Uscope: A Scalable Unified Tracer from Kernel to User Space

Junghwan Rhee, Hui Zhang, Nipun Arora, Guofei Jiang, Kenji Yoshihira
NEC Laboratories America
{rhee,huizhang,nipun,gfj,kenji}@nec-labs.com

***Abstract*—Unified tracing is the process of collecting trace logs across the boundary of kernel and user spaces, and has been used to understand the in-depth correspondence between low level events and application program context for diagnosing system failures and performance problems. Crossing the boundary from the kernel space to a user space to collect trace events from dual spaces imposes challenges compared to crossing the boundary in the other way from a user space to the kernel space due to multiple scheduled programs and diverse code layouts in the user space regarding the tracing target. In this paper, we propose a novel unified tracing system called Uscope to systematically trace kernel and unprecedented user code with low overhead. The key idea is to use an efficient variant of stack walking. Uscope lowers stack walking overhead by adjusting the scope of walking in two ways: (1) a highly configurable focus within the call stack, and (2) a per-application tracing that systematically tracks a dynamic set of new, exiting, or transforming processes and threads of an application software. This system is realized by using a flexible stack walking algorithm and a runtime kernel structure, Trace Map. These key features lead to low run-time overhead under 6% relative to native execution on a set of widely used benchmarks.**

Keywords*—*black-box unified event tracing; performance diagnostics; system troubleshooting; data centers

I. Introduction

Modern software systems are complex, often composed of many software layers such as application program binaries, third-party libraries, middleware (e.g., JBoss), low level system libraries (e.g., `glibc`), and kernels. All those layers are potentially subject to dependability problems of software stack such as failures, program bugs, or configuration errors, and can cause functional or non-functional problems during execution.

In order to diagnose such a deep software stack, there have been several tools such as DTrace [10], LTTng with UST [4], SystemTap [16] with utrace [5] that monitor across such boundaries and implement *unified tracing* of kernel and user space events. This is not trivial since the two spaces are isolated for reliability and security.

Figure 1 illustrates two different unified tracing mechanisms. Figure 1(a) demonstrates *precedented* user code tracing [10], [16]; Here, a set of predetermined code is provided as a target for user level probes, and what is traced are the invocations of these *known* probes and their kernel events. We call this type of tracing mechanism *Type 1 Unified Tracing*. However, predicting potential problematic code is not always possible, and in such cases *Type 1 Unified Tracing* would require tedious iterations to guess the problematic code that the probes should target.

978-1-4799-0913-1/14/$31.00 © 2014 IEEE

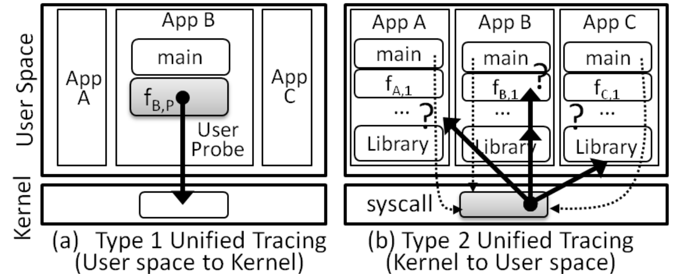

Fig. 1. Types of Kernel & User Space Unified Tracing.

Figure 1(b) presents another tracing mechanism for *unprecedented* user code [10], that we call *Type 2 Unified Tracing*. This approach can collect user space information whenever a kernel probe (e.g., a system call) is triggered. This is a highly desired feature for black-box diagnosis approaches which rely only on kernel level visibility (as compared to customized user-space solutions) to find the root-cause of the problem.

Existing Type 2 unified tracers (e.g., ustack of DTrace [10]) typically rely on a technique that scans the application stack called *stack walking* to collect unprecedented user code information. These solutions have been primarily used in offline debugging environment due to the concern of stack length and high overhead, and Uscope focuses on solving these challenges in an efficient manner. We describe such challenges in details:

Firstly, the code information in application stacks can be massive and complex. For example, a study from the ECLIPSE project [18] reported that the collected stack traces ranged from a single frame to 1,024 frames, with a median length of 25 frames. Clearly, blindly dumping the full stack imposes overwhelming overhead.

Using a static depth for stack walking does not properly mitigate this problem. Typically the top of the stack for system calls are filled by the call sites from code layers in the order of low level libraries, middleware, and the program. The stack frames of interest (e.g., main binary) may not be reached with a simple static depth due to a pile of call sites from other code layers.

