Diagnosing Performance Variations by Comparing Multi-Level Execution Traces

François Doray, *Student Member, IEEE* and Michel Dagenais, *Senior Member, IEEE*

Abstract—Tracing allows the analysis of task interactions with each other and with the operating system. Locating performance problems in a trace is not trivial because of their large size. Furthermore, deep knowledge of all components of the observed system is required to decide whether observed behavior is normal. We introduce TraceCompare, a framework that automatically identifies differences between groups of executions of the same task at the user space and kernel levels. Many performance problems manifest themselves as variations that are easily identified by our framework. Our comparison algorithm takes into account all threads that affect the completion time of analyzed executions. Differences are correlated with application code to facilitate the correction of identified problems. Performance characteristics of task executions are represented by a new data structure called enhanced calling context tree (ECCT). We demonstrate the efficiency of our approach by presenting four case studies in which TraceCompare was used to uncover serious performance problems in enterprise and open source applications, without any prior knowledge of their codebase. We also show that the overhead of our tracing solution is between 0.2 and 9 percent depending on the type of application.

Index Terms—Performance analysis, tracing, software visualization, concurrency, operating systems

◆

1 INTRODUCTION

PERFORMANCE is a critical requirement for many applications. Long delays are among the main sources of user frustration [2] and have a significant impact on revenue [3]. Despite that, it is often difficult for a single developer to understand all the factors that influence the performance of an application. This is mainly due to the multiple levels of abstraction that are supposed to ease software development: frameworks, operating systems and virtualization. Furthermore, diagnosing performance problems is hard because they don't necessarily trigger error conditions and often can't be easily reproduced.

Tracing is a technique that consists of recording events during the execution of a system. Events have a timestamp, a type and a payload. Tracing is well suited to analyze performance problems. Popular tracers achieve low overhead, which allows them to be enabled on production systems to capture bugs that occur infrequently. Tracing provides detailed chronological information, unlike profiling that only provides global statistics for a given time range. A trace can, for example, reveal what happened on the whole system during a long system call issued by a process of interest. However, unless user space applications are carefully instrumented, it is hard to relate the low-level events of a trace to the higher-level logic. Also, it is hard to locate abnormal behavior in the overwhelming amount of information contained in a trace, without deep knowledge of the observed system.

Our objective was to build a tool that automatically highlights differences between two sets of executions of the same task. The comparison must take into account all factors that have an impact on performance, including off-CPU wait time and interferences between processes. Also, it must relate any difference found with the relevant source code, a requirement that is crucial to ease the diagnosis and correction of problems [4]. Our tool should allow a developer that doesn't have a full knowledge of a system to discover and fix performance variations caused by factors such as configuration changes, programming errors or sporadic competition between tasks.

This paper introduces TraceCompare, a trace comparison tool and underlying algorithms that achieve all these goals. After reviewing relevant existing work, we describe a data model to summarize performance characteristics of task executions. We then explain how to record detailed multi-level events during an execution and how to process them to generate a database of executions based on our data model. We introduce a GUI that facilitates the identification of differences between groups of executions. We describe four *real* performance problems found in enterprise and open-source applications and explain how they were diagnosed using TraceCompare. Finally, we measure the overhead incurred by each component of our solution and show that it is low enough for use on production systems.

2 RELATED WORK

2.1 Analyzing Execution Traces

Our group proposed using a state history to simplify and accelerate the analysis of large traces [5]. A state history is a data structure that is built incrementally while a trace is read. It keeps track of various attributes such as the current running thread on each CPU or the total number of bytes read from the disk since the beginning of the trace. The data

- *F. Doray is with the Computer and Software Engineering Department, Ecole Polytechnique de Montreal, Montreal, QC H3T 1J4, Canada, now with Google, Mountain View, CA 94035.*
  E-mail: francois.pierre-doray@polymtl.ca.
- *M. Dagenais is with the Computer and Software Engineering Department, Ecole Polytechnique de Montreal, Montreal, QC H3T 1J4, Canada.*
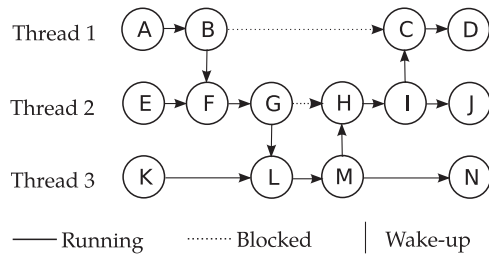 *E-mail: michel.dagenais@polymtl.ca.*

