized as follows.

- **Transparent inference of function latency in multiple layers based on stack traces:** We propose a novel performance inference technique to estimate the latency of program function instances with system stack traces [1] based on the continuity of calling context. This technique essentially converts system stack traces from OS tracers to latencies of function instances to enable fine-grained localization of performance anomalies.
- **Automated localization of internal and external performance bottlenecks via context-sensitive performance analysis across multiple system layers:** IntroPerf localizes performance bottlenecks in a context-sensitive manner by organizing and analyzing the estimated function latencies in a *calling context tree*. IntroPerf's ranking mechanism on performance-annotated call paths automatically highlights potential performance bottlenecks, regardless of whether they are internal or external to the subject programs.

Section 2 presents related work. The design of IntroPerf is presented in Section 3. The implementation of IntroPerf is described in Section 4. Section 5 presents evaluation results. Discussion and future work are presented in Section 6. Section 7 concludes this paper.

2. RELATED WORK

Root cause localization in distributed systems: There is a large body of research work on performance analysis and root cause localization in distributed systems [9, 11, 16, 18, 24, 29, 31, 28]. They typically focus on end-to-end tracing of service transactions and performance analysis on transaction paths across distributed nodes. These nodes are connected through *interactions* such as network connections, remote procedure calls, or interprocess procedure calls. While there is a similarity in the goal of localizing performance anomaly, the performance symptoms in our study do not necessarily involve interactions; instead, they require a finer-grained analysis because the localization target could be any possibly small code. For instance, if a performance anomaly is caused by an unexpectedly extended loop count or a costly sequence of function calls, the related approaches will not be able to localize it because all program execution is seen as a single node without differentiating internal code structures. Therefore, the related approaches are not directly applicable to our performance debugging problem due to their coarse granularity.

Profilers: Profilers [3, 6, 19] are tools for debugging application performance symptoms. Many tools such as `gprof` require source code to embed profiling code into programs. However, in a post-development stage these tools are often not applicable because the subject software may be in binary-only form. Also, user-space profilers may not be able to detect the slowdown of lower layer functions such as system calls due to their limited tracing scope.

Oprofile [6] provides whole system profiling via support from recent Linux kernels. Oprofile samples the program counter during execution using a system timer or hardware performance counters. Oprofile's output is based on a call graph, which is not context-sensitive. In addition, relatively infrequent appearance of lower layer code in the execution may lead to a less accurate capture of program behaviors. In contrast, system stack traces recorded by the OS kernel reliably capture all code layers in a context-sensitive way.

Dynamic binary translators: Dynamic binary translators [23, 26] are commonly used in the research community

[1]Calling context is a sequence of active function invocations as observed in a stack. We interchangeably use *calling context*, *call path*, and *stack trace event* in this paper because we obtain calling contexts from the stack traces.

and some of production profiling tools (e.g., Intel vTune is based on Pin) for binary program analysis. These tools can transparently insert profiling code without requiring source code. However, the significantly high performance overhead makes them suitable mainly for the development stage.

Performance bug analysis: Performance analysis has been an active area in debugging and optimization of application performance. Jin et al. [21] analyzed the characteristics of performance bugs found from bug repositories, and reported new bugs by analyzing similar code patterns across software. This method requires source code that is often not available in a post-development stage. StackMine [20] and DeltaInfer [32] are closely related to IntroPerf in the aspect of using run-time information to detect performance bugs. StackMine focuses on enabling performance debugging by clustering similar call stacks in a large number of reports. StackMine relies on PerfTrack for detecting performance bugs. PerfTrack works by inserting `assert`-like statements in Microsoft's software products for possible performance degradation of functions of interest. This is a very efficient and effective approach for generating warnings when the monitored functions experience a slowdown. However, it requires manual insertion of performance checks in certain locations of the source code that may become a bottleneck.

DeltaInfer analyzes context-sensitive performance bugs [32]. It focuses on workload-dependent performance bottlenecks (WDPB) which are usually caused in loops which incur extended delay depending on workload. IntroPerf is different from DeltaInfer in several aspects. First, it is less intrusive since it avoids source code modification required by DeltaInfer, and thus is suitable for a post-development stage. Second, in code coverage, DeltaInfer mostly focuses on characteristics of the main binary similar to profilers. IntroPerf, however, is able to cover all layers in the localization of root causes; this is due to its input, system-wide stack traces, which include information regarding the main binary, libraries, plug-ins, and the kernel.

Diagnosis with OS tracers: Production OS tracers are commonly used in modern OSes. DTrace [15] is the de facto analysis tool in Solaris and Mac OS X. Linux has a variety of tracing solutions such as LTTng [5], Ftrace [2], Dprobe [25], and SystemTap [27]. Event Tracing for Windows (ETW) is the tracing mechanism in Windows supported by Microsoft [1]. These tools are widely used for diagnosis of system problems. For instance, ETW is used not only by the performance utilities of Windows (e.g., Performance Monitor) but also as an underlying mechanism of other diagnosis tools (e.g., Google Chrome Developer Mode). Stack walking [8] is an advanced feature to collect stack traces on specified OS events. This feature is included in ETW from Windows Vista. Other OS tracers such as DTrace and SystemTap also have similar features. The prototype of IntroPerf is built on top of ETW stack walking, but its mechanism is generic and is applicable to other platforms.

Calling context analysis: Calling context has been used in program optimization and debugging [10, 35]. IntroPerf uses dynamic calling context from stack traces to differentiate function latencies in different contexts and enable context-sensitive performance analysis. Several approaches [12, 13, 30] have been proposed to efficiently encode calling context for various debugging purposes. If combined with OS tracers, these approaches will benefit IntroPerf by simplifying the context indexing process.

3. DESIGN OF INTROPERF

In this section, we present the design rationale of IntroPerf for transparent performance inference across multiple software layers.

Effective diagnosis of performance bugs in post-development stage poses several requirements:

- **RQ1**: Collection of traces using a widely deployed common tracing framework.
- **RQ2**: Application performance analysis at the fine-grained function level with calling context information.
- **RQ3**: Reasonable coverage of program execution captured by system stack traces for performance debugging.

In general, profilers and source code level debugging tools provide high precision and accuracy in analysis at the cost of stronger usage prerequisites such as availability of source code. In a post-development stage, such requirements may not be easily satisfied. Instead, IntroPerf uses system stack traces from OS tracers which are widely deployed for software performance analysis. These tracers operate at the kernel layer without strong dependency on programs or libraries; therefore, IntroPerf can be easily applied as an add-on feature on top of existing tracers without causing extra efforts in instrumenting or recompiling a program.

The metrics of runtime program behavior such as function latency and dynamic calling context have been used to analyze program performance issues [3, 7, 19]. This information is typically collected using instrumentation that captures function calls and returns in profilers and in source code-based approaches. In IntroPerf we aim to obtain fine-grained performance monitoring information without program instrumentation. Instead, such information is inferred using stack traces generated on OS kernel events, which occur in coarser granularity compared to program function calls, leading to lower overhead.

The last requirement in the design of IntroPerf is reasonable coverage of program execution by system stack traces. Our observation is that, even though there are inherent inference errors in the performance analysis results due to the coarse granularity of OS level input events, the accuracy is reasonable for our purpose – performance debugging. This is because performance bottleneck functions with stretched execution time have higher likelihood to appear in call stack samples compared to other functions with short execution time. The inferred results of IntroPerf of such functions hence give high accuracy to analyze the root causes of performance problems. This observation is intuitive and should apply to any sampling-based approach. Our evaluation results (Section 5) support this claim.

3.1 Architecture

The architecture of IntroPerf is shown in Figure 2. The input is system stack traces which are stack traces collected by OS tracers. As briefly described in Section 1, these events do not directly indicate function latency – a useful metric for bottleneck analysis – because the timestamps of the events are for OS events.