Secondly, existing kernel tracers [10] usually log the events (e.g., system call tracing) of the entire system by default. This design is understandable because the kernel serves all application processes. However, a runtime environment commonly has at least dozens of applications running for many reasons

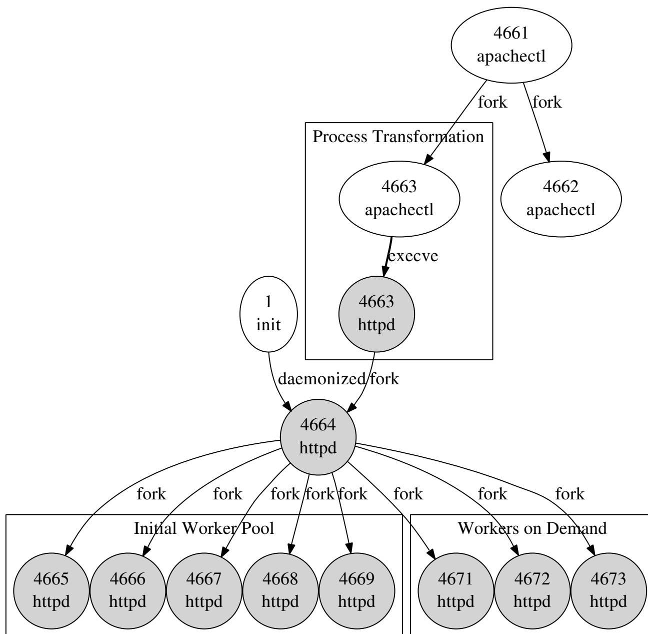

Fig. 2. Runtime Dynamics of the Processes in Apache Web Server. (Note that the graph is simplified by omitting additional worker threads having a similar structure due to the constraint of space).

such as GUI, system logging, resource management, and server programs. If we have a specific application program in mind to trace and debug, system-wide tracing is clearly not the best choice considering the runtime overhead and the log volumes.

Per-application kernel tracing has not been much studied especially on how it can be done automatically and efficiently. Tracing kernel events only for a program may not be a simple task contrast to tracking a pre-forked process whose PID is available because of the following reasons. First, determining whether the currently scheduled process should be traced could be costly without a careful design. In a naïve design, the process name could be used for this purpose. However, performing `strcmp` to match the process name for each kernel event is expensive. Second, the representation of an application could be highly dynamic. It may have a dynamic set of processes and threads forked, cloned, and exiting. Also as very tricky but common cases, programs can turn into other programs by replacing program binaries (e.g., `execve`).

As a concrete example, Figure 2 visualizes the processes of Apache web server. The challenging case is shown in the box labeled as "Process Transformation". Apache is launched by a utility, `apachectl`. It creates child processes and one of them (PID 4663:`apachectl`) turns into a web server process (PID 4663:`httpd`) through a `execve` call. It is crucial to properly handle this *transformation of processes* in order to trace a program from its start. Also Apache manages *a highly dynamic pool of worker processes* to handle requests. The head process (PID 4664) of the server daemon *dynamically increases the number of workers* when web requests are increasing. Note the process hierarchy is not reliable to represent runtime program status, because, for instance, the children of shells (e.g., `bash`, `cmd.exe`, `explorer.exe`), and program schedulers (e.g., `cron`) can be arbitrary programs.

In this paper, we propose a novel unified tracing system called Uscope, to support highly efficient unprecedented user code tracing from kernel events.

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

- *Flexible scope on call stack walking:* Uscope offers flexible configurations on what parts of the user stack to be traced. It provides a simple interface for tuning the tradeoff between run-time overhead and user code coverage, and enables adaptive tracing through a *flexible stalk walking* algorithm.
- *Efficient per-application tracing:* Current unprecedented unified tracing approach impose high overhead due to a wide scope of the full stack walking on multiple programs. We present an kernel structure called *Trace Map* that keeps a pool of trace targets to realize efficient per-application tracing.

The rest of the paper is organized as follows. Section II presents the related work. Section III describes the design of flexible stack walking. In Section IV, the implementation and evaluation of Uscope prototype is described. Section V presents discussions. Section VI concludes this paper.

II. Related Work

Kernel Tracer. Kernel level event tracers [4], [7], [8], [10], [11], [12], [15], [16], [17], [19], [21], [22] have been widely used to analyze system behavior and debug performance problems.

vPath [21] uses kernel events to construct request-processing paths based on the causality of events. Considering dynamic program control flow, solely relying on kernel events may impose inaccuracy in the inference of paths. User level context provided by Uscope can serve as new features to correlate the events and potentially improve the accuracy of the inference.

Magpie [8] collects kernel events and user tracepoints which are injected to capture application/middleware specific events. Dapper [19] is a similar approach developed by Google. These approaches use tracepoints for common libraries and RPC layers and it requires the knowledge on the system regarding what code should be the tracepoints. Uscope's unprecedented user code tracing can reduce the efforts to determine potential tracing points.

There is related work [7], [11], [12], [17] reconstructing paths or understanding the workload using whitebox, greybox, or blackbox approaches. Essentially there is a tradeoff between the intrusiveness to the code and the accuracy of the inference. Uscope works as a blackbox approach but has effects of whitebox approaches, thus improving the accuracy of inference while reducing the cost in deployment such as the source code requirement.