Fig. 1. Graph of dependencies between threads.

is stored in a way that allows fast queries about the value of an attribute at a given time, even when the state history exceeds the size of the main memory. The TraceCompass[1] trace analysis tool relies on this data structure for its interactive views. State histories allow the computation of system metrics for any given time interval in logarithmic time [6].

DTrace provides a dynamic language to compute summary statistics from these events as they occur [7]. This has the advantage of reducing the amount of data that needs to be recorded. However, it can be hard to know ahead of time what statistics will be needed. Therefore, multiple iterations can be required to diagnose a problem. This is impractical if the problem occurs only under specific conditions. Also, summary statistics don't provide as much insight into a problem as a detailed chronology of events.

2.2 Extracting Task Executions from a Trace

In previous work, our group presented a critical path algorithm to retrieve all execution segments that contribute to the latency of a task [8]. The algorithm starts by building a graph of dependencies between threads (Fig. 1). Horizontal edges represent the passage of time on a thread while vertical edges represent wake-up relationships between threads. A critical path is computed by recursively replacing the blocked edges of a thread by the edges and nodes of the waking thread. For the illustrated graph, the computed critical path would be $A, B, F, G, L, M, H, I, C, D$.

We identify dependencies between threads on the same host using the sched_wakeup event of the Linux kernel and dependencies between threads that span multiple hosts using TCP packet matching. The analysis works well with any kind of application because it requires no user space instrumentation. While this tool indicates in which threads time is spent, it does not relate this information to user space functions, which makes it difficult to use for fixing application code. Also, the interactive view shows only one execution at a time, which is impractical to compare latency among a great number of executions. Perfscope[9] is a performance analysis tool that provides a heuristic to find user space functions associated with events from a kernel trace. However, the results are not always accurate.

StackMine [10] uses events, that describe wake-up relationships between threads, to extract from a trace all the call stack events, that are on the critical path of a task execution that spans multiple threads. The extraction can be performed on a large number of traces in order to identify call stacks that have the biggest impact on the performance of a system. Similar call stacks are clustered in order to avoid

spreading the impact of a single performance problem across multiple call stacks. StackMine doesn't provide any way for the user to visualize the dependencies between the threads for which call stacks are extracted. Also, performance degradations that occur infrequently may be invisible in StackMine.

A portion of the requests received by Google are traced by Dapper [11]. The tool reconstructs the flow of individual requests using events generated by the company's communication libraries. Developers can access summary statistics or inspect individual requests through a Web interface. Filters help them find executions that contain abnormal latencies. Dapper relies on the fact that all communications go through a Google library and isn't designed for heterogeneous environments.

2.3 Comparing Task Executions

The "Frames" mode of Chrome developer tools presents the processing time of frames using bar charts [12]. Slow frames are shown as tall bars. Colors within each bar indicate in which states time was spent (JavaScript, rendering, GC...). Developers can determine why a frame took more time than others by comparing the distribution of time per state between different frames. The scope of this tool is limited to the browser. It doesn't take into account interactions between processes. A similar view was developed to compare the states in which real-time tasks spend their time from a kernel point of view [13].

A differential flame graph is a visualization tool to compare two CPU profiles [14]. It shows stack frames as superposed rectangles. The width of a rectangle is proportional to the time spent in a stack frame within the second profile. The color of a rectangle shows the difference between the time spent in the associated stack frame in the two profiles. Because a profile only provides total counts for a given period, this tool cannot reveal the cause of infrequent latencies. Flame graphs have been adapted to show different metrics, but they have never presented all the work done on the critical path of a task.

Comparisons of CPU profiles have been used to diagnose performance variations between multiple versions of the same application [15], evaluate the impact of configuration changes [16] and assess the scalability on an application with respect to the size of its input [17].

