PerfGuard: Binary-Centric Application Performance Monitoring in Production Environments

Chung Hwan Kim† Junghwan Rhee‡ Kyu Hyung Lee¶ Xiangyu Zhang† Dongyan Xu†

†Purdue University
West Lafayette, IN, USA
{chungkim, xyzhang, dxu}@cs.purdue.edu

‡NEC Laboratories America
Princeton, NJ, USA
rhee@nec-labs.com

¶University of Georgia
Athens, GA, USA
kyuhlee@cs.uga.edu

ABSTRACT

Diagnosis of performance problems is an essential part of software development and maintenance. This is in particular a challenging problem to be solved in the production environment where only program binaries are available with limited or zero knowledge of the source code. This problem is compounded by the integration with a significant number of third-party software in most large-scale applications. Existing approaches either require source code to embed manually constructed logic to identify performance problems or support a limited scope of applications with prior manual analysis. This paper proposes an automated approach to analyze application binaries and instrument the binary code transparently to inject and apply performance assertions on application transactions. Our evaluation with a set of large-scale application binaries without access to source code discovered 10 publicly known real world performance bugs automatically and shows that PerfGuard introduces very low overhead (less than 3% on Apache and MySQL server) to production systems.

CCS Concepts

•Software and its engineering → Software performance; Software testing and debugging; Software post-development issues;

Keywords

Performance diagnosis, post-development testing

1. INTRODUCTION

Diagnosis and troubleshooting of performance problems is an essential part of software development and maintenance. Traditionally, various performance tools [22, 14, 36, 21, 13, 34] have been extensively used by developers during the development and testing stages in order to identify inefficient code and prevent performance problems. However, unlike other software issues, preventing performance problems before software distribution is challenging [42] for the following reasons. First, modern software has complex dependency on many components developed by multiple parties. For example, an application may have dependency on third-party libraries as well as the system libraries to use the underlying operating system. Therefore, finding the root causes of performance problems requires investigation of the whole software stack of various software component layers [43]. Second, it is very challenging to identify performance issues during the development because software vendors have limited time and environments to test various complex usage scenarios. Consequently, there have been efforts to diagnose performance problems during production deployment, long after the development stage [43, 38, 32, 54].

Production-run performance diagnosis has been performed generally in two major ways, which complement each other and often are used together. First, software vendors maintain bug reporting systems [15, 12, 23]. These systems are used for reporting software issues such as performance and failures issues. Users can voluntarily report the details of their performance issues, for instance, how to reproduce the symptom, the specifications of their system, etc. Second, some software vendors embed code logic to detect unexpected performance delay and to report the incident to the vendors automatically [16]. Specifically such logic monitors the performance of semantically individual operations of a program[1] and raises an alarm if their latency exceeds predetermined thresholds. However, the cost of human efforts to support such logic and thresholds is high due to requirements to perform in-depth analysis on possible application behaviors and to determine the range of its reasonable execution time. In addition, the location to insert the logic needs to be manually determined considering its functionality and run-time impact. Such manual efforts may involve human errors due to the misunderstanding of complex program behaviors, particularly when dealing with large-scale software. Although automating the process could save significant efforts in performance debugging and testing, such feature is not implemented by many software vendors in practice.

Furthermore, software *users* at the deployment stage require performance diagnostics for production software without source code or deep knowledge of the target application. For instance, service providers use open source programs

FSE'16, November 13-19, 2016, Seattle, WA, USA
© 2016 ACM. ISBN 978-1-4503-4218-6/16/11...$15.00
DOI: http://dx.doi.org/10.1145/2950290.2950347

[1] Such operations are also known as application transactions, user transactions [54], units [44], or business transactions [5, 19, 10]. We will use *application transactions* herein.

or 3rd-party software as part of their large infrastructure. Monitoring their performance in the production stage is important due to their impact on the overall quality of the service. However, the lack of source code and code knowledge for instrumentation make the monitoring challenging. There are application performance management (APM) services available in the market [5, 19, 10], but those services require developers to modify the application code for inserting the monitoring API provided by the service or support a limited range of known applications and platforms with prior analysis. Table 1 lists the APM services and existing techniques for application performance diagnosis in comparison. Among the compared approaches, only PerfGuard automatically identifies application transactions and monitors their performance with no code knowledge using only binaries.

To provide a generally applicable, binary-centric framework for performance monitoring, we designed and implemented PerfGuard, which automatically transforms the binaries of an application without its source code or prior code knowledge to monitor its performance at the production run.

PerfGuard performs dynamic analysis of a target application and generates a *performance profile*, which is a "summary" of the application performance behavior. It is used as a "hint" to create a set of monitoring code in the binary format, called *performance guards*, to be inserted into the binaries of the application to diagnose its performance in the production environment. During the production run, the inserted performance guards automatically monitor the performance of specific application transactions that are automatically determined by our analysis. An unexpected performance delay triggers a performance assertion set by the performance guard, and invokes a performance diagnosis to help developers and users resolve the issue.

PerfGuard utilizes *program execution partitioning* (a.k.a. *units* [44]) to automate the recognition of application transactions. Each *unit* is a segment of the execution of a process that handles a specific workload. During the profiling of the application, the units of the identical or similar control flow are clustered into the same *unit type*. PerfGuard determines the time threshold for each unit type during the profiling. A set of performance guards are generated based on these units and embedded into the application binaries. During run-time, the type of each unit is inferred by utilizing the unit's distinct control flow and the run-time stack depth. The performance of each unit execution is examined using the time threshold determined for that unit type.

PerfGuard does not rely on specific types of performance bugs. Instead, it focuses on finding time delays caused by various hardware/software issues and collecting useful footprints for troubleshooting them at the production run. To show the effectiveness of PerfGuard, we have analyzed the binaries of 6 popular applications and accurately detected 10 publicly known real world performance problems without source code or code knowledge. The performance impact of PerfGuard in these applications is less than 3% (§7).

The contributions of this paper are summarized as follows.

- Automated analysis of application behaviors and their performance profile from only binaries based on program execution partitioning.

- Enabling a lightweight and transparent insertion of performance monitoring and diagnostic logic without source code or code knowledge for the production stage.

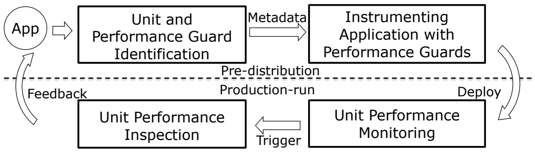

App
Unit and Performance Guard Identification
Metadata
Instrumenting Application with Performance Guards
Pre-distribution
Production-run
Feedback
Deploy
Unit Performance Inspection
Trigger
Unit Performance Monitoring

Figure 1: PerfGuard Architecture.

- Implementation and evaluation of a prototype with various applications in the Microsoft Windows platform showing its practicality discovering performance bugs automatically in the real world software.

§2 presents the design overview of PerfGuard. The key idea of unit-based performance analysis is presented in §3. §4 shows how to identify units and performance guards. Instrumentation of applications with performance guards is presented in §5. Implementation and evaluation of PerfGuard are presented in §6 and §7. §8 and §9 respectively show related work and discussions. §10 concludes this paper.

2. DESIGN OVERVIEW

The overall architecture of PerfGuard is presented in Figure 1. PerfGuard takes the binaries of a target application as an input and automatically discovers application transactions from a set of training runs to produce a performance profile incrementally. The preciseness of the performance profile increases as more training runs are performed. By analyzing the performance profile PerfGuard generates a set of binary code snippets, called *performance guards*, that are injected into the binaries on the discovered application transactions to monitor application performance. While the instrumented application is in the production stage, the performance guards detect potential performance anomalies and inspect the application's state to find their root causes.

To discover application transactions, PerfGuard leverages the fact that a majority of large-scale applications are *event-driven* and incorporate *a small number of loops* in each event handler [44]. For instance, many server applications run "listener" and "worker" loops to receive and handle client requests. Also the applications with a graphical user interface (GUI) are based on loops to respond to user actions. Based on this, PerfGuard partitions the application's execution into *units*, with each unit corresponding to one iteration of such a loop. Intuitively, each unit represents an individual task that the application accomplishes at an event.