Stack Walking and Debuggers. Stack walking is the process inspecting a program call stack and reporting the active stack frames (called a stack trace) at a certain time point during the execution of a program (e.g., breakpoint, fault). This technique is commonly used during interactive and

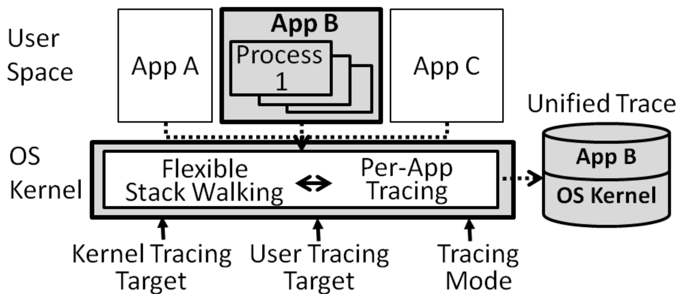

Fig. 3. Overview of Uscope.

post-mortem debugging. Most developers would be familiar with the `backtrace` command in gdb [2]. kgdb and kdb provide remote and local kernel debugging if the kernel is built to support it [6]. Debugging tools in other development environments such as Eclipse, Visual Studio, also provide similar functionalities.

The functionality provided by these tools is extremely useful when trying to understand code flow. However, it is impractical for debugging large-scale softwares. This is mostly because they impose a heavy overhead which becomes a big drawback as code complexity increases. Uscope addresses this problem by finding the corresponding user-space functions for interested events in the system call trace. This comes with a low overhead stack walk which is more practical.

Unified Tracing. Several tools such as DTrace PID provider [10], LTTng with UST [4], SystemTap [16] with utrace [5] provide unified tracing that support the collection of kernel events and user events. However, the major difference of such tools compared to Uscope is that they track the user probes which are predetermined user level program functions (i.e. only functions that have been instrumented will be traced).

DTrace ustack [10] is the closest work providing Type 2 unified tracing with the global scope as well as Type 1 unified tracing. In a deep stack layer, application code is mostly hidden below library code. DTrace's static stack walking depth can limit the tracing impact, but it may not effectively reach user code stack frames of interest. In contrast, Uscope can achieve efficient yet effective tracing of unprecedented user code by flexibly skipping unrelevant code ranges (i.e., low level system libraries) and unrelevant processes/threads in stack walking.

III. Design of Uscope

In this section, we present the design of the Uscope tracing system (shown in Figure 3). Uscope requires three user inputs: Kernel Tracing Target, User Tracing Target, and Tracing Mode. The Kernel Tracing Target is the set of kernel events (e.g. system calls) the user wants to inspect. The User Tracing Target is the application that the user wishes to focus on (e.g., `httpd`). The Tracing Mode specifies the scheme regarding the granularity of stack walking.

The core novelty introduced by Uscope is a variation of stack walking mechanism designed for efficient collection of unified traces. Uscope is composed of the following two components: Flexible Stack Walking and Per-Application Tracing.

A. Flexible Stack Walking

A single program may consist of multiple code ranges due to causal software components such as libraries or plug-ins. The monitoring granularity on this group of information can be customized to provide flexibility and further optimize the monitoring performance. For instance, developers may be interested in what user code is triggering kernel events. However, the top of the user stack could be filled by multiple layers of low level library code such as `glibc`. Uscope provides a configurable *Tracing Mode*, and perform efficient stack walking by defining the scope with in the call stack so that the walking can be completed as soon as the requested information is collected.

Figure 4 presents four example cases of Tracing Modes. More exploration in other configuration or dynamic configuration could be interesting for further study. In these examples, we use a notation R_j for a code range of a binary or a library and $C_{j,q}$ for a call site that belongs to R_j. q is the index of call sites within the code range ($1 \sim n_j$). There is a budget for the maximum number of call sites (S) for each stack walking.

(1) **Mode 1 (Application Mode).** This mode focuses on efficiency of the monitoring overhead and the size of log information. It captures the last function call site that invokes a system call in the main binary (C_{1,n_1}). Thus it guides developers to go back to the last place in their code on kernel events. Note that there are $K-1$ libraries between the kernel and the user binary (R_1). Thus traditional stack walking approaches with limited walking distance may not be able to reach the application code. Flexible walking allows to skip intermediate libraries and jump into the binary of interest.

(2) **Mode 2 (Application All Mode).** This mode is an extension of Mode 1. Uscope attempts to capture all call sites within the main binary ($C_{1,1}, \cdots, C_{1,n_1}$) as far as slots for recording are available. The maximum number of call sites for this mode is $max(n_1, S)$.