To address this challenge, IntroPerf provides transparent inference of application performance, which is the second block from the left in Figure 2. The key idea of our inference

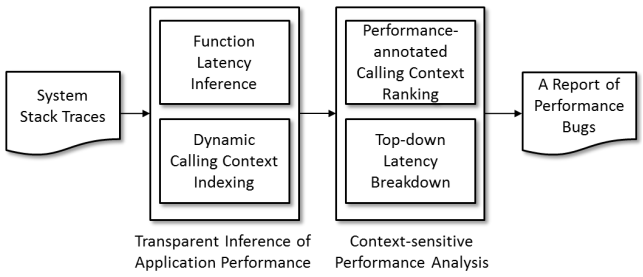

Figure 2: Architecture of IntroPerf

mechanism from stack traces is the *continuity of a function context in the stack*: measuring how long a function's context continues without a change in the stack. Essentially IntroPerf converts system stack traces to a set of function latencies (Section 3.3) along with their calling context information. This context shows a specific sequence of function calls from the "main" function; therefore, it is useful to detect any correlation between performance anomaly and how a function is executed. This context-sensitive performance analysis requires recording and retrieving calling context frequently and efficiently; to do that, IntroPerf has a component for dynamic calling context indexing (Section 3.2).

The third component in Figure 2, context-sensitive performance analysis, determines which functions are performance bottlenecks and in which calling contexts they exist. Our mechanism infers function latency from all layers of the stack in the traces. This leads to potential overlaps of function latencies. The true contributing latency is extracted using the hierarchy of function calls (Section 3.4.1). Our algorithm then ranks the performance of each dynamic calling context and lists the top latency calling context and the top latency functions within each context (Section 3.4.2), which are the list of performance bug candidates generated as the output of IntroPerf.

Next, we will present the underlying functions to support context-sensitive performance analysis and then present our main idea for the inference of function latency.

3.2 Calling Context Tree Construction and Dynamic Calling Context Indexing

Calling context has been widely used in program optimization and debugging [10, 12, 35, 30]. At runtime there are numerous instances of function calls with diverse contexts, forming a distinct order of a function call sequence starting from the "main" function. To reason about the relevance between performance symptoms and calling contexts, it is necessary to associate each function call with its indexed dynamic calling context.

There are several approaches proposed to represent calling context in a unique and concise way [13, 30]. Such approaches require maintaining the context at runtime, and therefore, demand mechanisms to instrument the code and compute the context on the fly. As an offline analysis, we adopt a simple method to represent a dynamic calling context concisely. We use a variant of the calling context tree (CCT) data structure [10]. By assigning a unique number to the pointer going to the end of each path, we index each path with a unique integer ID. Here we define several notations to represent the information.

Algorithm 1 Dynamic Calling Context Indexing

```
   s_i = ⟨f_1, ..., f_k⟩: a call stack      ▷ order: bottom to top
 1: function PathToID(s_i, CCT_w)
 2:     Node *v = &root of CCT_w
 3:     for f in s_i do
 4:         v = getNode(v, f)
 5:     if v.pid == −1 then
 6:         v.pid = path_counter++
 7:         map_path[v.pid] = v
 8:     return v.pid
 9: function getNode(Node* v, function f)
10:     if {∃v'|f == v'.f and v' ∈ v.children} then
11:         return v'
12:     else
13:         v' = new Node(); v'.parent = v; v'.f = f
14:         v.children = v.children ∪ {v'}
15:         v'.id = node_counter++
16:         map_node[v'.id] = v'
17:         return v'
18: function IDtoPath(i, CCT_w)
19:     Node *v = map_path[i]
20:     s = ∅
21:     while *v != root of CCT_w do
22:         s = {v.f} ∪ s
23:         v = v.parent
24:     return s
```

Let a calling context tree be a tree denoted by $\langle F, V, E\rangle$: $F = \{f_1, \ldots, f_m\}$ is a set of functions at multiple layers in a program's execution. $V = \{v_1, \ldots, v_n\}$ is a set of nodes representing functions in different contexts. Note there could be multiple nodes for a function as the function may occur in distinct contexts. $E = \{e_1, \ldots, e_o\} \in V \times V$ is a set of function call edges. An edge e is represented as $\langle v_1, v_2\rangle$ where $v_1, v_2 \in V$. A calling context (i.e., a call path) $p_k = \langle v_1, \ldots, v_k\rangle$ in the CCT is a sequence of function nodes from the root node v_1 to a leaf node v_k. We use the ending function node to uniquely identify a call path.

The input to IntroPerf is a system stack trace $T = \langle(t_1, s_1), \ldots, (t_u, s_u)\rangle$ which is a sequence of OS events with stack traces. Each trace event is represented as a pair: (1) a dynamic calling context $s_i = \langle f_1, \ldots f_k\rangle$ and (2) the timestamp t_i of an OS kernel event (e.g., a system call) which triggers the generation of s_i. `PathToID` in Algorithm 1 presents how to generate the nodes and edges of a dynamic calling context tree for s_i and index it with a unique ID. Each node has several properties: $v.f \in F$ indicates the function that v represents; $v.children$ is a set of edges connecting v and its child nodes; $v.id$ is the node ID number; $v.pid$ is the path ID in case v is a leaf node.

Once a path is created in the CCT, any path having the same context will share the same ID by traversing the same path of the tree. The pair of the unique path ID and the pointer to the leaf is stored in a hash map, `map_path`, for quick retrieval of a path as shown in the `IDToPath` function.

Our context-sensitive performance analysis uses this indexing scheme and the enhanced CCT as the underlying mechanism and data structure to store and efficiently retrieve intermediate analysis results.

3.3 Inference of Function Latencies

In this section, we describe how we infer the latency of function instances using system stack traces which include

Table 1: Inference of function instances of the example shown in Figure 3. I: `isNew`, A: `AllNew`, L: `LastStack`, T: `ThisStack[0]`, R: `Register[0-2]`.

Time	t_1					t_2					t_3					t_4				
Depth	I	A	L	T	R	I	A	L	T	R	I	A	L	T	R	I	A	L	T	R
0	1	1	-	A	(t_1,t_1,A)	0	0	A	A	(t_1,t_2,A)	0	0	A	A	(t_1,t_3,A)	0	0	A	A	(t_1,t_4,A)
1	1	1	-	B	(t_1,t_1,B)	0	0	B	B	(t_1,t_2,B)	1	1	B	C	(t_3,t_3,C)	0	0	C	C	(t_3,t_4,C)
2	1	1	-	D	(t_1,t_1,D)	0	0	D	D	(t_1,t_2,D)	1	1	D	D	(t_3,t_3,D)	-	-	D	-	-

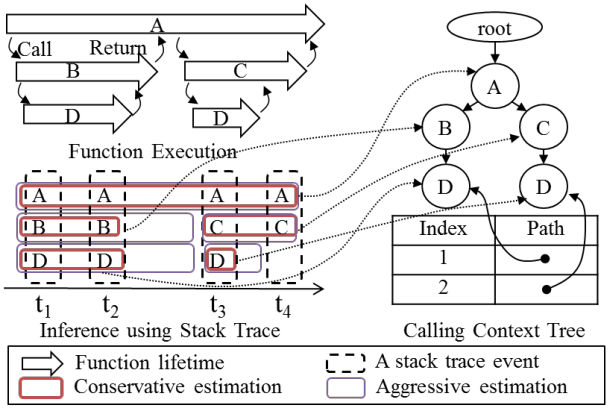

Figure 3: Inference of function latencies across multiple layers using stack trace events.

the function information across the application, intermediate libraries, and the kernel (e.g., system calls).

Figure 3 presents the high-level idea of the inference algorithm 2. The sub-figure at top left shows an example of the latencies between the calls and returns of five function instances: A, B, C, D, and D' (the notation on the second D' indicates its distinct context). Capturing these function call boundaries precisely would require code instrumentation for profilers. Instead, the input to our algorithm is a set of system stack traces which are illustrated as four dotted rectangles at times t_1, t_2, t_3, and t_4.