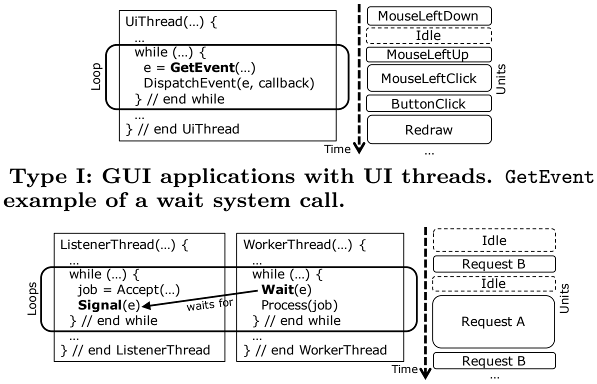
PerfGuard performs profiling on the application to find the code locations to be instrumented for unit performance monitoring. For the classification of units during a production run, PerfGuard clusters units with different calling contexts based on their control flow using a hierarchical clustering algorithm. The distinct calling contexts of each unit type allow PerfGuard to automatically identify which type of unit the application is executing at run-time without code knowledge. The profiling also runs a statistical training to estimate reasonable execution time from which we determine the performance threshold of each unit type.

The inserted performance guards monitor the execution time of units while the application is in the production stage. If the execution time of a unit exceeds the pre-determined threshold of the corresponding unit type, PerfGuard automatically detects it as a potential performance anomaly and triggers an inspection of the unit's execution.

Table 1: The comparison of the approaches for performance diagnosis of native applications. The `App Transaction Performance Aware` column represents whether the approach is aware of the performance of application transactions. The `No Code Knowledge Required` and `No Source Code Required` columns respectively show whether the approach requires code knowledge and the source code. The events at which a program state is collected is shown in the `Monitored Events` column. The `Selection of App Transactions` column shows whether an application transaction is determined automatically or manually.

Category	Name	App Transaction Performance Aware	No Code Knowledge Required	No Source Code Required	Monitored Events	Selection of App Transactions
Tracer (kernel)	Ftrace [13]	✗	✓	✓	System Events	N/A (These approaches are not aware of app transactions.)
	LTTng (kernel tracer) [34]	✗	✓	✓	System Events	
	IntroPerf [43]	✗	✓	✓	System Events	
Profiler	perf [22]	✗	✓	✓	System Events, Periodic Sampling	
	OProfile [21]	✗	✓	✓	Periodic Sampling	
	gperftools [14]	✗	✓	✗	Periodic Sampling	
	Gprof [36]	✗	✓	✗	All Functions	
	Valgrind/Callgrind [50]	✗	✓	✓	All Functions	
Tracer (user)	LTTng (userspace tracer) [34]	✗	✗	✗	Manually Selected Functions	
Embedded Performance Diagnosis	PerfTrack [16]	✓	✗	✗	App Transactions	Manual
Application Performance Management (APM)	AppDynamics* [5]	✓	✗	✗	App Transactions	Manual
	New Relic* [19]	✓	✗	✗	App Transactions	Manual
	Dynatrace* [10]	✓	✗	✗	App Transactions	Manual
APM	**PerfGuard**	✓	✓	✓	**App Transactions (*units*)**	**Automatic**

*We do not consider the known applications and platforms that these APM services support with prior analysis.



```
UiThread(...) {
  ...
  while (...) {
    e = GetEvent(...)
    DispatchEvent(e, callback)
  } // end while
  ...
} // end UiThread
```

(a) Type I: GUI applications with UI threads. `GetEvent` is an example of a wait system call.

```
ListenerThread(...) {
  ...
  while (...) {
    job = Accept(...)
    Signal(e)
  } // end while
  ...
} // end ListenerThread
```

```
WorkerThread(...) {
  ...
  while (...) {
    Wait(e)
    Process(job)
  } // end while
  ...
} // end WorkerThread
```

(b) Type II: Server programs with listener and worker threads. `Signal` and `Wait` are an example of a signal-wait relationship.

Figure 2: Event processing loops and unit execution.

3. UNIT-BASED PERFORMANCE ANALYSIS

PerfGuard achieves performance profiling and monitoring at the granularity of a *unit*, which is a segment of a program's execution that represents a semantically independent operation of the program. Previous work [44] discovered that a wide range of applications are dominated by event processing loops regardless of the platform and the language used. The execution of such an application can be partitioned into units by identifying the event processing loops – one iteration of an event processing loop indicates a unit. PerfGuard leverages *unit-based performance monitoring and analysis* to automatically generate performance guards and help the diagnosis of application performance issues. Unlike previous work [44] that identifies units based on the functional aspects of event processing loops only, our unit identification considers their performance-related aspects.

Figure 2 illustrates the event processing loops of applications in two broad categories, GUI and server applications. A GUI application typically uses an event processing loop in a thread (that is, the UI thread) that responds to the user's actions, as illustrated in Figure 2a. At each iteration of the event processing loop, the thread is blocked by a system call that waits for a user action (e.g., a button click). Commodity operating systems and GUI libraries provide such a *wait system call* (`GetEvent` in Figure 2a), so applications can take user actions as events (e.g. the `GetMessage` Windows API). When the user performs an action, the loop dispatches the event (`DispatchEvent`) to the corresponding callback function to handle the event. After the event is handled (that is, after the execution of the unit), the loop steps to the next iteration and the UI thread waits for another user action.

Server programs, such as Apache HTTP Server and MySQL Server, employ multiple event processing loops across different threads to handle client requests. In Figure 2b, a listener thread and a worker thread are presented as two while loops which have a *signal-wait relationship*. The listener thread receives a client request and sends a signal to a worker thread for dispatching. The worker thread then handles the received request. Once the request is processed, the worker thread is blocked and waits for a new signal from the listener thread. This model is commonly supported by commodity operating systems with signal and wait system calls (e.g., `SetEvent` and `WaitForSingleObject` for Windows, and `kill` and `wait` for the variants of Unix).

We note that the waiting time of an event-driven program (either a GUI or a server application) for a user action or a client request should not be considered as part of the unit execution time as such idle time can be arbitrarily long regardless of performance issues. In addition, once a unit starts execution, any latencies (e.g., thread blocking and context switch time) should be included in the unit execution time since such latencies affect the wall-clock time that the user perceives. Specifically PerfGuard detects the start and the end of each unit execution by instrumenting the wait system calls in the two models. A return from a wait system call in an event processing loop indicates the start of a unit, and an invocation of the wait system call represents its end.

4. IDENTIFICATION OF UNITS AND PERFORMANCE GUARDS

In this section, we present how units are automatically identified and binaries are instrumented for unit performance monitoring and diagnosis.

4.1 Unit Identification

In order to identify units, PerfGuard finds event processing loops through dynamic binary analysis [46] on the training workload of the application. We use dynamic analysis since performance is inherently a property of run-time program behavior and our analysis is based on system call invocation. PerfGuard first identifies all the loops executed by the training [48]. Next, we leverage the event processing models (aforementioned in §3) to identify the event processing loops among all the loops previously found. To be determined as an event processing loop, a loop must either:

1. Invoke a wait system call that receives an external event, or
2. Have a signal-wait relationship with another loop using a certain system call pair.

The entries of the event processing loops and the wait system calls are then instrumented to identify the unit boundaries during profiling and production runs. We use the parameters of the signal and wait system calls as hints to map which thread sends a signal to which waiting thread. If multiple event loops are nested, we use the top level event processing loops taking the largest granularity to cover all nested loops.

4.2 Unit Classification Based on Control Flow

An event processing loop receives different types of events as it iterates. The execution of a unit depends on the type of the event that it handles. For example, a GUI application may receive multiple button click events over time. Depending on which key or button is clicked, the handling unit will have a distinct execution.