(3) **Mode 3 (Library Mode).** In this mode, Uscope captures the last call sites of each code range from the lowest layer to the highest layer up to S call sites. It is useful to validate what kinds of code component or libraries are involved with a system call since a call site from each code range is sampled. The maximum number of call sites is $max(k, S)$ where k is the total number of code ranges.

(4) **Mode 4 (All Mode).** This mode provides the most amount of details of user code information which is the full call stack information.

When a kernel event to be recorded is triggered, Uscope identifies the user stack and walks it to collect a list of user code call sites leading to the current kernel event. During this process, Uscope offers tradeoffs to speed up the tracing by flexibly control the walking granularity. We call this mechanism *Flexible Stack Walking*. Algorithm 1 shows how it works in details.

Tracing Mode represents the flexible walking granularity, which can adjust the monitoring overhead. If we focus on some code assuming other parts or complex libraries are separately inspected, we can improve monitoring performance efficiency by only focusing on such code. This is controlled by listing

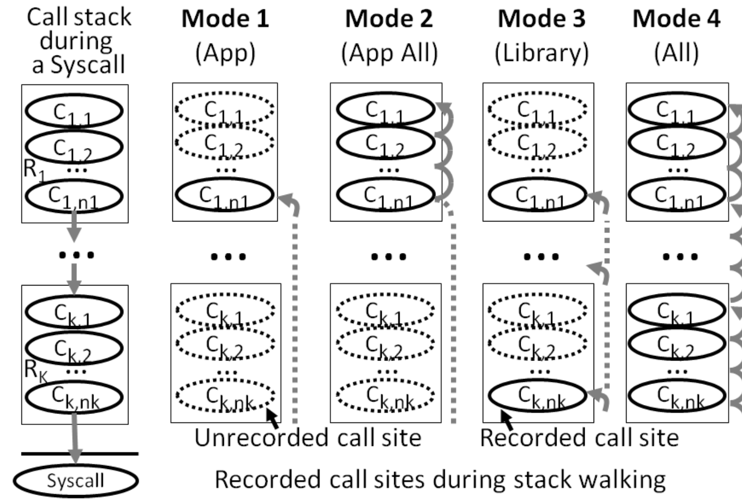

Fig. 4. Examples of Tracing Modes and User Code Information (Call Sites) Collected by Uscope.

Algorithm 1 Flexible Stack Walking

A : Availability map of code ranges
S : Maximum number of call sites to be recorded
PTR : data structure reference from Trace Map
$PTR.CR$: Code ranges, $PTR.CC$: Cache of CR
B : Output buffer array (size is S).
1: **function** FLEXIBLEWALK($UserStack, PTR, B$)
2: $i = 0$; $j =$ | Stack end - Current stack pointer |
3: Initialize $A[|PTR.CR|]$ with the configuration
4: **while** $i < S$ and $j > 0$ **do**
5: $c = Walk(UserStack)$; $j = j - 1$
6: **if** $CollectCallSite(c, A, PTR) > 0$ **then**
7: $B[i] = c$; $i = i + 1$
8: **function** COLLECTCALLSITE(c, A, PTR)
9: $(v, idx) = CheckCRLT(c, PTR)$
10: **if** $v > 0$ **then**
11: **if** $A[idx] == 0$ **then**
12: **return** 0
13: **else**
14: $A[idx] = A[idx] - 1$
15: **return** 1
16: **else**
17: **return** 0
18: **function** CHECKCRLT(c, PTR)
19: **if** c belongs to a code range idx in $PTR.CC$ **then**
20: **return** $(1, idx)$
21: **else**
22: **if** c belongs to a code range in $PTR.CR$ **then**
23: Add the code range idx to $PTR.CC$
24: **return** $(1, idx)$
25: **else**
26: **return** $(0, 0)$

only such code in the scope of code ranges. This information is organized in our data structure called *Code Range Lookup Table (CRLT)* (shown as CR and CC in Algorithm 1). CR is a refined set of code ranges that is only inspected by Uscope, and CC is a cache of CR to speed up lookups.

The main function for stack walking is `FlexibleWalk`. The entire output will be recorded in the buffer B (size S). This buffer is used by multiple code ranges of the program. Tracing mode will configure how many call sites will be allowed for each code range. As the algorithm walks the stack, it fills the output buffer. During the process, the availability map of code ranges (A) tracks how many call sites are remaining to be recorded in each code range. It is initialized in line 3 by the configuration of Tracing Mode.

As Uscope walks the user stack (line 5), the scanned value is checked for (1) whether it belongs to a code range under tracking and (2) whether it can be recorded in the buffer regarding the configuration. The first check is performed by the `CheckCRLT` function. It checks whether the call site belongs to the cache (CC) in the CRLT. If not, the list is used for the lookup and the found entry is put into the cache. The second check is done by consulting the availability map (A).

In extreme cases such as deep recursion, the excessive size of collected call sites could potentially incur high overhead. To avoid such undesired effects, we also use a global budget of traced information S.