The algorithm infers the latency of function calls based on the *continuity of a function context*. In Figure 3, we can see that function `A` continues from time t_1 to t_4 without any disruption in its calling context; thus, the algorithm considers this whole period as its latency. On the other hand, function `D` experiences a change in its context. During t_1 and t_2, it was called in the context of `A->B->D`. However, at t_3 its context changes to `A->C->D` leading to discontinuity of its context even though function `D` stays at the same stack depth. Note that this discontinuity propagates towards the top of the stack. If there are any further calls from `D` in this case, their context will be disrupted all together. The algorithm scans the stack trace events in the temporal order and tracks the continuity of each function in the stack frame along with its context.

Our approach infers function latencies in two modes, conservative estimation and aggressive estimation, as illustrated in Figure 3. The conservative mode estimates the end of a function with the last event of the context while the aggressive mode estimates it with the start event of a distinct context. In our analysis, we mainly use the conservative mode. However, we also observe that these two modes are complementary when context change is frequent.

Algorithm 2 Function Call Latency Inference

T : a system stack trace, s_i: a call stack at time t_i

1: **function** INFERFUNCTIONINSTANCES(Trace T, CCT_w)
2: **for** (t_k, s_k) in T **do**
3: $newStackEvent(t_k, s_k, CCT_w)$
4: **for** d in $(0, ..., |Register|)$ **do**
5: $closeRegister(t_{last}, d)$
6: **function** NEWSTACKEVENT(t_k, s_k, CCT_w)
7: initialize $ThisStack, LastStack, Register$
8: $AllNew = 0$, $d = 0$
9: $pid = PathToID(s_k, CCT_w)$
10: **for** f in s_k **do**
11: $isNew = (f == LastStack[d])$
12: **if** $AllNew$==1 **then**
13: $isNew = 1$ ▷ Context change propagates.
14: **if** $isNew$==1 **then**
15: $AllNew = 1$ ▷ Context change triggered.
16: $ThisStack$[d++] $= (f, d, isNew)$
17: **if** $|LastStack| > |ThisStack|$ **then**
18: $i = LastStack[|LastStack| - 1]$
19: **while** $i >= |ThisStack|$ **do**
20: $closeRegister(t_k, i)$
21: remove $LastStack[i$ - - $]$
22: reverse $ThisStack$
23: **for** $(f, d, isNew)$ in $ThisStack$ **do**
24: **if** $isNew == 1$ **then**
25: $closeRegister(t_k, d)$
26: $nid = getNodeIDfromCCT(pid, d, |s_k|, CCT_w)$
27: $Register[d] = [t_k, t_k, f, nid, pid]$
28: **else**
29: $Register[d][1] = t_k$ ▷ The end time stamp
30: $LastStack[d] = f$
31: $t_{last} = t_k$
32: **function** CLOSEREGISTER(t_k, d)
33: $(t_s, t_e, f, nid, pid) = Register[d]$
34: $t_a = t_k - t_s$; $t_c = t_e - t_s$
35: $newFunctionInstance(f, pid, nid, t_a, t_c)$
36: remove $Register[d]$

Now we present how Algorithm 2 processes the illustrated example in detail step by step. The states of variables in each step of the algorithm are illustrated in Table 1. Function `inferFunctionInstances` calls `newStackEvent` for each stack event. This function tracks the continuity of a function context by putting each frame of the stack in a register as far as the context continues. `isNew` is a boolean value showing whether a function newly appears or not (Line 11). Table 1 shows that `isNew` (shown as I) is set to 1 for all stack levels at time t_1 due to their initial appearance. Then at time t_2 the values become 0 due to the identical stack status. The duration of the context continuity of the currently active functions in the stack are tracked using the `Register` array (shown as R in the Table). On a new function, its registration is performed in Line 27. As the function's context continues over time, its duration in `Register` is up-

dated accordingly (Line 29). For instance, three function instances are registered at time t_1 as shown in the R column (6th column in Table 1). At time t_2 the duration of all three functions are updated from t_1 to t_2 (11th column).

If discontinuity of a function context is detected, the contexts of the functions in the higher level of the stack should all be discontinued because they are from a different context. This mechanism is performed by enforcing `isNew`'s status (Line 12-15). Function D appears twice (at times t_2 and t_3) at the same stack depth. But they have different contexts and hence are considered distinct instances.

Function returns are inferred using the moments when the context changes or at the end of the trace. When that happens, the registered functions are closed by calling `closeRegister` (Lines 5, 20, 25). Inside this function the new function instance is created as `newFunctionInstance`.

The inferred function latencies are associated with their contexts by recording the corresponding function node in the calling context tree. When a function is registered, the corresponding CCT node ID (nid) is obtained using the current context (pid) and the depth (d) of the given function in the context (Line 26-27). Later when this function is removed from `Register` and stored as an inferred function instance (Line 32-36), the CCT node ID (nid) is recorded along with the latency.

The inferred latency of a program L is a set of function latency instances ($l \in L$) where $l = \langle f, pid, nid, t_a, t_c \rangle$. t_a and t_c are aggressive and conservative estimations of a function latency, respectively. nid is a function node ID in CCT (when the node is $v \in V$, $v.id = nid$). pid is the ID of the call path that this node belongs to. $f \in F$ is the function ID that this node represents.

We manage the extracted function latency instances in two ways. First, we keep the inferred latencies in a list to enable the examination of individual cases in a timeline for the developers. Second, we aggregate the instances in the CCT so as to have an aggregated view of function latencies accumulated in each context. This is performed using the nid field which can directly point to the corresponding node in the CCT using data structure `map_node` in Algorithm 1. A function node v is extended to have three additional fields: $v.C$ is the number of function counts accumulated in each node; $v.\mu_a$ and $v.\mu_c$ are the average function latencies in the aggressive and conservative modes respectively. Let L' be a subset of L where its estimated function instances belong to a function node, v_j, whose node ID is j, $L' = \{l \mid l \in L,\ l.nid = j\}$. v_j's average function latency is computed as follows.

$$v_j.\mu = \frac{1}{v_j.C} \sum_{l \in L'} l.t,$$

The operations on function latency are applied to both estimation modes, and we will use one expression in the following description for brevity.

We call this extended CCT with the accumulated function latency a *performance-annotated calling context tree (PA-CCT)*. We will use it in the next stage to localize performance bugs.

3.4 Context-Sensitive Analysis of Inferred Performance

IntroPerf provides a holistic and contextual view regarding the performance of all layers of a program by associating the inferred function latencies and the calling contexts available in the stack traces. We use such information to localize the root causes of performance bugs at the function level along with its specific calling context, which is valuable information to understand the bugs and generate patches. In this section we present how to automatically localize a potential performance bottleneck in a PA-CCT that we construct in the previous stage.

To determine the likely root causes of performance symptoms in terms of latency and context, we perform several steps of processing of the inferred function latencies. First, we normalize the function latencies at multiple layers to remove overlaps across call stack layers and extract true contributing latencies in a top down manner. Second, we present a calling context-based ranking mechanism which localizes potential root cause contexts and the bottleneck functions within the contexts automatically.