To monitor applications while distinguishing such differences without prior knowledge, PerfGuard classifies units based on their control flows with the granularity of a function call and analyzing the calling contexts. The units with the same calling context are classified as the same unit type.

Figure 3 presents the examples of the unit calling contexts of a GUI application and a server application: 7-Zip File Manager (Figure 3a) and Apache HTTP Server (Figure 3b), respectively. During the program's execution, some calling contexts are shared by units of different unit types (illustrated as ovals with no fill). PerfGuard maintains a list of *candidate unit types* of the current context. For example, when the program runs `OnNotifyList` in Figure 3a, the candidate list contains two unit types: `Key press event on file list` and `Redraw event on file list`. We discuss the details of how unit types are recognized at run-time using the list of candidate unit types in §4.4.

4.3 Unit Clustering

In general, program profiling through dynamic analysis could be incomplete. This can cause a unit with an undiscovered calling context to appear during a production run. PerfGuard addresses this issue by mapping a new unit to the closest known unit using a hierarchical clustering of their calling contexts. By grouping units with similar control flows together, a newly observed unit is handled the same as the group with its closest siblings (i.e., most similar control flow and, by extension, time threshold).

Unit Distance Calculation. Given a pair of units X and Y, we first derive a set of call paths P_X and P_Y from the call tree of each unit: $P_X = \{p_1, ..., p_m\}$ and $P_Y = \{q_1, ..., q_n\}$ where a call path $p_k = \{v_1, ..., v_k\}$ is a sequence of function nodes from the root node v_1 to a leaf node v_k. The distance between a call path p_i in P_X and a call path p_j in P_Y is calculated based on the longest common subsequence (LCS) [39] of the two paths. The distance $d(p_i, q_j)$ is defined as

$$d(p_i, q_j) = \frac{\max(|p_i|, |p_j|) - |LCS(p_i, p_j)|}{\max(|p_i|, |p_j|)} \tag{1}$$

We use the lengths of the LCS and the longer call path between p_i and q_j to normalize the distance value. Then, based on the calculated call path distances, the unit distance $D(X, Y)$ is defined as

$$D(P_X, P_Y) = \frac{\sum_{p_i \in P_X} \sum_{q_i \in P_Y} d(p_i, q_j)}{|P_X| \cdot |P_Y|} \tag{2}$$

Algorithm 1 presents the pseudo-code that computes the distance between two units $Unit_A$ and $Unit_B$.

```
function GetPathSet(Tree $t$)
    return GetPath($t.root, \emptyset$)
function GetPath(Node $v,\ p'$)
    $P = \emptyset$
    if $v.children = \emptyset$ then
        $P = P \cup \{p' \cdot v\}$
    else
        for $v_i \in v.children$ do
            $P = P \cup$ GetPath($v_i,\ p' \cdot v$)
    return $P$
function PathDistance($p_i, q_j$)
    $\ell$ = LCS($p_i,\ q_j$)                    ▷ Longest common subsequence
    $b = \max(|p_i|, |q_j|)$
    return $(b - |\ell|)/b$
function UnitDistance($Unit_A,\ Unit_B$)
    $P_A$ = GetPathSet($Unit_A.callTree$)
    $P_B$ = GetPathSet($Unit_B.callTree$)
    $sum = 0$
    for $p_i \in P_A$ do
        for $q_j \in P_B$ do
            $sum = sum +$ PathDistance($p_i,\ q_j$)
    return $sum$ / ($|P_A| \times |P_B|$)
```

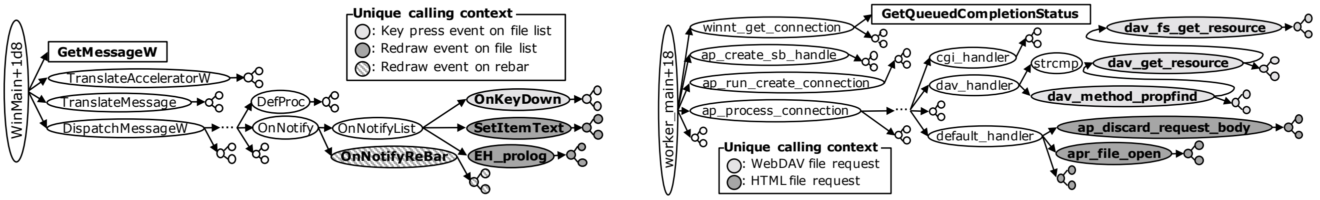
Algorithm 1: Unit distance calculation.

Once the distance between every pair of units is calculated, a hierarchical clustering is performed on the distance matrix; similar units are grouped into the same cluster, and the units in the same group are classified as the same unit type. PerfGuard then identifies the common calling contexts across all units in each cluster to use the distinctness of the calling contexts to represent the unit type. The calling contexts of uncovered units can later be marked and examined by analyzing the call stack at run-time if a performance delay is detected.

4.4 Run-Time Unit Type Inference

PerfGuard inserts performance guards into a set of functions that can monitor distinct units based on calling contexts. However, identifying the type of a unit at an arbitrary point of production run without program semantics is challenging due to the following reasons. First, the current practice for identifying calling context at run-time is examining the full call-stack of a process (a.k.a. stack walking). However, this may incur high performance overhead if it is performed frequently at run-time. Second, given a current context, the type of the current unit cannot be known *a priori*. PerfGuard uses two techniques to handle these challenges with high accuracy.

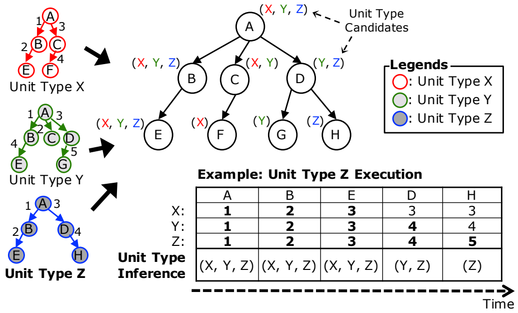
Run-Time Calling Context Identification. If many performance guards are used at run-time, PerfGuard can-



(a) Calling contexts of three unit types of 7-Zip File Manager.

(b) Calling contexts of two unit types of Apache HTTP Server.

Figure 3: Example unit calling contexts observed by PerfGuard. In both graphs, the left-most node represents an event processing loop. The rectangle nodes are wait system calls. Other nodes indicate function calls along with the edges showing the control flows of the units. The nodes in filled ovals are unique calling contexts of the units of distinct unit types. The graphs are simplified for readability.



Example: Unit Type Z Execution

	A	B	E	D	H
X:	**1**	**2**	**3**	3	3
Y:	**1**	**2**	**3**	**4**	4
Z:	**1**	**2**	**3**	**4**	**5**
Unit Type Inference	(X, Y, Z)	(X, Y, Z)	(X, Y, Z)	(Y, Z)	(Z)

Figure 4: Illustration of unit type inference at production run. The unit types with the largest number of occurrences are chosen to be the most likely types of the unit.

not rely on stack walking to understand the current calling context due to high run-time overhead. Instead, PerfGuard infers the calling context by using *the stack frame pointer register* that commodity processors support (e.g., the EBP register in the x86 architecture) to identify the current context efficiently. The function call conventions of modern architectures make sure that the frame pointer register holds the start address of the stack memory region that the current function is using. At the entry of the event processing loop, the stack frame pointer is recorded. Later when an inserted performance guard is executed PerfGuard reads the stack frame pointer again. The difference between the two stack pointers approximates the current calling context. We call the stack pointer difference the *stack depth.* Our evaluation in §7 shows that this approach can identify calling contexts with negligible overhead and has sufficient accuracy to support PerfGuard compared to stack walking.

Unit Type Inference From Context. After identifying the current calling context, PerfGuard must determine the type of the current unit in order to use the corresponding time threshold for detection of performance anomaly. Finding the correct unit type at run-time, however, is not straightforward because the unit type is not conclusive when the calling context is shared by units of different unit types. In order to determine the most probable unit type at runtime, PerfGuard identifies the potential unit types for each calling context and assign them as *unit type candidates* during profiling. At a production run, PerfGuard tracks the occurrences of unit type candidates as the functions are executed in real time. The most likely unit type is inferred using the number of accumulated occurrences.