IntroPerf [18] is a tool to compare the time spent in different stack frames between multiple executions of a program. Unlike differential flame graphs, it is not limited to the analysis of CPU consumption: it takes into account the time spent waiting off-CPU. Differences between the executions are shown using a heat map where each row represents an individual call stack and each column represents a level of the stack. Since the standard deviation of the latency of each stack frame is not taken into account, stack frames that have an inherently variable latency might be wrongly highlighted on the heat map. Also, IntroPerf is not designed for the analysis of multithreaded applications. Finally, the user has to provide a set or traces with normal executions and a set of traces with slow executions. This is not adapted to a case where normal and slow executions are interleaved in the same trace.

TraceDiff [19] is a visualization tool to compare two function call traces. A function call trace contains an event for

1. http://projects.eclipse.org/projects/tools.tracecompass

every function entry and exit. TraceDiff displays two such traces simultaneously as mirrored icicle plots. Corresponding function calls are connected by hierarchical edge bundles. This facilitates the identification of individual function calls that didn't take the same time or weren't executed in the same order in the two traces. However, it doesn't allow simultaneous comparison of many executions to reveal trends. Also, generating function call traces is costly. Other researchers have developed a heuristic to get an approximate function call trace with low overhead [18]. Efficient algorithms to compute the optimal alignment between two function call traces have also been developed [20].

Comparison of system call traces recorded on different operating systems has been used to detect intrusions [21]. Since some variations are inherent to the use of different operating systems, the authors transform low-level events into higher-level concepts to allow a meaningful comparison. Their algorithm computes a correlation score for two traces, but doesn't show individual differences. Also, it doesn't take timing into account.

2.4 Statistical CPU Profiling

A statistical CPU profiler is a tool that can record the full call stack of a program at regular intervals. This reveals which functions use more CPU time, with minimal overhead. While statistical profilers don't provide the chronological order of events, which is required to analyze interactions between multiple tasks, some of these techniques could be reused to get insight into the logic of user space programs, without requiring manual instrumentation.

Google Perftools provides a CPU profiler [22]. A dynamic library must be linked with programs to analyze. In its initialization phase, the library registers a `SIGPROF` signal handler and starts an `ITIMER_PROF` timer. The timer is decremented whenever a thread of the process consumes CPU time. When a thread reaches the expiration of the timer, the `SIGPROF` signal handler is invoked and captures the thread's call stack. Alternatively, `perf` leverages support from the Linux operating system to profile program [23].

Capturing the call stack of a program, when compiled with the frame pointer, is just a matter of following a linked list formed by base pointers pushed on the stack. However, in order to keep the `ebp` register available for computation, GCC does not preserve the frame pointer for x86-64 binaries, since version 4.6.0.[2] The `.eh_frame` sections of these Executable and LinkableFormat (ELF) binaries provide rules to restore the register values of the previous (oldest) stack frame from any instruction [24]. A call stack can be captured by applying these rules to retrieve the value of the instruction pointer at each stack frame. The `.eh_frame` section is present, even when debugging information is stripped, because it is required to handle exceptions.

The `libunwind` library implements the logic to capture call stacks using rules from the `.eh_frame` section. For good performance, it recognizes stack frames that use a standard layout, and caches that information to handle them with an optimized algorithm when they are encountered again [25]. Google Perftools relies on `libunwind` to
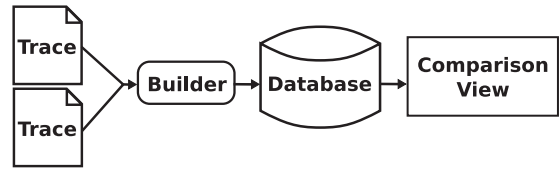

Fig. 2. Architecture of TraceCompare.

capture call stacks online. Instead, `perf` records two pages of memory and runs `libunwind` offline [26]. This reduces the computation overhead, but leads to huge profile files.

A CPU profile can be represented using a calling context tree (CCT); a data structure introduced by [27] and reused by [4], [28]. In a CCT, each node represents a call stack. The children of a node associated with a call stack C of size n are associated with call stacks of size $n + 1$ prefixed with C. The root of the CCT represents the empty call stack. Each node is annotated with the time spent in its associated call stack. Unlike a call tree which contains a distinct node for each individual function call, a CCT combines all calls associated with the same call stack into a single node. This means that a CCT will be much smaller than a call tree for an execution that contains a lot of loops. A CCT does not show interactions between different threads.