Algorithm 3 Top-Down Latency Normalization and Differential Context Ranking

```
 1: function LatencyNormalization(Node *v)
 2:     μ_children = 0;
 3:     for v_i ∈ v.children do
 4:         LatencyNormalization(v_i)
 5:         μ_childen += v_i.t_a
 6:     v.μ_own = v.μ − μ_children
 7: function getPathSet(CCT_w)
 8:     return getPath(the root of CCT_w, [], 0)
 9: function getPath(Node *v, p)
10:     P = ∅
11:     if v.children = ∅ then
12:         P = P ∪ {p · v}
13:     else
14:         for v_i ∈ v.children do
15:             P = P ∪ getPath(v_i, p · v)
        return P
16: function DiffRankPaths(N, CCT_base, CCT_buggy)
17:     P_base = getPathSet(CCT_base)
18:     P_buggy = getPathSet(CCT_buggy)
19:     for p' ∈ P_buggy do
20:         if {∃p ∈ P_base | p' ≡ p } then
21:             c = Σ diff μ of equivalent nodes in p' and p
22:         else
23:             c = Σ μ in each node in p'
24:         ΔP.append(c, p')
25:     for (c, p') ∈ ΔP do
26:         Sort all function nodes in p' with regard to v.μ
27:         Annotate the rank of each node v in p' in v.rank
28:     Sort all paths in ΔP with regard to c
29:     return top N paths of ΔP
```

3.4.1 Top-Down Latency Normalization

IntroPerf estimates the latency of all function instances (i.e., duration between their invocations and returns) in the call stack. While this estimation strictly follows the definition of function latency, raw latencies could be misleading for identifying inefficient functions because there are overlaps in the execution time across multiple stack layers as illustrated in Figure 4. With raw latencies, function `A` would be determined as the slowest function because its duration is the longest. Without properly offsetting the latency of child functions, such top level functions (e.g., "main") that stay at the bottom of the call stack would always be considered as expensive functions. Note that this challenge does not oc-

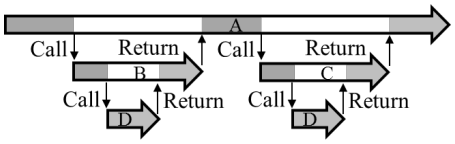

Figure 4: Top-down latency normalization. Non-overlapped latency is shown in shade.

cur in profilers because their approaches are bottom-up by sampling the program counters and always accounting for latency at the low shaded levels in Figure 4.

To remedy this problem, we use the association between callers and callees in the CCT. As shown in Figure 4, the latencies of callees always overlap with the latency of their caller. This relationship can be expressed as the following formula. Let $V' = \{v_i | \langle v_j, v_i \rangle \in v_j.children\}$, and the non-overlapped latency of v be $v.\mu_{own}$:

$$v_j.\mu = v_j.\mu_{own} + \sum_{v_i \in V'} v_i.\mu$$

Based on this observation, we address the above issue by recursively subtracting the latency of callee functions from the caller functions in a PA-CCT. Given the root node of a PA-CCT, function `LatencyNormalization` (Line 1-6 in Algorithm 3) recursively traverses the entire tree and subtracts the sum of the child node latencies from its parent's, leaving the latency truly contributing to the execution of that function.

3.4.2 Performance-Annotated Calling Context Ranking

In this section, we present how IntroPerf automatically localizes the likely cause of performance problems. We use the runtime cost of executed code estimated by the inferred latency from system stack traces to localize the root cause. While this approach is able to localize hot spot symptoms, there is an additional challenge in finding out which code region should be fixed because program semantics need to be considered. For instance, the root cause of high latency could be due to a bug triggered inside a function. On the other hand, it is also possible that the root cause is in other related functions such as a caller of a hot spot function because it introduces the symptom due to its invocation parameters. Our validation case in the evaluation section 5.1 indeed shows that in some applications the final patched functions could be away from the hot spots by a couple of frames in the call stack.

Our approach *ranks hot calling contexts*, which expose closely related functions in the call stack such as callers and callees in addition to the hot function particularly in the context when a high latency is triggered. The invocation relationship in calling contexts allows developers to inspect neighboring functions in the call stack that may have impact on the hot spot function and to find the most suitable code region to patch especially when complex code semantics are involved.

In order to locate the hot spot calling contexts, we generate a set of call paths by traversing a CCT and rank them. Let P_s be a set of paths observed in an execution s. Function `getPathSet` in Algorithm 3 recursively traverses the CCT and generates P_s. As the algorithm moves from a node to its child node, the path is updated by concatenating the function node ($p \cdot v$). When the algorithm reaches a leaf node which does not have any children, it stores the path from the root to the current node in the path set (P_s).

One challenge in hot spot ranking to investigate performance bugs is that some code is inherently CPU intensive and ranked high regardless of workload changes (e.g., crypto, compression functions). While such code region needs careful inspection for optimization, its behavior is not unusual in the developers' perspective because it reflects the characteristics of the program. Performance bugs reported in the community typically describe unexpected symptoms that are triggered in a particular input or workload (e.g., a large input file). In that sense, such originally costly code is less important for our purpose to determine the root cause of a performance bug symptom.

To address this problem and improve the quality of ranking output, we employ a systematic *differential method* in the ranking of calling contexts. The method uses two sets of CCTs produced from the workload samples under two different inputs: a *base* input that a program was tested and confirmed to work as expected with, and another *buggy* input that a user discovered and reported to have a performance bug symptom (e.g., a moderate sized input and a large input). By subtracting the inferred execution summary based on a base input from that based on a buggy input, we prioritize the hot spots sensitive to the buggy input in a higher rank and reduce the significance of the commonly observed hot spots. To apply this technique, we first need to define the equivalence of paths in multiple workloads.

Let there exist two paths which respectively belong to P_1 and P_2 ($p_k = \langle v_1, \ldots, v_k \rangle,\ p_k \in P_1,\ p'_k = \langle v'_1, \ldots, v'_k \rangle,\ p'_k \in P_2$). We define that two paths are equivalent ($p_k \equiv p'_k$) if the represented functions of two paths are identical in the same order. The differential cost of the paths is calculated as follows: P_1 and P_2 are for a base input and a buggy input, respectively. In comparison between the two sets of paths in dynamic calling contexts, a new context may appear due to the buggy input. In such a case, we consider the latency of the base workload as zero and use only the context of the buggy input.

$$\begin{aligned} &\sum_{x=1}^{k} (v'_x.\mu - v_x.\mu) && : \exists p_k,\ \exists p'_k,\ p_k \equiv p'_k \\ &\sum_{x=1}^{k} v'_x.\mu && : Otherwise \end{aligned}$$

The paths are ranked using the above cost formula and the top N ranked functions are listed for evaluation (function `DiffRankPaths` in Algorithm 3). The number of dynamic calling contexts can go up to tens of thousands even though a partial workload is sampled depending on the complexity of workload. Such context ranking significantly reduces analysis efforts by limiting the scope to a small set of *hot calling contexts*. Furthermore, for each hot calling context we provide the ranking of function nodes illustrating *hot functions* inside the path.

Later in Section 5 (Figure 7), we will present a few cases of real world performance bugs with the top-N% hot calling contexts and hot functions in the ranked order in the heatmap (i.e., a color-mapped table with rows for distinct contexts, columns for the functions within a path, and colors for function latencies). It provides a concise and intuitive view on context-sensitive hot spots and assists developers

by automatically narrowing down their focus from massive amount of dynamic calling contexts to a few highly ranked code contexts.

4. IMPLEMENTATION

IntroPerf is built on top of a production tracer, Event Tracing Framework for Windows (ETW) [1] – a tracing facility available on Windows (introduced in Windows 2000) and is used by management tools such as `Resource Monitor`. We use ETW to generate system stack traces, which are the stack dumps generated when kernel events happen, thus including a range of function information from user functions to kernel functions. Stack traces are generated on a selection of kernel events specified as input to the framework as the stack walking mode. We use two configurations: (1) system call events, and (2) system call + context switch events. In addition, we include several miscellaneous events in tracing (e.g., process, thread, and image loading events) without stack walking. These events are included as a common practice to disclose execution status during tracing such as the loaded program images and the memory address of the kernel necessary to understand the trace.

The front-end of IntroPerf parses the ETL (Event Tracing Log, the output of ETW) files and extracts both kernel and user space function information which are sliced for each process and thread. The back-end consists of several stages of programs performing the construction of the calling context tree, inference of function latencies, and ranking of the performance annotated-CCT. The entire framework including both the front-end and back-end has a total of 42K lines of Windows code written in Visual C++. All experiments are done on a machine with Intel Core i5 3.40 GHz CPU, 8GB RAM, and Windows Server 2008 R2.