B. Per-Application Tracing

In Type 2 unified tracing, when a kernel target is executed, a monitor inspects the corresponding code in the user space. Given that OS kernel is a low level service, the kernel event could be triggered by any process running in the OS. If the application is not the target application of interest, walking user stack will incur needless run-time and storage overhead. Since stack walking is very costly, it is critical to avoid such case to keep the monitoring overhead minimal.

Uscope dynamically activates the kernel tracer itself depending on the processes in the user space, with the focus on the specified application. Selectively activating tracing on a certain group of processes requires the capability to test whether the current process belongs to the target process group, which is identified by the process name. One naïve method is to compare the name of current process and the name of the user tracing target. The overhead for a string match could be

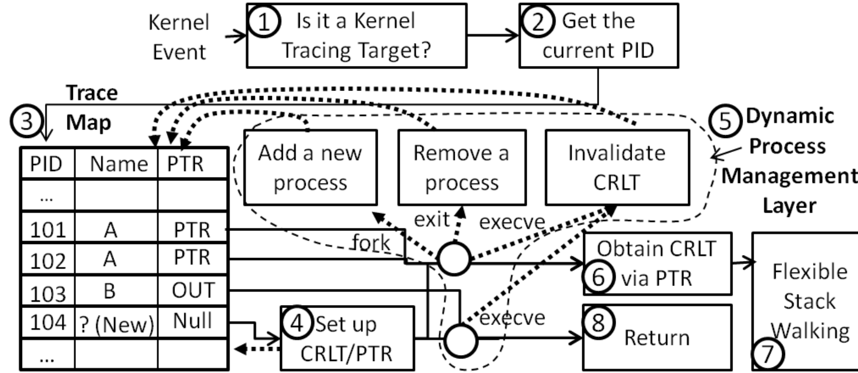

Fig. 5. Operation Steps of Trace Map.

expensive if programs trigger the traced kernel functions very frequently.

Trace Map. This problem is solved in Uscope by managing a pool of traced processes that we call *Trace Map* and avoiding expensive name matchings. It provides an efficient checking of current user program context *in the constant time* with a dynamic array indexed by PIDs. This structure is managed in *non-critical execution paths* thus achieving low amortized overhead.

As shown in Figure 5, efficient checking of Uscope tracing target takes the following steps:

- Step ① : The arrival of a kernel event in the kernel tracing target
- Step ② : The process ID (PID) of the current user program is derived from kernel data structures.
- Step ③ : We use a dynamic array indexed by the PID number to achieve the constant time in the checking.
- Step ④ : When there is a new process, it is not known yet whether it is a process of the user tracing target or not. Thus at the *first* kernel event of a new process, its process name is compared with the user tracing target's name. The PTR field for this process of the Trace Map is initially `Null` indicating its identity is not discovered yet. After this comparison, it is marked either as a PTR which points to a CRLT structure or `OUT` (a special value) indicating no tracing is necessary depending on whether the process belongs to user tracing target or not. From the second event of this PID, the PTR field of the map is retrieved and the traceability is determined quickly in the constant time.
- Step ⑤ : Another important issue is maintaining the Trace Map to ensure accuracy. This problem is solved by capturing all kernel events (e.g., `fork`, `exit`, and `execve`) that can make a change in the status of processes and configuring the map accordingly before the checking occurs. This function is shown as *dynamic process management layer* in Figure 5.
- Step ⑥ : If the kernel event belongs to the user tracing target, the necessary information for stack walking is retrieved from the Trace Map using the PTR field.

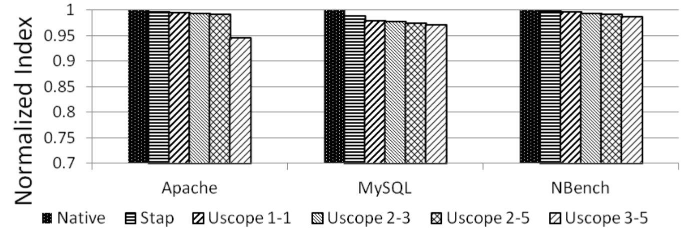

Fig. 6. Performance Comparison of Native Execution (Native), SystemTap (Stap), and Uscope (Uscope 1-1, 2-3, 2-5, 3-5).

- Step ⑦ : The user stack walking is performed on this kernel event.
- Step ⑧ : If it is not a target, which is determined by the value, `OUT`, in the map, it returns without stack walking.

In addition to fast determination of the tracing target, another important role of Trace Map is to reorganize the information necessary for stack walking to perform walking efficiently. When a new process is determined, the code ranges of this process is copied from a kernel data structure (e.g., `vm_area_struct` in Linux) to CRLT and it is linked to the entry of this process in the Trace Map. This map covers the main binary and dynamically linked libraries. Statically linked libraries are combined into the main binary during the compilation stage. Thus Uscope does not need extra handling. Instead they are treated as functions in the binary.