Figure 4 illustrates how PerfGuard infers the unit type as the program executes. In this example, there are three unit types (X, Y and Z) composed of eight function calls (A-H). Each unit type has different calling context. For example, the unit type Z has the call sequence A-B-E-D-H. The columns of the table show the distinct function calls that the unit executes over time (A in the leftmost column is executed first and H is executed last). Each row (except the bottom-most one) tracks the number of the occurrences of the unit types that PerfGuard has observed. The bottom-most row shows the top unit types in the ranking based on the number of occurrences marked. The unit types with the largest number of occurrences are selected. If more than one unit types become candidates due to the tie of their numbers PerfGuard chooses the one with the smallest time threshold assigned. This allows PerfGuard to detect performance anomalies conservatively. As the execution proceeds, the number of the selected unit types reduces and becomes finally one in the last column after the function H is executed.

We have observed that in the execution of real programs the number of unit type candidates often becomes one at the early stage of unit execution since modern applications typically have many contexts with large depths including library functions which make the unit types unique. Memory overhead is not a significant issue since PerfGuard only has to keep the most recent occurrence numbers (the right-most column of the table) in memory with a fixed number of unit types. In our experiments, the memory overhead was less than 1 MB.

Our unit type inference along with the unit clustering mitigates the incompleteness of dynamic analysis by limiting the impact of newly observed units in production environments. §7.3 shows that our context inference has significantly better scalability with high accuracy in comparison with stack walking.

4.5 Estimating Unit Time Threshold

During the profiling of training runs, PerfGuard records the execution time of each unit and estimates the time threshold of the unit based on time samples. The threshold $\tau(A)$ of a unit type A is calculated based on the series of the time records $R_A = \{r_1, ..., r_n\}$ where $\mu(R_A)$ is the arithmetic mean, $\sigma(R_A)$ is the standard deviation of the time records, and k is a constant:

$$\tau(A) = \mu(R_A) + k \cdot \sigma(R_A) \quad (3)$$

```
$p$: address of intercepted code, $f$: stack frame pointer
$S$: thread local storage, $I$: input performance profile
function onLoopEntry($p, f$)
    $S.loop = I$.findLoop($p$)                          ▷ Set current loop
    $S.loop.f = f$                          ▷ Remember stack frame pointer
function onUnitStart                            ▷ Wait system call return
    if $S.loop \neq \phi$ then
        initialize $S.unit$
        $S.unit.types = \emptyset$
        $S.unit.t_{start}$ = getTimestamp()
function onUnitContext($p, f$)
    if $S.unit \neq \phi$ then
        $t_{end}$ = getTimestamp()
        $d = |S.loop.f - f|$                                  ▷ Stack depth
        $id = (p, d)$                                          ▷ Context ID
        $c = S.loop$.findContext($id$)
        for $u_i \in c.unitTypes$ do
            $S.unit.types[u_i].hits++$
        $U' = \emptyset$
        for $u_i \in S.unit.types$ do
            if $u_i.hits == \max(S.unit.types.hits)$ then
                $U'$.append($u_i$)                 ▷ Top unit type candidates
        $u' = \phi$
        for $u_j \in U'$ do
            if $u_j.t_{threshold} == \min(U'.t_{threshold})$ then
                $u' = u_j$
        $t_{elapsed} = t_{end} - S.t_{start}$
        if $t_{elapsed} > u'.t_{threshold}$ then          ▷ Performance check
            Assert(...)                                        ▷ Inspection
        $t_{check}$ = getTimestamp() $- t_{end}$
        $S.t_{start} = S.t_{start} + t_{check}$
```

Algorithm 2: Performance guards.

k is determined depending on the amount of performance variance that the developer would like to allow at the production. During production runs, the performance of the units are checked using the thresholds in the performance guards.

4.6 Performance Guard Generation

After the performance profiling, PerfGuard generates a set of performance guards which are procedure calls to our shared library functions. Algorithm 2 presents the three library functions for the performance guards. One of the functions is chosen for each performance guard based on the code where the performance guard will be inserted into.

- `onLoopEntry` is called on the entry of an event processing loop. When a program starts and a thread enters an event processing loop, this function records the stack frame pointer in the thread's local storage. PerfGuard begins to monitor the performance of the unit when the thread finishes waiting for an event.
- `onUnitStart` intercepts the return from a wait system call. It records the time when a wait system call returns.
- `onUnitContext` is invoked at the interception of the calling contexts instrumented for performance checks. It infers the type of the current unit (§4.4) and performs an inspection (the `Assert`) if it violates the predefined threshold. The inspection algorithm can be determined by the administrators or developers depending on their needs (e.g., a memory dump). In our evaluation, we examine the thread's call stack as one case to demonstrate the effectiveness of PerfGuard. Call stack traces provide a wealth of information for troubleshooting performance issues since they provide not only the function executed at the anomaly but also the sequence of caller functions leading to the function.

```
* PG_for_X (target, stackframe) {          Foo (a, b) {
*   <Save registers>                          ...
*   <Set target and stackframe>      +        CALL PG_for_X
*   <Save error number>                       instruction X
*   ;; Do performance check                   ...
*   <Restore error number>                  }
*   <Restore registers>
*   return
* }
```

Figure 5: Code instrumentation for a performance guard (abstract code). A performance guard and the patch applied to the program code are tagged with '*' and '+' respectively.

5. INSTRUMENTATION OF APPLICATIONS WITH PERFORMANCE GUARDS

This section presents how performance guards are inserted into a program. Inserting new code into a binary program is in general not trivial due to the constraints in the binary structure such as the dependencies across instructions. PerfGuard has following requirements in functionalities.

Instrumenting Arbitrary Instructions. First, instrumenting arbitrary instructions is important to support any type of application code. In order to provide this characteristic, PerfGuard should be able to insert performance guards in any position of code. Our technique is based on Detours [40] which replaces target instructions with a jump instruction and executes the original instructions after the inserted code (i.e., trampoline). However, Detours assumes the replaced instruction block is a subset of a basic block; thus, it does not allow the program to jump into the middle of the instruction block. To solve this challenge we use `NOP` insertion and a binary stretching technique [33]. PerfGuard inserts `NOP` instructions before every instruction instrumented in the binary, and replaces each `NOP` instruction block with a `CALL` instruction when the program is loaded. PerfGuard generates performance guards in the form of a shared library and modifies the import table of the main executable, so it can be loaded by the OS's loader.

Low Run-Time Overhead. Second, keeping low overhead is required so that PerfGuard can be deployed in production environments. The overhead depends on the number of functions that are instrumented and how frequently the instrumented functions are executed. To keep the overhead minimal, PerfGuard instruments a subset of the functions that are effective for unit-based performance diagnosis. In our prototype implementation, we instrument the functions that are shared by the least number of units of distinct types. This is the key mechanism that enables PerfGuard to minimize the number of functions to instrument while maintaining high accuracy in the unit type inference. Later in §7, we show that PerfGuard achieves very low overhead.

Side-Effect Free. Lastly, performance guards should not interfere with the original application code. Therefore, any code in performance guards should execute so that it can record and recover the original program state respectively before and after performance guard's execution. PerfGuard preserves the application's original state by saving the execution state in memory before a performance check and restoring it after the execution of a performance guard as illustrated in Figure 5. PerfGuard also preserves the stack memory layout for compatibility since it can be used during performance inspection.

6. IMPLEMENTATION

The design of PerfGuard is general to be applied to any OS such as Windows, Unix, Linux, and Mac OS. For our evaluation, we implemented PerfGuard on Microsoft Windows due to its popularity in enterprise environments and a wide variety of closed source software that PerfGuard can be applied due to its binary-centric design.