In the presentation of our evaluation, we use the debugging information in the program database (PDB) format for convenience of illustration and validation of data. However, it is not a requirement for our mechanism because our framework, instead, can represent instructions (e.g., function entries) by their offsets within the binary. As an example usage scenario, IntroPerf can generate detailed bug localization reports on problematic performance symptoms only based on the binary packages on users' sites. If the user determines that a root cause belongs to a software component, he/she can report it to the vendor. The developers on the vendor side should have the debugging information which is stripped in the release. They can interpret the details of IntroPerf's report with function names using symbolic information.

The Visual Studio compiler produces PDB files for both executable files (e.g., `*.exe`) and libraries (e.g., `*.dll`) regardless whether it uses the debug or release mode. Most Microsoft software products (e.g., Internet Explore, Windows kernel, and low level Windows subsystems) provide such information which can be automatically retrieved from the central server with a configuration. Therefore interpreting the software layers of Windows including the kernel and GUI system is straightforward even though Windows is a proprietary OS. Most open source projects make their symbol files available in addition to the binary packages to assist debugging. If not, this information can be easily generated by compiling the code.

Currently ETW does not support the stacks of virtual machines (e.g., Java) or interpreters due to limited parsing capability of non-native stacks. Supplemental stack interpretation layers such as in `jstack` [4] will resolve this problem.

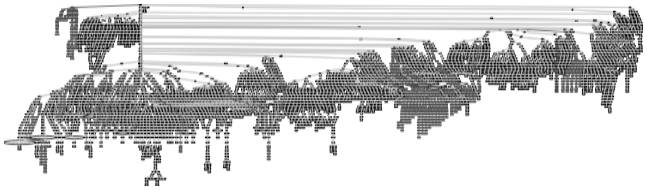

Figure 5: Dynamic calling contexts of the Apache 45464 case showing hundreds of distinct call paths.

5. EVALUATION

In this section, we evaluate several aspects of IntroPerf experimentally. Here are the key questions in the evaluation:

- How effective is IntroPerf at diagnosing performance bugs?
- What is the coverage of program execution captured by system stack traces?
- What is the runtime overhead of IntroPerf?

5.1 Localizing Root Causes of Performance Bugs

IntroPerf enables transparent performance introspection of multiple software layers which includes relative performance cost of individual functions in each specific context. This information is valuable for developers to understand "where" and "how" (in terms of function call sequences) performance bugs occur and eventually to determine the most suitable code regions to be fixed. In this section, we evaluate the effectiveness of IntroPerf in localizing the root causes of performance bugs in a set of open source projects which are actively developed and used.

Evaluation setup: We selected open source projects with various characteristics such as server programs (Apache HTTP server: web server, MySQL: database server), desktop applications (7zip: file compressor/decompressor), and a low-level system utility (ProcessHacker: an advanced version of Task Manager) to highlight generic applicability of IntroPerf. The life span of these projects range from five to eighteen years as of 2014 with continuous development and improvement of code due to popularity and user bases.

Input to experiments: As the input cases to be analyzed, we use the performance bug symptoms that are reported by users. We checked the forums of the projects where users post their complaints on performance issues, and collected cases which include instructions to trigger the performance issues. In addition to the workload described to trigger symptoms, we created another workload with a similar input on a smaller scale as a base input to offset costly code which is not closely relevant to the symptom.

IntroPerf analyzes bug symptoms without any prior knowledge; thus it is capable of identifying the root causes of new bugs. Such a process, however, typically requires non-trivial turn-around time for evaluation by developers and patch generation. While our long-term evaluation plan includes finding un-patched bugs, in this paper, for validation purposes we use the cases whose patches are available to

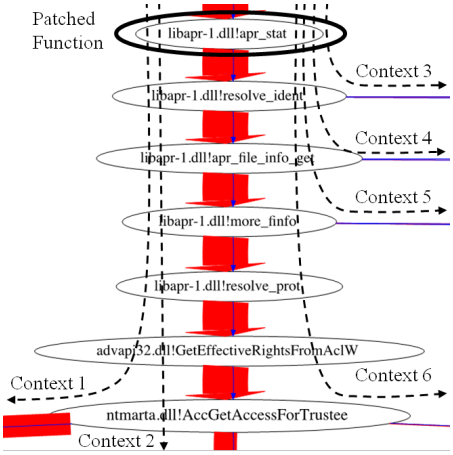

Figure 6: A zoomed-in view of the PA-CCTs of the Apache 45464 case.

compare with IntroPerf's output derived with zero knowledge and evaluate its efficacy.

Performance-annotated calling context tree (PA-CCT): A calling context tree provides a very informative view to developers to understand the root cause by distinguishing the overhead along with distinct call paths. IntroPerf creates this view on all layers by overlaying the estimated function latencies from system stack traces. Figure 5 illustrates this holistic view of a dynamic calling context for the Apache 45464 case with hundreds of distinct call paths. Due to its complexity, we show a zoom-in view in Figure 6 around a costly function call path of Figure 5. Each node represents a function and an edge shows a function call. We use two workloads in a combined view showing the edges from two PA-CCTs. The workloads of a large input and a small input are respectively illustrated as thick red arrows and thin blue arrows. The thickness of the arrows represents the inferred latency showing a clear contrast of the performance of the two workloads. Note that a PA-CCT shows a more informative view than a program CCT due to the distinct calling context in all software component layers.

The details of the PA-CCTs in the evaluated cases are presented in Table 2. Columns 3, 4, 5, and 6 show the runtime program characteristics captured in stack traces: the number of loaded program binaries and libraries ($|I|$), the number of distinct dynamic calling contexts ($|P|$), the total number of functions present in the stack ($|F|$), and the average length of paths (l).

Costly calling contexts and functions: While human inspection of a PA-CCT is useful for analysis on a small scale, for a PA-CCT with non-trivial size, as shown in Figure 5, the amount of details would be overwhelming for a manual approach. Hence IntroPerf provides an automated ranking of costly (i.e., hot) calling contexts. Algorithm 3 ranks call paths using their runtime cost, which is the sum of the function's aggregated inferred latencies.

In this evaluation, we use two metrics: hot calling contexts (i.e., paths) and hot functions within the paths. To validate how effective IntroPerf is, we compare IntroPerf's results with the patch (considered as the ground truth) and present how closely IntroPerf's metrics match the ground truth using two distance metrics. A path distance, p_{min}, represents the similarity between the bottleneck path and the root cause path in the ground truth using the top (minimum) rank of the path that includes a patched function. A function distance, f_{min}, shows the effectiveness of IntroPerf in a finer function level using the minimum number of edges between the most costly function node and a patched function node *within a path*.

The comparison result is presented in columns 7-10 in Table 2. As in-depth information, top ranked costly calling contexts are further illustrated as heat-maps in Figure 7, where each row represents the cost of one path and the top row is the call path (context) with the highest estimated latency. The latency of an individual function is represented in colors (log10 scale) showing the bottlenecks in red and the patched functions (ground truth) are illustrated as circled "P" marks. To ease the identification of a bottleneck within a path, the function with the highest latency is marked as 0 and the adjacent functions are marked with the distance from it.

In all cases, we can confirm that the bottleneck contexts ranked by IntroPerf effectively cover the patched functions from the data shown in Table 2 and illustrated in Figure 7 while such data vary depending on the characteristics of the program and the performance symptoms. For instance, the patch can be placed in a hot function if the bottleneck is in the function. On the other hand, a patch could be placed in its caller if its parameter is anomalous. Note that in both cases IntroPerf offers *an opportunity to correlate the bottleneck and the patch* using calling context performance ranking. It enables developers to easily check the adjacent functions around the bottleneck function in the context and to reason about the suitable program points for patches.