IV. Evaluation

Uscope is implemented by enhancing a production kernel tracer, SystemTap [16], that allows a kernel agent to hook kernel events without modifying kernel source code. Uscope places an interposition layer in SystemTap to additionally perform a light weight stack walking on selected processes under tracing. Since SystemTap can dynamically add or remove hooks in the kernel, Uscope can be simply detached from the kernel at runtime when disabling the tracing, and cause zero overhead. Currently we support stack walking for system calls because they are invoked by user programs and thus have corresponding user level context.

A. Performance Overhead

We evaluated Uscope by comparing its performance to its base system, SystemTap, and the native execution. We chose three well known benchmark suites, the Apache benchmark, the MySQL benchmark, and NBench. These benchmarks test not only realistic server workload but also CPU and memory systems intensively.

Figure 6 shows the normalized performance of SystemTap and Uscope compared to native execution. In three benchmarks, performance numbers are scaled, such that native performance is normalized to 1 (shown as "Native"). Hence, higher values mean better performance in all cases. The tested machine has an Intel Core 7 CPU processor with 8GB of RAM, and its OS is Redhat Enterprise Linux Server 5 64 bits (x86_64).

"Stap" represents the application performance with only system call tracing by SystemTap. We have measured Uscope's performance with several configurations; "Uscope 1-1", "Uscope 2-3", "Uscope 2-5", and "Uscope 3-5" respectively represent performance traced by Uscope configured with Mode 1 (S=1), Mode 2 (S=3), Mode 2 (S=5), and Mode 3 (S=5).

Apache. We tested a commonly used web server, Apache `httpd` (version 2.2.10) with Apache HTTP server benchmarking tool (`ab`), which is configured with 100 concurrency and one million (1×10^6) requests to challenge the web server. We used the "Transfer rate (Kbytes/sec)" from the report of `ab`. The impact of the overhead of Uscope is reasonable in this test given the volume of workload demand. Its workload incurred under 0.8% overhead for Mode 1 and 2 (S=3 and 5), and 5.6% overhead for Mode 3 with S=5 in the throughput (Transfer rate) of the benchmark.

MySQL. MySQL is a widely used open source database. We tested this software (version 5.6.10) with the MySQL Benchmark suite which consists of 9 tests: `alter-table`, `ATIS`, `big-tables`, `connect`, `create`, `insert`, `select`, `transactions`, and `wisconsin`. We used the total time taken to finish the benchmark suite. Uscope is light weight for this intensive workload. It incurred less than 3.3% overhead in the Mode 1, 2 (S=3 and 5), and 3 (S=5) in this benchmark.

NBench. This benchmark is a port to Linux/Unix of BYTE's Native Mode Benchmarks (verison 2.2.3). We used "Memory Index", "Integer Index", and "FP Index" from the results of this benchmark. NBench is mostly composed of memory, integer, and floating point intensive operations. Uscope rarely affects computational workload like NBench. The highest overhead is 1.3%.

B. Case Study: Root-Cause Localization via System Call/User Code Co-Analysis

Uscope can provide an excellent aid in assisting kernel level debuggers to localize the problem in user-space. There have been several approaches [9], [14], [20] using user context to localize bugs. Uscope can be utilized as a building block of such approaches; thus providing user context transparently by avoiding source code modification or predetermined probes to instrument programs. As one concrete example of such usages, here we demonstrate one usage scenario on localizing the root-cause of a performance problem enabled by the combination of kernel and user code information of Uscope. The purpose of this section is illustrating the new feature of Uscope that can introspect the application inside and provide clues on the root-cause of a performance problem.

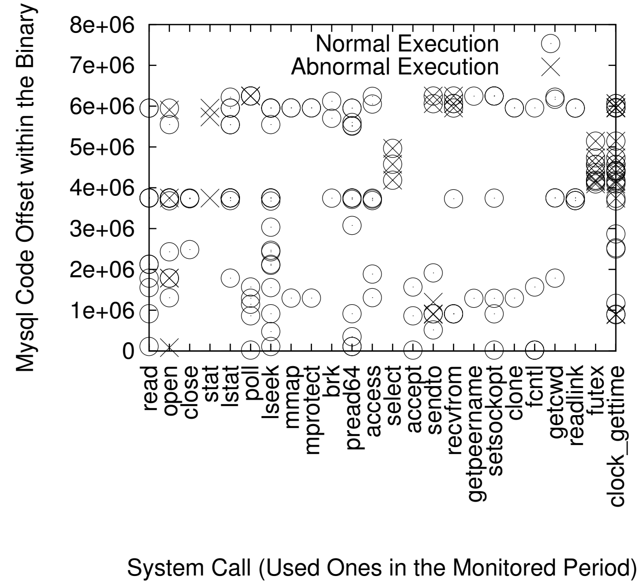

Fig. 7. Comparison of User Code (Y-axis) and System Call (X-axis) Mapping in Normal and Abnormal Execution for Troubleshooting a MySQL Performance Problem.