3 SOLUTION

In this section, we present the design and implementation of TraceCompare. First, we describe a data model to store performance characteristics of task executions in a database. Second, we explain how to record the information required to build this database with minimal overhead through tracing. Third, we explain algorithms used to build the database from traces. Finally, we present the user interface that highlights differences between groups of task executions. The general architecture of the tool is summarized in Fig. 2.

3.1 Data Model

We introduce a new data structure called enhanced calling context tree (ECCT) to represent the performance characteristics of task executions. Our database of executions and the algorithms that power our GUI rely on this new data structure.

While a CCT represents call stacks encountered on a single thread, an ECCT represents all latencies that affect the duration of a task execution, even if they come from different threads. Also, for each latency, the ECCT encodes call stacks captured on each thread that is part of the chain of dependencies connecting the latency to the analyzed task. Despite the fact that this information is crucial to help application developers find source code associated with observed performance problems, we believe that the ECCT is the first data structure to encode it.

For executions that spend all their time on the CPU and involve a single thread, the structure of the ECCT is identical to that of a CCT. We annotate each node with the time spent in the associated call stack during the analyzed execution. Optionally, more annotations such as the number of page faults or amount of memory allocated in the call stack can be added.

For executions that contain off-CPU wait times or that involve multiple threads (possibly distributed on multiple

2. https://gcc.gnu.org/onlinedocs/gcc-4.6.0/gcc/Optimize-Options.html

Time	Thread 1	Thread 2
1	Call A	Call X
2	Call B	Return X
3		Call C
4		Return C
5	Wait R_1	Call C
6		Wait block device
7		Release R_1
8	Return B	Return C
9	Call C	
10	Return C	Call X
11	Return A	Return X

Fig. 3. ECCT for a multithreaded task execution, without parallel work.

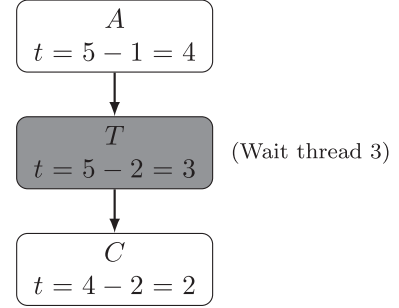| Time	Thread 1	Thread 2	Thread 3
1	Call A		
2		Call B	Call C
3	Wait R_1 and R_2	Release R_1	
4		Return B	Release R_2
5	Return A		Return C

Fig. 4. ECCT for a multithreaded task execution with parallel work.

duration wouldn't change the total duration of the analyzed task execution. A comparison of this ECCT, with the ECCT of an execution in which R_1 is released after R_2, would correctly reveal that the call to B has replaced the call to C on the critical path of the task.

3.2 Tracing Task Executions

We now describe the techniques used to gather the information required to build the data model that we just described.

The Linux Trace Toolkit Next Generation LTTng[3] tracer can record events from the Linux kernel and from user space applications into a single trace [29]. It is also designed to have a minimal overhead on traced systems. It is therefore well suited to our goal of collecting all the factors that contribute to the execution time of tasks in production environment.

3.2.1 Execution Delimiters Events

A single trace usually contains events belonging to multiple executions that occurred in series or in parallel. To allow a comparison between different executions, it is necessary to demultiplex these events. To do so, the trace must provide a way to identify the start and end points of each execution. Sometimes, existing events of the Linux kernel can be used for this purpose. For example, the `syscall_exit_accept` event (generated when a connection is accepted on a socket) and the `syscall_entry_shutdown` event (generated when a connection is closed) correctly delimit requests received by an Apache server. An advantage of using existing events is that no access to the source code is required. When no existing events can act as delimiters, LTTng-UST probes must be inserted statically in the source code. Different probe types can be used to delimit different execution types. That way, our analysis tool is able to tag executions when they are inserted in the database so that the GUI only allows comparison of executions of the same type.