The identification of units requires the fine grained inspection of code execution to monitor run-time instructions. We used the Pin dynamic instrumentation platform [46] for this feature. Also, we instrumented the Windows APIs in Table 2 that represent signal and wait system calls to identify units. Our evaluation shows that these cover all the applications we have tested.

Table 2: Windows APIs instrumented for unit identification.

Wait	Signal-Wait
`GetMessage`	`PostQueuedCompletionStatus-GetQueuedCompletionStatus`
`ReadConsole`	`SetEvent-WaitForSingleObject`
`accept`	`ResetEvent-WaitForSingleObject`
`recv`	`PulseEvent-WaitForSingleObject`

In order to retrieve high resolution time stamps for performance checks, we used CPU performance counters via the `QueryPerformanceCounter` Windows API [24], which is provided in Windows XP and later.

When calculating the time threshold of a unit type, we use $k = 4$ for our experiments based on the *three-sigma rule* [53]. With $k = 4$ a value falls within the range of the standard deviation with the probability of 99.993%, so the false alarm rate is 0.007% statistically. Existing work [54] also leverages a similar approach with $k = 3$.

For the insertion of performance guards into programs, we used an extended version of Detours [40] which we significantly improved to support instrumentation of arbitrary instructions. BISTRO [33] is used to stretch the application binaries and insert `NOP`s before instrumentation. To inspect the call stack on a performance anomaly, we used the `CaptureStackBackTrace` Windows API [8].

7. EVALUATION

In this section, we evaluate several aspects of PerfGuard experimentally. All experiments are performed on a machine with an Intel Core i5 3.40 GHz CPU and 8 GB RAM running Windows Server 2008 R2. We focus on answering the following key questions in our evaluation:

- How successfully can PerfGuard diagnose real world performance problems?
- How effective is unit clustering based on control flow in performance monitoring?
- How robust and accurate is the run-time context inference?
- What is the performance overhead of PerfGuard?

7.1 Diagnosing Real World Performance Bugs

PerfGuard enables efficient performance bug diagnosis in production environments. Once the performance guards are inserted into a program, PerfGuard monitors the units of the program at run-time. If any unit type has longer execution time than its threshold, it is reported along with its call stack. We chose 6 popular Windows applications to evaluate PerfGuard on various event-based programs: servers (Apache HTTP Server and MySQL Server), text-based clients (MySQL Client), and GUI programs (7-Zip File Manager, Notepad++, and ProcessHacker). After studying over 200 bug reports for those 6 applications, we collected 10 performance bugs caused by diverse root causes: incorrect use of APIs, unexpectedly large inputs, and poor design.

Table 3 shows the performance bugs detected by PerfGuard. The first two columns show the program name and the identification of this bug. The following five columns describe the characteristics of the unit type where the performance bug is detected. $|UC|$, $|P|$, and $|F|$ respectively show the number of distinct call trees, the average number of call paths, and the average number of functions in the unit type. $|PG|$ is the number of performance guards inserted into the program to detect the bug. t is the time threshold in milliseconds for the unit type.

Figure 6 shows the example traces that PerfGuard generates at the detection of increased latencies caused by three performance bugs (Apache 45464, MySQL 15811, and 7-Zip S3) in Table 3. When a performance bug is detected, PerfGuard takes a snapshot of the call stack. Note that, the call stack at the time of root cause and the call stack at the time of bug detection share part of the call stacks in common, but there may be minor differences because the program could have called or returned from several functions since the problematic logic was executed. The gap between the time of bug (located manually) and the time of PerfGuard's detection is calculated as the difference between their respective call stacks, shown as d in Table 3 and Figure 6. Essentially this number may represent the developer's manual effort to find a root cause from the detection point, and we aim to find a balance between minimizing d and adding too many performance guards which will increase the run-time overhead.

The two columns, Root Cause Binary and Root Cause Function, show the root cause of performance bugs, which are manually determined from their bug reports and resolutions maintained in the software's bug repositories. We note that this information is collected only for the evaluation purposes as the ground truth and PerfGuard does not need nor have access to such information.

In all cases, the performance bugs are correctly detected along with the specific details on the buggy unit types. The comparison between the stack on detection and the root cause shows that the call stack distance (d) was 1-8 (i.e., only 1 to 8 functions away). After examining just a few functions in the call stack provided by PerfGuard, developers will be able to easily identify the bug's root cause. In addition to the 6 applications that are open source, we successfully analyzed and found units from a set of proprietary software shown in Table 4.

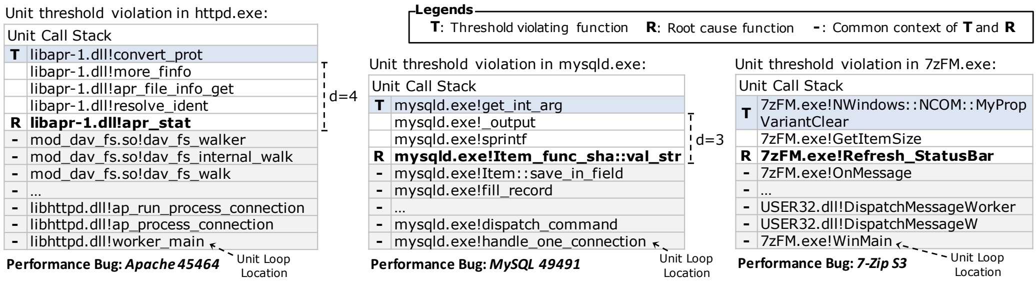
Table 4: Unit identification of proprietary software.

Program Name	Unit Loop Binary
Acrobat Reader	`AcroRd32.dll`, Internal Library
Visual Studio 2008	`msenv.dll`, Internal Library
Windows Live Mail	`msmail.dll`, Internal Library
Evernote	`Evernote.exe`, Main Binary
Calculator	`calc.exe`, Main Binary
WordPad	`mfc42u.dll`, External Library
Paint	`mfc42u.dll`, External Library

The `Unit Loop Binary` column shows the binary where the unit loop is detected. We do not use these applications in the evaluation due to the lack of ground truth and space constraints.

Table 3: Evaluation of PerfGuard on the root cause contexts of real world performance bugs.

Program	Bug	Unit Type Characteristics					PerfGuard Evaluation		
Name	ID	$\|UC\|$	$\|P\|$	$\|F\|$	$\|PG\|$	t	d	Root Cause Binary	Root Cause Function
Apache	45464 [7]	8	17423	635	138	9944	4	`libapr-1.dll`, Internal Library	`apr_stat`
MySQL Client	15811 [17]	24	255126	106	13	997	8	`mysql.exe`, Main Binary	`my_strcasecmp_mb`
MySQL Server	49491 [18]	8	270454	980	303	2079	3	`mysqld.exe`, Main Binary	`Item_func_sha::val_str`
7-Zip FM	S1 [3]	3	30503	140	115	122	3	`7zFM.exe`, Main Binary	`RefreshListCtrl`
7-Zip FM	S2 [2]	2	27922	139	127	109	1	`7zFM.exe`, Main Binary	`RefreshListCtrl`
7-Zip FM	S3 [4]	3	4041	65	15	110	2	`7zFM.exe`, Main Binary	`Refresh_StatusBar`
7-Zip FM	S4 [1]	6	26842	143	120	101	3	`7zFM.exe`, Main Binary	`RefreshListCtrl`
Notepad++	2909745 [20]	16	352831	711	370	6797	6	`notepad++.exe`, Main Binary	`ScintillaEditView::runMarkers`
ProcessHacker	3744 [9]	1	47910	86	23	3104	4	`ProcessHacker.exe`, Main Binary	`PhAllocateForMemorySearch`
ProcessHacker	5424 [25]	32	62136	69	19	10	5	`ToolStatus.dll`, Plug-in	`ProcessTreeFilterCallback`


Legends
T: Threshold violating function
R: Root cause function
-: Common context of T and R
Unit threshold violation in httpd.exe:
d=4
Unit Loop Location
Performance Bug: Apache 45464
Unit threshold violation in mysqld.exe:
d=3
Unit Loop Location
Performance Bug: MySQL 49491
Unit threshold violation in 7zFM.exe:
d=2
Unit Loop Location
Performance Bug: 7-Zip S3