5.2 External Root Causes of Performance Bugs

So far we have presented the cases that the root causes of performance bug symptoms are present *inside* the program. In a post-development stage, software components are installed, integrated, and updated in an un-coordinated manner. If any of the multiple layers has a problem, it will impact the overall performance of the software. Note that this problem is beyond a typical scope of traditional debugging or testing techniques because the analysis requires the capability to inspect multiple software layers that are separately built.

Since IntroPerf uses system stack traces that include information at multiple system layers, IntroPerf is not limited to the analysis of software's internal code but is able to address performance problems external to the program binary as well. In this section, we evaluate potential cases that a dependent but *external* software component, such as a plug-in or an external library, causes performance issues. It shows the unique advantage of IntroPerf: *introspecting all vertical layers and identifying the source of performance impact* automatically without requiring source code.

This type of performance bugs are in general not well studied because the root causes may depend on the integration and deployment of external components including the system environment. This type of performance anomaly tends to be quite common in the field. However, because of currently limited documentation of real cases and time constraints, we choose one of the components on which the target software depends, and manually inject a time lag into

Table 2: Evaluation of IntroPerf on the root cause contexts of real world and injected performance bugs.

Program Name	Bug ID	Program Characteristics				IntroPerf Evaluation				Internal/ External	Ground Truth
		$\|I\|$	$\|P\|$	$\|F\|$	l	p_{min}	f_{min}	Root Cause Binary	Root Cause Function		
Apache	45464	29	319	712	40.96	1	36	`libapr-1.dll`, Internal Library	`apr_stat`	Internal	Patch
MySQL	15811	36	1051	1275	31.22	1	0	`mysql.exe`, Main Binary	`strlen`	Internal	Patch
MySQL	49491	13	144	368	33.71	3	5	`mysqld.exe`, Main Binary	`Item_func_sha::val_str`	Internal	Patch
ProcessHacker	3744	23	2704	1172	49.34	1	0	`ProcessHacker.exe`, Main Binary	`PhSearchMemoryString`	Internal	Patch
7zip	S1	28	1160	1182	72.21	11	16	`7zFM.exe`, Main Binary	`CPanel::RefreshListCtrl`	Internal	Patch
7zip	S2	33	1793	1496	59.61	3	16	`7zFM.exe`, Main Binary	`CPanel::RefreshListCtrl`	Internal	Patch
7zip	S3	22	656	819	58.78	1	15	`7zFM.exe`, Main Binary	`CPanel::Post_Refresh_StatusBar`	Internal	Patch
7zip	S4	30	1002	1274	55.95	2	16	`7zFM.exe`, Main Binary	`CPanel::RefreshListCtrl`	Internal	Patch
ProcessHacker	5424	25	1488	978	40.60	1	54	`ToolStatus.dll`, Plug-in	`MainWndSubclassProc`	External	Patch
ProcessHacker	-	26	1241	906	41.56	1	0	`ToolStatus.dll`, Plug-in	`NcAreaWndSubclassProc`	External	Inject
Internet Explorer	-	92	18716	6168	71.64	1	0	`MotleyFool.dll`, Toolbar Plug-in	`CMFToolbar::GetQuote`	External	Inject
Miranda	-	42	1032	1245	52.40	1	0	`Yahoo.dll`, Plug-in	`upload_file`	External	Inject
Apache	-	14	77	302	26.12	3	0	`mod_log_config.so`, Plug-in	`ap_default_log_writer`	External	Inject
Apache	-	18	96	331	25.89	2	0	`mod_deflate.so`, Plug-in	`deflate_out_filter`	External	Inject
VirtualBox	-	39	1288	1031	39.36	1	0	`QtCore4.dll`, External Library	`QEventDispatcherWin32::processEvents`	External	Inject

it. Injection of the code enables the validation by knowing the ground truth of the fault. In the long term, we expect to study real cases in future work.

Evaluation setup: For evaluation, we selected the cases where the source of the bottleneck is separate from the main software binary. Such external modules are integrated on the deployment site or optionally activated such as independent libraries which the main software is relying on or plug-ins implemented as dynamic libraries. In addition to one real case of ProcessHacker, we have six delay-injection cases: `ToolStatus` is a plug-in for ProcessHacker. `MotleyFool` is a toolbar for Internet Explorer. Miranda is a multi-protocol instant messenger and we selected the file transfer module of the Yahoo messenger plug-in. VirtualBox relies on several libraries for graphics and multimedia. We injected a bottleneck in the Qt library. Also we injected bottlenecks in the `mod_log_config` and `mod_deflate` modules of Apache HTTP server which log and compress user requests respectively.

Ranked costly contexts: The bottom 7 rows of Table 2 show the results of IntroPerf's capability of localizing the external root causes of performance bottlenecks. In all cases, IntroPerf successfully identified the root cause of the injected bottleneck with high accuracy.

5.3 Coverage of System Call Stack Traces

The system stack trace is a collection of snapshots of the program stack sampled on OS kernel events. Therefore, the sampled calling context and function instances will be limited to the frequency of the kernel events (e.g., system calls, context switch). However, we found that this coarse granularity suffices for diagnosis of performance anomalies based on a critical observation that *the quality of this sampled view improves as the latency of a function increases.* This property leads to a larger number of appearances of function contexts and instances in the stack traces and higher accuracy in the inference accordingly. In this section, we will experimentally confirm this hypothesis.

Evaluation method: The necessary information for this experiment is the fine-grained function call instances and the kernel events which trigger the generation of system stack traces. We used a dynamic translator, Pin [23], to capture function calls and returns. System stack traces are generated by taking the snapshots of the stack on kernel events. System calls are captured by Pin because they are triggered by the program. However, context switches are driven by the kernel; hence, they are hard to capture in user mode experiments on Pin. Therefore, such events are simulated by taking stack snapshots for every time quantum in the offline trace analysis.

We evaluate three configurations regarding the kernel events to generate system stack traces: (1) system calls, (2) system calls and low rate context switches, and (3) system calls and high rate context switches. The context switch quantum in Windows systems vary from 20 ms to 120 ms depending on scheduling policies and the configuration of the system [14]. We use 120 ms as the low rate of context switch in the second configuration as a conservative measure. We also evaluate the higher context switch rate (20 ms interval) in the third configuration. These three configurations present the views of stack traces in three sampling rates. Note that the dynamic translation framework is slower than the native execution having an effect similar to executing on a slow machine. Thus we mainly focus on the relative comparison of the three configurations.

We evaluate the coverage of stack traces in two criteria. First, the dynamics of calling context are analyzed to evaluate the diversity of call paths shown in the stack traces. The second criterion is regarding the function call instances that are captured by the stack traces.

Coverage of dynamic calling context: Figures 8 (a)-(c) illustrate the coverage of dynamic calling context based on the three configurations described above. The X-axis shows the order of dynamic calling contexts in percentage sorted by the latency in the ascending order. The graph shows the coverage of dynamic calling contexts for all calling contexts whose ranks are the same or higher than the order shown in the X-axis, and the Y-axis shows the coverage of dynamic calling contexts in percentage. For example, if x is 0%, the corresponding y value shows the coverage of all dynamic calling contexts. If x is 99%, y shows the coverage of calling contexts with the top 1% high latencies.

Each figure shows multiple lines due to MySQL's usage of multiple processes/threads. In the experiment using Pin on Windows, we observed that some threads only execute OS functions in their entire lifetime. We used the processes and threads that include the application code (e.g., "main") in the execution since OS-only behavior could be dependent on platforms (e.g., program runtime setup) and may not be representative application characteristics.