This problem happens in a multi-tier service testbed, Petstore [3], which consists of three tiers: Apache 2.2.3, JBoss 3.2, and MySQL 5.6.10. The symptom of the problem is that the web requests are not successfully handled and the pages with missing or garbled information are returned to clients. We first localized MySQL node is the suspect by replacing each tier with a backup node. Still we do not have any clue what caused this problem and need to understand more precise application level root-cause to fix this problem and recover this node. Before and after this anomaly happens, we generated Uscope traces using Mode 2 ($S = 3$) for a period of test requests which are generated by accessing the links for the menus in the main Petstore page to list the details of animals.

Note that MySQL is a large scale application that heavily relies on threads. With the basic configuration, over 20 threads were running and among those at least 8 or 9 threads generated system calls during the monitored period. For existing tools that require to specify pid numbers, it will be a hassle to list all processes/threads. And it is not obvious to select some pids because it is hard to know which threads are main workers just based on numbers. Moreover when a new thread or process is forked or exits, it will be tricky to add and remove the process automatically on time without missing its system calls. Uscope handles such complex issues smoothly with the automated process target tracking technique.

Figure 7 illustrates the summary of MySQL's execution. X axis shows the list of system calls triggered by `mysqld` during the workload. In addition, the application code triggering such system calls is represented as an offset within the code binary

in the Y axis. This offset is mapped to a specific code line of the program. Multiple code sites per a system call are marked in the Figure without hierarchical priority for simplicity. $\bigcirc$ and $\times$ marks respectively represent MySQL's code triggering system calls before and after the performance problem.

First of all, the comparison of two sets of system calls clearly indicates anomaly due to distinct distribution of system calls for similar user requests. More important step next is to understand what specific problem caused this performance anomaly. Uscope brings a new observation by stitching kernel information and unprecedented user code information together as shown in Figure 7.

In general the abnormal execution has limited diversity of system calls lacking several types of system calls that exist in normal execution. A noticeable difference is that some system calls such as `read` and `accept` syscalls only exist in normal execution while `stat` syscalls exist only in the abnormal execution. Regarding the call stack information from Uscope, `read` syscalls are triggered by the user code, `my_read`, and `accept` are triggered by `handle_connections_sockets`. These functions are commonly triggered when MySQL serves database entries replying to the requests from the second tier. Lacking such events confirms that in the abnormal workload MySQL fails to handle requests. `stat` system calls are triggered by `my_stat` which was called by `archive_discover`. This call sequence indicates the abnormal workload has a problem in finding the database file.

This performance anomaly was introduced by misleading operations in the database tier while the MySQL server operates at runtime. Specifically MySQL's internal data files are damaged causing the database tables to be tampered with. As the consequence, MySQL could not serve the queries from the second tier (i.e., JBoss) properly causing incomplete web pages to the users.

This *co-analysis* based on system calls and user code call stack information reveals rich information on user program internals illustrating what functions execute or fail therefore explaining the root-cause of this performance problem. Uscope enables deep understanding on application details without expensive debugging techniques such as `gdb` or `ptrace` which do not scale well in realistic workload.

C. Case Study: Transparent Deadlock Root-Cause Inspector

In this section, we present another usage case to use Uscope for understanding a concurrency bug. Concurrency bugs are often hard to be reproduced as they are triggered due to non-determinism in parallel execution of processes/threads. This debugging process is much easier with Uscope as it provides a light weight stack trace of the program *on the spot*.

We applied Uscope to a real case of Apache web server (Case number: Apache #42031 [1], also studied in [13], [23]). In this case, there are two pthread mutex variables involved: `timeout` and `idlers`. This bug is triggered when a worker thread is blocked on the `timeout` mutex before it signals the listener thread. (Apache threads are implemented using `pthreads`, which internally use `futex` system calls which are in turn used by Uscope to capture the call stack).

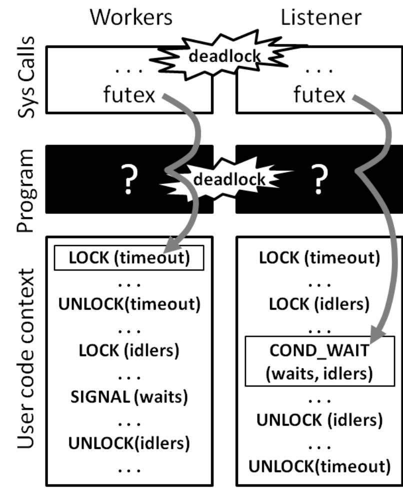

Fig. 8. Revealing User Context of a Blackbox Program under Deadlock (Apache Bug #42031). Uscope is applied on `futex` system calls. The presented code is simplified to assist the understanding.