	Unit Call Stack
T	libapr-1.dll!convert_prot
	libapr-1.dll!more_finfo
	libapr-1.dll!apr_file_info_get
	libapr-1.dll!resolve_ident
R	**libapr-1.dll!apr_stat**
-	mod_dav_fs.so!dav_fs_walker
-	mod_dav_fs.so!dav_fs_internal_walk
-	mod_dav_fs.so!dav_fs_walk
-	...
-	libhttpd.dll!ap_run_process_connection
-	libhttpd.dll!ap_process_connection
-	libhttpd.dll!worker_main

	Unit Call Stack
T	mysqld.exe!get_int_arg
	mysqld.exe!_output
	mysqld.exe!sprintf
R	**mysqld.exe!Item_func_sha::val_str**
-	mysqld.exe!Item::save_in_field
-	mysqld.exe!fill_record
-	...
-	mysqld.exe!dispatch_command
-	mysqld.exe!handle_one_connection

	Unit Call Stack
T	7zFM.exe!NWindows::NCOM::MyProp VariantClear
	7zFM.exe!GetItemSize
R	**7zFM.exe!Refresh_StatusBar**
-	7zFM.exe!OnMessage
-	...
-	USER32.dll!DispatchMessageWorker
-	USER32.dll!DispatchMessageW
-	7zFM.exe!WinMain

Figure 6: Sample traces automatically generated by PerfGuard for three performance bugs in Table 3.

Table 5: Performance of top 10 costly unit types. μ is the mean in μsecond. c is the performance variance in percentage where $c = \sigma/\mu \times 100$ and σ is the standard deviation.

(a) Apache HTTP Server.

Rank	μ	c
1	839	1.02
2	695	5.3
3	540	20.4
4	517	2.3
5	511	1.4
6	507	21.0
7	488	27.9
8	467	5.8
9	462	21.9
10	439	8.1
		Avg. 11.52

(b) 7-Zip File Manager.

Rank	μ	c
1	1325005	36.1
2	1208225	7.5
3	1008056	4.6
4	952741	6.6
5	703649	10.9
6	96492	3.4
7	89088	17.8
8	81647	7.4
9	80450	3.5
10	79584	11.1
		Avg. 10.89

7.2 Performance Distribution of Clustered Units

PerfGuard monitors application performance by recognizing units, which are clustered into unit types based on control flow automatically. In this section, we evaluate the effectiveness of our unit clustering. We assume that the same type of units have a consistent execution behavior and thus have similar execution time. To show that our assumption is valid in real-world applications, we measured the means and variances of unit execution time in Apache HTTP Server and 7-Zip File Manager, which represent server programs and GUI applications, respectively. A set of performance guards were created and inserted into each application binary in order to record the unit time during the program execution. Table 5 shows this data for the applications. Each row represents the performance statistics of one unit type, which is a group of units clustered by the similarity of control flow. Relative standard deviation in percentage (c) is used to show the performance variance.

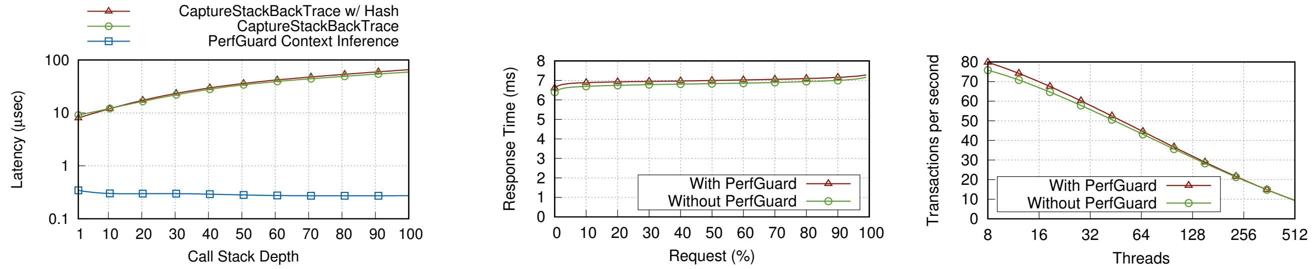
To generate realistic workloads for Apache HTTP Server, we used the web pages provided by Bootstrap [6]. These pages have various sizes and contents. ApacheBench (`ab`) is used to request each of the 20 Bootstrap examples 10,000 times with the concurrency of 4 threads. This yielded a much larger number of units, 1,000,010, but due to the similarity among them (e.g., similar page fetches) these units were clustered into only 26 unit types.

For 7-Zip File Manager, we performed the standard set of UI actions that users would take. Specifically, we navigated through various directories using the file manager, created and deleted files, compressed and uncompressed 10-100 files, and clicked on the menu and tool bar items. 40598 units were recognized and they were categorized into 358 unit types.

In both cases, the performance deviation of the top 10 most costly unit types is about 11 percent (11.52% for Apache HTTP Server and 10.89% for 7-Zip File Manager). Note that the variation is quite low since performance bugs typically incur with more than 11 percent of latency difference. This result shows that the automated clustering based on control flow accurately captures similar behaviors exhibiting similar run-time latencies. We find that the performance deviation of each unit type is relatively small despite the varying input workload because the variation in workload (both in size and content) likely alters the program's control flow. We observed that those performance deviations also come from other factors, such as underlying libraries and system calls, that contribute to the variation of unit performance.

7.3 Run-Time Context Inference

Overhead of Run-Time Context Inference. In this experiment, we measure the overhead of our run-time context inference, the key mechanism used in unit type inference. Figure 7a shows the overhead of Windows APIs (`CaptureStackBackTrace`) versus our approach using different call stack sizes. Traditional stack-walk approaches cause non-negligible overhead due to traversing all stack frames to infer the unit type. Moreover, the overhead significantly increases as the call stack grows. In contrast, our mechanism is much faster and is not affected by the size of the call stack. Such scalability is achieved because our context inference only has to read a hardware register regardless of the depth of the call stack. In our experiments, an average call stack size at the instrumented functions is 23.6 and Figure 7a shows that our approach (0.284 μsec) is 64 times



(a) Overhead of run-time context inference (log-scale): Windows API `CaptureStackBackTrace` and PerfGuard.

(b) Performance of Apache HTTP Server with and without PerfGuard.

(c) Performance of MySQL Server with and without PerfGuard.

Figure 7: Performance evaluation of PerfGuard.

faster than Windows stack walking (18.11 μsec with hashing and 17.4 μsec without hashing).

Accuracy of Run-Time Context Inference. In this experiment, we study the accuracy of run-time context inference technique. We choose one server (Apache HTTP Server) and one client application (MySQL Client). We execute each application with a set of training inputs and record a calling context and a stack depth for each function call.

For a given function and stack depth, if there exists only one calling context, we can correctly infer that context at run-time. However, if there exists more than one calling context for the stack depth, we cannot uniquely indicate the calling context. Our approach was able to correctly identify the calling context from 93.91% of function calls in Apache and 99.06% of function calls in MySQL. In other words, the stack depth uniquely indicates the call stack 93.93% of the time in Apache, and 99.06% in MySQL. Note that even though we fail to uniquely infer the calling context for a few cases, it does not mean that PerfGuard will fail to identify the corresponding unit type. Our unit type inference technique (discussed in §4.4) still may identify the correct unit types as the program executes the following functions.

The result in this experiment shows our technique is significantly faster than traditional stack-walk based technique while we can accurately infer the calling context at run-time.

7.4 Performance Overhead of PerfGuard

PerfGuard is designed to be used in the production stage, so its run-time overhead is a critical evaluation metric. We present this data in the following two sub-categories.

Microbenchmarks on Performance Guards. A major source of performance overhead of PerfGuard is performance guard components. Thus, it is important to keep the overhead of performance guards as low as possible. Table 6 shows the latency of the individual performance guard functions.

Table 6: Latency of performance guards.