In the overall scope ($x = 0$), the stack traces cover under 20% of dynamic calling contexts. However, the results are improved on high latency contexts which are the focus of IntroPerf for performance diagnosis. The right sides of

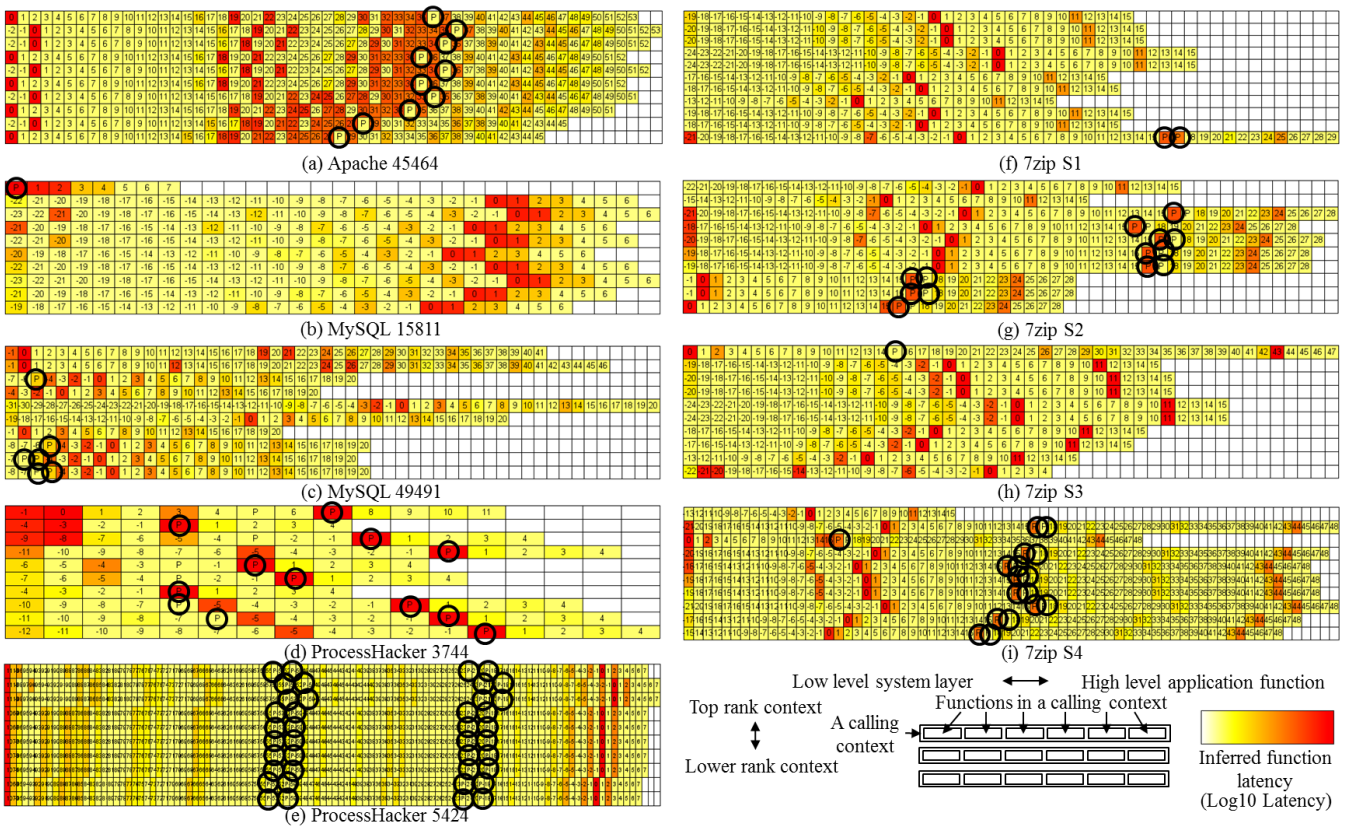

Figure 7: Top ranked contexts of performance bug symptoms. Each row represents a calling context. Columns represent the functions within contexts. The root causes in the ground truth are highlighted with circles.

Figures 8 (a)-(c) show that the coverage increases for top ranked contexts. For the top 1 % slowest functions shown at the right end of the X-axis ($x = 99$), the coverage increases to 34.7%-100% depending on processes/threads.

Coverage of function call instances: Figures 8 (d)-(f) show the coverage of individual function call instances in the stack traces. In all the executions, most threads covered 1.2%-8.05% of the instances. However, for the top 1% high latency functions the coverage increases to 29.1%-100%.

Table 3 summarizes the coverage analysis of dynamic calling contexts and function call instances on three programs: Apache HTTP server, MySQL server, and 7zip. Apache and MySQL are the examples of web and database server programs with I/O oriented workload. 7zip is a data compression/decompression program which would be an example of a desktop application with intensive CPU workload. The table shows different characteristics of these programs. The second column shows the percentage of system calls compared to regular function calls.

Although there are variations in the behavior of the three programs due to their characteristics as shown in columns, the general observation applies in the same way in the comparison between the coverage of the entire scope (columns "Cov. of Context: All," "Cov. of Instances: All") and the coverage of top 1% slowest contexts and latencies (columns "Cov. of Context: Top 1%," "Cov. of Instances: Top 1%"): the coverage of contexts and function instances is significantly higher for high latency functions. In other words, for slow functions experiencing performance bug symptoms, IntroPerf generates higher quality views compared to shorter (time-wise) functions. This is the core reason for IntroPerf's effectiveness in performance bug diagnosis despite the relatively coarse granularity of stack traces.

5.4 Performance

IntroPerf can work with any system stack trace from production event tracers [1, 15, 27], which are already deployed and being used in production systems. IntroPerf is an offline trace analysis engine. The tracing efficiency is entirely determined by the design and implementation of OS kernel tracers. Even though studying the performance of the tracing sessions is not the focus of this work and cannot be generalized due to the differences and evolution of tracing mechanisms, we present the tracing performance that we measured for ETW [1] to provide some insight into the potential overhead of current production tracers.

We present the overhead for the tracing sessions of three benchmarks: the Apache benchmarking tool (`ab`), MySQL benchmark suite (`sql-bench --small-test`), and 7zip benchmark (`7z.exe b`). `Ab` measures the time that an Apache HTTP server takes to handle 10k requests. `Sql-bench` provides a set of sub-benchmarks that measure the wall-clock times of database operations such as SELECT, INSERT, UPDATE, etc. The Apache and MySQL benchmarks are performed in a local network. 7zip has a built-in benchmark that operates with the `b` option which measures the compression and decompression performance with internal data.

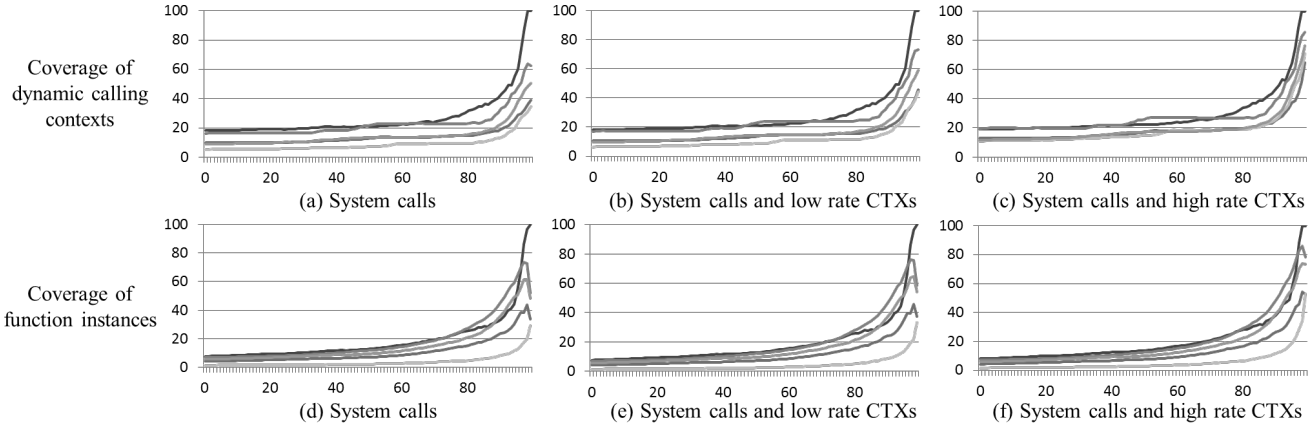

Figure 8: Coverage of dynamic calling contexts (a)-(c) and function instances (d)-(f) of MySQL database.