We used Mode 2 ($S = 5$) to collect up to the last five call sites within the main `httpd` binary at the time when the `futex` system calls were invoked. Uscope provides what user code was executed at `pthread` locks and unlocks when the deadlock occurs: `apr_thread_mutex_call` (a wrapper of `pthread_mutex_call`) call in a worker thread and `apr_thread_cond_wait` (a wrapper of `pthread_cond_wait`) call in a listener thread. The user code during the deadlock pinpoints the exact combination of the conditions that cause the deadlock, which is the very detailed information to help the understanding of the bug.

Compared to user level tools such as `gdb` or core dumps based on `ptrace`, Uscope does not suffer performance perturbation caused by a user level tracing mechanism. In particular this case could be analyzed by such tools because it remains in a steady state (i.e., deadlock). Otherwise it is hard to know *when* to apply those tools. Uscope, however, can investigate bug scenarios without any such assumption on the application status regardless whether a program enters a steady state or not.

V. Discussion

System-Call-Driven. Uscope is motivated by the application of stack walking mechanisms in conjunction with kernel event traces to debug large scale applications. Due to its implementation that Uscope is system-call-driven (i.e. functions in the stack leading to a system call can be tracked), function calls not resulting in system calls are not included in the current evaluation. However, this coverage can be easily extended by triggering stack walking in an extended set of kernel functions. For instance, context switch events are effective candidates to sample user space call stacks even when the program has mostly user space workload.

Native vs. Non-native Programs. Uscope works for native programs written in C/C++ languages. Non-native programs such as Java applications have different mechanisms for the call stacks. Our current implementation of Uscope does not go into the stacks other than native programs. This is not

a technical challenge of Uscope, but simply an aspect of implementation. For instance, the viability of stack unwinding in Java can be observed in Java debugger (e.g., `jdb`) and Java stack inspection tools (e.g., `jstack`), and a similar stack unwinding implementation can be added to Uscope.

VI. Conclusion

In this paper, we presented the design and implementation of a novel kernel tracer, Uscope, that provides efficient unified tracing of kernel and unprecedented user code. Our prototype demonstrates that unified tracing can be implemented with the overhead as low as under 6% compared to native execution in a widely used set of benchmarks. We also presented the usage scenarios of Uscope illustrating how unified traces can benefit performance diagnosis and root-cause analysis of bugs.

References

[1] Apache Bug #42031 42031 - EventMPM child process freeze. https://issues.apache.org/bugzilla/show_bug.cgi?id=42031.

[2] gdb: The GNU Project Debugger. http://sources.redhat.com/gdb/.

[3] *The Java PetStore 2.0*. http://www.oracle.com/technetwork/java/index-136650.html.

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

[5] SystemTap User-Space Probing. http://sourceware.org/systemtap/SystemTap_Beginners_Guide/userspace-probing.html.

[6] Using kgdb, kdb and kernel debugging internals. http://kernel.org/doc/htmldocs/kdb.html.

[7] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proceedings of the nineteenth ACM symposium on Operating systems principles (SOSP '03)*, 2003.

[8] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation (OSDI'04)*, 2004.

[9] M. D. Bond, G. Z. Baker, and S. Z. Guyer. Breadcrumbs: efficient context sensitivity for dynamic bug detection analyses. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation (PLDI '10)*, 2010.

[10] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the annual conference on USENIX Annual Technical Conference 2004 (USENIX '04)*, 2004.

[11] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks (DSN '02)*, 2002.

[12] U. Erlingsson, M. Peinado, S. Peter, and M. Budiu. Fay: extensible distributed tracing from kernels to clusters. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*, 2011.

[13] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems (ASPLOS XIII)*, 2008.

[14] B. Lucia and L. Ceze. Finding concurrency bugs with context-aware communication graphs. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 42)*, 2009.

[15] 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.

[16] 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.

[17] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: detecting the unexpected in distributed systems. In *Proceedings of the 3rd conference on Networked Systems Design & Implementation (NSDI '06)*, 2006.

[18] A. Schroter, N. Bettenburg, and R. Premraj. Do stack traces help developers fix bugs? In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 118 –121, may 2010.

[19] 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.

[20] W. N. Sumner, Y. Zheng, D. Weeratunge, and X. Zhang. Precise calling context encoding. *IEEE Transaction of Software Engineering*, pages 1160–1177, 2012.

[21] 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 *Proceedings of the 2009 conference on USENIX Annual technical conference (USENIX '09)*, 2009.

[22] A. Tamches and B. P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Proceedings of the third symposium on Operating systems design and implementation (OSDI '99)*, 1999.

[23] Y. Wang, S. Lafortune, T. Kelly, M. Kudlur, and S. Mahlke. The theory of deadlock avoidance via discrete control. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '09)*, 2009.