Performance guard	Latency (μseconds)
`onLoopEntry`	3.52
`onUnitStart`	1.31
`onUnitContext`	4.37

The latency is measured from 10,000 executions of a performance guard component and the table shows the aggregated time. In all three cases, the latency of a single performance guard execution is less than 0.5 nanosecond.

Overall Impact on Application Performance. Figure 7b and Figure 7c show the performance impact of PerfGuard on Apache HTTP Server and MySQL Server respectively. The two server programs are selected due to the availability of their benchmarking tools.

In Figure 7b, the X-axis is the number of requests sent in percentage and the Y-axis is the response time. We used ApacheBench (`ab`) to generate 10,000 web requests. Figure 7c shows the number of completed transactions per second by MySQL Server with an increasing number of threads (8∼512). SysBench is used for the benchmarking to query a table with 10,000 rows concurrently.

200 functions of Apache HTTP Server and 100 functions of MySQL Server are instrumented by PerfGuard as described in §5 to show the overhead. The results show the overhead by PerfGuard on two applications is less than 3%.

8. RELATED WORK

Execution Partitioning. BEEP [44] derives the units of program execution using loop iterations. We leverage the approach of BEEP in the Windows platform to identify application execution units. Unlike BEEP, which is designed to detect the inter-dependency of units for security analysis, PerfGuard identifies units based on temporal loop relationship and focuses on the classification of units with similar control flow for performance diagnosis.

Performance Debugging in Production. PerfTrack [16] provides a performance debugging feature used in Microsoft's software. This system manually inserts hooks into the key functions of applications to measure their run-time performance and report any incident when their execution time surpasses internal thresholds. This approach is effective due to developers' domain knowledge regarding which code represents the key functions and how long their execution should be. Its downside is that it requires the source code and the application knowledge, which may not be readily available, and the manual efforts for the threshold determination and source instrumentation. In this paper we aim to enable a general functionality similar to PerfTrack applicable to any software without source code or domain knowledge.

Several approaches use call stacks to investigate performance problems. IntroPerf [43] infers the performance of individual functions in the call stacks captured by a system event tracer (e.g. ETW [11]). PerfGuard determines the key application workload using unit type identification and the observation of its run-time performance is done by instrumented code. Therefore it achieves a more efficient and focused monitoring compared to sampling based IntroPerf.

There exist techniques that use information from OS reports for performance and/or reliability issues. In particular, Microsoft has a system called Windows Error Reporting

(WER) [26] which collects call stacks from numerous Windows machines. StackMine [38] models CPU consumption bugs and "wait bugs" based on clustered call stack patterns from call stack mining which allows to identify common root causes from diverse call stacks in different configurations and deployments. Bartz et al. [27] proposed a machine learning based scheme to compute call stack edit distance from failure reports providing a probabilistic similarity metric for failure parts. As shown in these approaches, in order to understand the relevance of code to the root cause of a bug, call stack is a key structure to determine its context becoming the index of code execution. A general utility function (e.g., `malloc`) can be used by multiple functions and depending on its caller its execution may show corresponding behavior. To recognize this distinction, PerfGuard uses call stack both in unit identification and at run-time. After unit identification, we derive a performance threshold for an unit context which is recognized at run-time and its threshold is verified.

Log2 [35] is a cost-aware logging mechanism that controls the overhead of logging while keeping its performance diagnosis capability. This approach is useful in practice, but they have different focus from our work: a cost-aware run-time logging.

Performance Profilers. Performance analysis tools, such as [36, 14, 21, 22, 13, 34], are popular in the development stage to determine the bottleneck of software functions. These tools constantly and blindly sample CPU usages of target program, some of which rely on profiling code embedded into the program which requires a compilation option. The accuracy of performance diagnosis highly depends on the frequency of sampling and higher frequency causes higher performance overhead. Therefore they are not commonly used in the production stages.

Advanced techniques such as genetic algorithm [55], symbolic execution [31, 28, 59] and guided testing [37] are used to automatically generate the test inputs to trigger performance bugs. PerfGuard can complement these approaches by providing better accuracy in unit threshold determination and run-time detection of threshold violation.

Xiao et al. [57] proposed a technique to use different workloads to identify workload-dependent performance bottlenecks in GUI applications. Our technique may potentially complement their approach by providing the unit type classification and supporting the run-time violation detection. In future work, we also plan to leverage their technique to accurately identify workload-dependent unit thresholds in training runs.

CARAMEL [51] proposed a static analysis technique to automatically find and fix performance bugs that break out of the loop wasting computation from C++ and Java code.

Toddler [52] finds repetitive memory accesses in loops for debugging Java program performance. In comparison, PerfGuard focuses on finding time delays caused by various reasons (memory accesses can be one of them) for native programs.

Mobile App Performance. There are approaches [54, 45] that monitor mobile application binaries to identify the critical paths of user transactions and provides detailed performance breakdowns on the detection of performance issues. Compared to that, our approach utilizes a more general concept of workload units based on the execution partitioning and unit clustering to target a wider scope of applications.

9. DISCUSSION

Target Applications of PerfGuard. PerfGuard automatically recognizes application units and detects performance bugs associated with their latency. Therefore, the main target of PerfGuard is the set of applications with the concept of workload units (also known as app transactions) and bounds of expected response time. Most server programs, interactive software, and GUI applications belong to this category.

The performance of server programs are typically defined by service-level agreements (SLA) which are standardized service contracts regarding quality and responsibilities on violations, agreed between the service provider and the service user. Given a request from a client, the server should respond within a bounded response time.

Another set of programs with workload units is the programs with the graphical user interface (GUI) because people have limited tolerance on the response in practice. Related approaches in human interface [30, 47, 49] show the acceptable response delay for a user interface is around 100 ms. User-interactive applications can take benefits of PerfGuard due to their workload units and corresponding performance.

Training and Variance of Workload. PerfGuard requires a training stage to generate a profile of software workload. It is theoretically incomplete and there is a chance that inexperienced workloads can appear during production runs. To address this issue, PerfGuard uses the unit type inference which allows a variation of unit control flow. Our evaluation shows that PerfGuard is sufficiently effective in monitoring the performance of popular applications. Table 3 and Figure 7 show that PerfGuard effectively detects real world performance bugs with negligible run-time overhead. Also existing techniques [58, 56, 41, 29] can be leveraged to improve our training coverage. We envision future integration of PerfGuard and these techniques.

10. CONCLUSION

We present PerfGuard, a novel system to enable performance monitoring and diagnosis without source code and code knowledge for a general scope of software. PerfGuard automatically identifies loop-based units and determines their performance thresholds. Binary only software is transparently instrumented to include performance guards, which monitor application performance efficiently and report detailed call stacks on the detection of performance anomalies. In the evaluation of six large scale open source software with ten real-world performance bugs, PerfGuard successfully detected all of them with the run-time overhead under 3% showing an efficient solution for troubleshooting performance problems in production environments.

11. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive comments. This research was supported, in part, by NSF under awards 1409668, 1320444, and 1320306. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of NSF.

12. REFERENCES

[1] 7-Zip / Discussion / Open Discussion:7-Zip FM hangs for “Create a Folde... https://sourceforge.net/p/sevenzip/discussion/45797/thread/d5fc0d67/.

[2] 7-Zip / Discussion / Open Discussion:7-Zip hangs when rename a file. https://sourceforge.net/p/sevenzip/discussion/45797/thread/257422c5/.

[3] 7-Zip / Discussion / Open Discussion:7zip file manager hangs in “Flat V... https://sourceforge.net/p/sevenzip/discussion/45797/thread/1cd9e6f4/.

[4] 7-Zip / Discussion / Open Discussion:7zip File Manager Hangs when selections chang. https://sourceforge.net/p/sevenzip/discussion/45797/thread/edd64358/.

[5] AppDynamics : Application Performance Management & Monitoring. https://www.appdynamics.com/.

[6] Bootstrap: the world’s most popular mobile-first and responsive front-end framework. http://getbootstrap.com/.