Table 3: Coverage of dynamic calling context and function instances in multiple sampling schemes of system stack traces in percentage. Ranges are used due to the data of multiple threads. Sys: system calls, LCTX: system calls and low rate context switch events, HCTX: system calls and high rate context switch events.

Program	Syscall	Cov. of Context: All			Cov. of Context: Top 1 %			Cov. of Instances: All			Cov. of Instances: Top 1 %		
Name	Rate	Sys	LCTX	HCTX	Sys	LCTX	HCTX	Sys	LCTX	HCTX	Sys	LCTX	HCTX
Apache	0.33-2.79	18.7-19.3	19.3-20	22.3-24.5	58.6-84.6	61.8-92.3	75.3-100	1.5-20.1	1.5-20.3	1.8-21.6	27-86.7	30.1-93.3	42.6-100
MySQL	0.21-1.48	5.3-18.2	6.4-18.2	10.8-19.2	34.7-100	44.2-100	64.7-100	1.2-7.48	1.3-7.48	1.7-8.05	29.1-100	33.3-100	52.3-100
7zip	0.11-5.03	13.6-47.5	14.5-49.4	17.5-47.5	38.5-100	41-100	50.1-100	0.6-30.2	0.6-30.2	0.8-31.2	16.6-100	18.6-100	26.5-100

Our configuration of ETW flushes the trace records to the disk when the 512 MB kernel buffer is full. The goal of this experiment is to measure the runtime impact due to the generation of stack traces. To minimize the impact due to the flush to disk, we stored the trace on RAM disk. ETW provides another option, on-the-fly trace processing, instead of flushing events to disk. If a custom tracer can be built, it can further lower the overhead.

Figure 9 shows the overhead for tracing OS kernel events along with system stack traces. When system calls and miscellaneous events, which are required to comprehend events described in Section 4, are recorded, the overhead is 1.38%-8.2%. If context switch events are additionally collected, the overhead increases to 2.4%-9.11%. Note that IntroPerf is not required to be active for the entire execution of a program because OS tracers can dynamically start and stop a tracing session without interfering applications. A reasonable scenario would be to sample the abnormal period when the program begins to show it. Moreover, with the efficiency of OS tracers being improved, IntroPerf will be able to perform faster offline analyses.

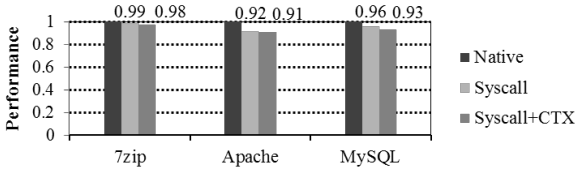

Figure 9: Overhead of ETW stack trace collection.

6. DISCUSSION AND FUTURE WORK

Our evaluation in Section 5 is based on a simple and conservative scheme (i.e., conservative estimation shown in Figure 3) to estimate the latencies of functions. A more relaxed latency estimation scheme (e.g., aggressive estimation or variants) may have either increased or decreased accuracy depending on which scheme closely matches the actual timing of function calls and returns. Improving the estimation accuracy with advanced schemes would be an interesting direction for future work.

Applications heavily relying on system functions may cause high frequency of kernel events. Since the overhead of the OS tracer is likely proportional to the frequency of the events recorded, it would be important to lower the number of recorded events for efficiency as far as the coverage is not significantly affected. Sampling could be leveraged to mitigate the problem. Existing sampling schemes such as those employed by traditional profilers could be similarly effective.

7. CONCLUSION

We present IntroPerf, a performance inference technique that transparently introspects the latency of multiple layers of software in a fine-grained and context-sensitive manner. We tested it on a set of widely used open source software, and both internal and external root causes of real performance bugs and delay-injected cases were automatically localized and confirmed. The results showed the effectiveness and practicality of IntroPerf as a lightweight application performance introspection tool in a post-development stage.

Acknowledgments

We thank Brendan Saltaformaggio and anonymous reviewers who provided valuable feedback for the improvement of this paper.

8. REFERENCES

[1] Event Tracing for Windows (ETW). http://msdn.microsoft.com/en-us/library/windows/desktop/aa363668(v=vs.85).aspx.

[2] Ftrace: Function Tracer. https://www.kernel.org/doc/Documentation/trace/ftrace.txt.

[3] gperftools: Fast, multi-threaded malloc() and nifty performance analysis tools. https://code.google.com/p/gperftools/.

[4] jstack - Stack Trace. http://docs.oracle.com/javase/7/docs/technotes/tools/share/jstack.html.

[5] LTTng: Linux Tracing Toolkit - next generation. http://lttng.org.

[6] Oprofile: a system-wide profiler for linux systems. http://oprofile.sourceforge.net/.

[7] perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/.

[8] Stack Walking (Windows Driver). http://msdn.microsoft.com/en-us/library/windows/desktop/ff191014(v=vs.85).aspx/.

[9] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *SOSP'03*.

[10] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *PLDI'97*.

[11] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling. In *OSDI'04*.

[12] M. D. Bond, G. Z. Baker, and S. Z. Guyer. Breadcrumbs: efficient context sensitivity for dynamic bug detection analyses. In *PLDI'10*.

[13] M. D. Bond and K. S. McKinley. Probabilistic calling context. In *OOPSLA '07*.

[14] M. Buchanan and A. A. Chien. Coordinated thread scheduling for workstation clusters under windows nt. In *Proceedings of the USENIX Windows NT Workshop 1997*.

[15] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *USENIX'04*.

[16] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *DSN'02*.

[17] T. Chilimbi, B. Liblit, K. Mehra, A. Nori, and K. Vaswani. Holmes: Effective statistical debugging via efficient path profiling. In *ICSE'09*.

[18] U. Erlingsson, M. Peinado, S. Peter, and M. Budiu. Fay: extensible distributed tracing from kernels to clusters. In *SOSP '11*.

[19] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: a call graph execution profiler. *SIGPLAN Not.*, 39(4):49–57, Apr. 2004.

[20] S. Han, Y. Dang, S. Ge, D. Zhang, and T. Xie. Performance debugging in the large via mining millions of stack traces. In *ICSE'12*.

[21] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and detecting real-world performance bugs. In *PLDI'12*.

[22] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *PLDI'05*.

[23] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05*.

[24] M. L. Massie, B. N. Chun, and D. E. Culler. The ganglia distributed monitoring system: Design, implementation and experience. In *Parallel Computing*, 2004.

[25] R. J. Moore. A universal dynamic trace for linux and other operating systems. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, 2001.

[26] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI '07*.

[27] V. Prasad, W. Cohen, F. C. Eigler, M. Hunt, J. Keniston, and B. Chen. Locating system problems using dynamic instrumentation. In *Proceedings of the 2005 Ottawa Linux Symposium (OLS)*, 2005.

[28] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: detecting the unexpected in distributed systems. In *NSDI'06*.

[29] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.

[30] W. Sumner, Y. Zheng, D. Weeratunge, and X. Zhang. Precise calling context encoding. *Software Engineering, IEEE Transactions on*, 38(5), 2012.

[31] B. C. Tak, C. Tang, C. Zhang, S. Govindan, B. Urgaonkar, and R. N. Chang. vpath: precise discovery of request processing paths from black-box observations of thread and network activities. In *USENIX '09*.

[32] X. Xiao, S. Han, D. Zhang, and T. Xie. Context-sensitive delta inference for identifying workload-dependent performance bottlenecks. In *ISSTA'13*.

[33] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. Sherlog: Error diagnosis by connecting clues from run-time logs. In *ASPLOS'10*.

[34] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage. Improving software diagnosability via log enhancement. In *ASPLOS'11*.

[35] X. Zhuang, M. J. Serrano, H. W. Cain, and J.-D. Choi. Accurate, efficient, and adaptive calling context profiling. In *PLDI'06*.