[7] Bug 45464 - WebDav filesystem module is extremely slow. https://bz.apache.org/bugzilla/show_bug.cgi?id=45464.

[8] CaptureBackStackTrace. https://msdn.microsoft.com/en-us/library/windows/desktop/bb204633(v=vs.85).aspx.

[9] don’t use HEAP_NO_SERIALIZE - it’s very slow · processhacker2/processhacker2@5d340db. https://github.com/processhacker2/processhacker2/commit/5d340dbfdfe50d93f1be016f05507492cfaa443a.

[10] Dynatrace : Application Performance Management & Monitoring. https://www.dynatrace.com/.

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

[12] Fossil: Home. https://www.fossil-scm.org/.

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

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

[15] Home :: Bugzilla :: bugzilla.org. https://www.bugzilla.org/.

[16] Inside Windows 7 - Reliability, Performance and PerfTrack. https://channel9.msdn.com/Blogs/Charles/Inside-Windows-7-Reliability-Performance-and-PerfTrack.

[17] MySQL Bugs: #15811: extremely long time for mysql client to execute long INSERT. https://bugs.mysql.com/bug.php?id=15811.

[18] MySQL Bugs: #49491: Much overhead for MD5() and SHA1() on short strings. https://bugs.mysql.com/bug.php?id=49491.

[19] New Relic : Application Performance Management & Monitoring. https://newrelic.com/.

[20] Notepad++ / Discussion / [READ ONLY] Open Discussion:Notepad++ hangs when opening large files. https://sourceforge.net/p/notepad-plus/discussion/331753/thread/8c702407/.

[21] OProfile: a system-wide profiler for Linux systems. http://oprofile.sourceforge.net/.

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

[23] Project Tracking for Teams - FogBugz. https://www.fogcreek.com/fogbugz/.

[24] QueryPerformanceCounter function. https://msdn.microsoft.com/en-us/library/windows/desktop/ms644904(v=vs.85).aspx.

[25] ToolStatus: major startup/search performance fix · processhacker2/processhacker2@5aeed15. https://github.com/processhacker2/processhacker2/commit/5aeed15656dad7099ec8df87aaf1c350ffb8e2ee.

[26] Windows Error Reporting. https://msdn.microsoft.com/en-us/library/windows/desktop/bb513641(v=vs.85).aspx.

[27] K. Bartz, J. W. Stokes, J. C. Platt, R. Kivett, D. Grant, S. Calinoiu, and G. Loihle. Finding Similar Failures Using Callstack Similarity. In *Proceedings of the Third Conference on Tackling Computer Systems Problems with Machine Learning Techniques*, SysML ’08, Berkeley, CA, USA, 2008.

[28] J. Burnim, S. Juvekar, and K. Sen. WISE: Automated Test Generation for Worst-case Complexity. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE ’09, Washington, DC, USA, 2009.

[29] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI ’08, Berkeley, CA, USA, 2008.

[30] S. K. Card, G. G. Robertson, and J. D. Mackinlay. The Information Visualizer, an Information Workspace. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI ’91, New York, NY, USA, 1991.

[31] B. Chen, Y. Liu, and W. Le. Generating Performance Distributions via Probabilistic Symbolic Execution. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE ’16, New York, NY, USA, 2016.

[32] D. J. Dean, H. Nguyen, X. Gu, H. Zhang, J. Rhee, N. Arora, and G. Jiang. PerfScope: Practical Online Server Performance Bug Inference in Production Cloud Computing Infrastructures. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC ’14, New York, NY, USA, 2014.

[33] Z. Deng, X. Zhang, and D. Xu. BISTRO: Binary Component Extraction and Embedding for Software Security Applications. In *Computer Security - ESORICS 2013 - 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings*, 2013.

[34] M. Desnoyers and M. R. Dagenais. The LTTng tracer: A low impact performance and behavior monitor for GNU/Linux. In *Proceedings of the Linux Symposium*, 2006.

[35] R. Ding, H. Zhou, J.-G. Lou, H. Zhang, Q. Lin, Q. Fu, D. Zhang, and T. Xie. Log2: A Cost-Aware Logging Mechanism for Performance Diagnosis. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, Santa Clara, CA, July 2015.

[36] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A Call Graph Execution Profiler. *SIGPLAN Not.*, June 1982.

[37] M. Grechanik, C. Fu, and Q. Xie. Automatically Finding Performance Problems with Feedback-directed Learning Software Testing. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, Piscataway, NJ, USA, 2012.

[38] 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 34th International Conference on Software Engineering*, ICSE '12, Piscataway, NJ, USA, 2012.

[39] D. S. Hirschberg. Algorithms for the Longest Common Subsequence Problem. *J. ACM*, Oct. 1977.

[40] G. Hunt and D. Brubacher. Detours: Binary Interception of Win32 Functions. In *Proceedings of the 3rd Conference on USENIX Windows NT Symposium - Volume 3*, WINSYM '99, Berkeley, CA, USA, 1999.

[41] L. Jiang and Z. Su. Profile-guided Program Simplification for Effective Testing and Analysis. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, New York, NY, USA, 2008.

[42] 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, New York, NY, USA, 2012.

[43] C. H. Kim, J. Rhee, H. Zhang, N. Arora, G. Jiang, X. Zhang, and D. Xu. IntroPerf: Transparent Context-sensitive Multi-layer Performance Inference Using System Stack Traces. In *The 2014 ACM International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '14, New York, NY, USA, 2014.

[44] K. H. Lee, X. Zhang, and D. Xu. High Accuracy Attack Provenance via Binary-based Execution Partition. In *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*, 2013.

[45] Y. Liu, C. Xu, and S.-C. Cheung. Characterizing and Detecting Performance Bugs for Smartphone Applications. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, New York, NY, USA, 2014.

[46] 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 *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, New York, NY, USA, 2005.

[47] R. B. Miller. Response Time in Man-computer Conversational Transactions. In *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, AFIPS '68 (Fall, part I), New York, NY, USA, 1968.

[48] T. Moseley, D. Grunwald, D. A. Connors, R. Ramanujam, V. Tovinkere, and R. Peri. LoopProf: Dynamic Techniques for Loop Detection and Profiling. In *Proceedings of the 2006 Workshop on Binary Instrumentation and Applications*, WBIA '14, 2006.

[49] B. A. Myers. The Importance of Percent-done Progress Indicators for Computer-human Interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '85, New York, NY, USA, 1985.

[50] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, New York, NY, USA, 2007.

[51] A. Nistor, P.-C. Chang, C. Radoi, and S. Lu. CARAMEL: Detecting and Fixing Performance Problems That Have Non-intrusive Fixes. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, Piscataway, NJ, USA, 2015.

[52] A. Nistor, L. Song, D. Marinov, and S. Lu. Toddler: Detecting Performance Problems via Similar Memory-access Patterns. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, Piscataway, NJ, USA, 2013.

[53] F. Pukelsheim. The Three Sigma Rule. *The American Statistician*, May 1994.

[54] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh. AppInsight: Mobile App Performance Monitoring in the Wild. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, Berkeley, CA, USA, 2012.

[55] D. Shen, Q. Luo, D. Poshyvanyk, and M. Grechanik. Automating Performance Bottleneck Detection Using Search-based Application Profiling. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, New York, NY, USA, 2015.

[56] S. Thummalapenta, S. Sinha, N. Singhania, and S. Chandra. Automating Test Automation. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, Piscataway, NJ, USA, 2012.

[57] X. Xiao, S. Han, D. Zhang, and T. Xie. Context-sensitive Delta Inference for Identifying Workload-dependent Performance Bottlenecks. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, New York, NY, USA, 2013.

[58] R. Yandrapally, S. Thummalapenta, S. Sinha, and S. Chandra. Robust Test Automation Using Contextual Clues. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, New York, NY, USA, 2014.

[59] P. Zhang, S. Elbaum, and M. B. Dwyer. Automatic Generation of Load Tests. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, Washington, DC, USA, 2011.