3.2.2 Critical Path Events

When an execution involves a single thread running 100 percent of the time on a CPU, it is trivial to retrieve its

hosts), the CCT is enhanced so that all factors that contribute to the total time of an execution are taken into account. First, for any off-CPU wait time, an artificial function named after the wait reason (timer, block device, network, preemption...) is pushed on the stack. Second, whenever a thread is waiting for another thread, the stack of the second thread is concatenated to that of the first thread. This rule can be applied recursively as many time as required when there is a chain of dependencies between threads. The dependencies can be either direct (a thread is waiting for another thread to compute a result or perform an action) or indirect (a thread is waiting for another thread to release a resource such a the CPU or a mutex).

One might be concerned about the case where indirect dependencies between two different task executions lead to the concatenation of their ECCT. A safeguard could be added to stop the recursion in such a case. However, in our experience, it was rather useful to be able to see which parts of an execution could block other executions. Programs that use a limited number of worker threads have ECCTs of reasonable size, even when a lot of requests are queued.

Fig. 3 shows the expected ECCT for a task execution that spans 2 threads. Observe that the calls to X on thread 2 are not represented in the ECCT, because they are not considered to be on the critical path of the analyzed task execution.

Fig. 4 shows the expected ECCT for task execution that involves parallel work. Observe that the call to B on thread 2 is not represented in the ECCT, because it is not on the critical path of the analyzed task execution. Indeed, reducing its

3. https://lttng.org

TABLE 1
Mapping Between Interrupt Context and Wait Reason

Interrupt Context	Wait Reason
Soft IRQ 1	Timer
Soft IRQ 2, 3	Network
Soft IRQ 4, 5	Block Device
IRQ 19, 23[a]	USB Device
High Resolution Timer	Timer
None	Other Thread

a. Users should run cat/proc/interrupts to make sure that this is right on their system.

events from the trace: we simply look for events between the start and end points on the execution's thread. However, in parallel systems, most executions require collaboration between multiple threads. TraceCompare relies on the critical path algorithm proposed by our group and introduced in Section 2.2 to find all thread segments that contribute to the total time of an execution. This algorithm requires a few events of the Linux kernel to work properly.

The sched_switch event indicates that a new thread starts running on a CPU. Its payload gives the tid of the new thread and tells whether the previous thread was preempted or blocked. The sched_wakeup event is emitted when a blocked thread becomes ready to run. The reason why the thread was blocked can be deduced from the interrupt context in which the sched_wakeup event is emitted. Table 1 presents a mapping between interrupt contexts in which a sched_wakeup event can be emitted and the reason why the waked up thread was waiting.

The irq_handler_entry, irq_handler_exit, hrtimer_expire_entry, hrtimer_expire_exit, softirq_entry and softirq_exit events allow to keep track of interrupts handled on each CPU in order to correctly interpret sched_wakeup events. The inet_sock_local_in and inet_sock_local_out events provide the sequence number and flags of TCP packets. Using these fields, incoming and outgoing packets can be matched to uncover dependencies between threads that communicate through TCP.

3.2.3 Call Stacks Events

TraceCompare must make it easy to relate performance variations, between multiple executions of the same task, to erroneous logic in user space applications. LTTng allows developers to insert custom tracepoints in their code to correlate kernel events with the logic of user space applications. This is, however, not suited to the requirements of TraceCompare. Indeed, we want to build a solution that developers can use on systems that they don't fully understand. Requiring to create tracepoints would represent a major maintenance burden and developers would inevitably forget to instrument important parts of their applications. Requiring instrumentation of third-party libraries would be even harder.

TraceCompare gets insight into user space applications without any manual instrumentation by capturing their call stack in two different contexts. The first context in which the stack is captured is at the expiration of a timer that is decremented when a thread is running, a technique already used by statistical profilers. However, that doesn't provide information about call stacks that block the thread. An interesting observation is that most blockings occur within system calls. For example, a thread can block while it waits for a mutex to be available (futex()) or for data to be read from a file (read()). Therefore, also capturing the call stack on system calls gives a complete portrait of the contexts in which a thread spends its time. To our knowledge, TraceCompare is the first tool that combines CPU and system call stacks to create a call stack history that accurately describes both on-CPU and off-CPU time.

Statistical CPU profilers can keep track of the number of captured samples per call stack using an hash table. This is sufficient to generate summary statistics for the full analyzed period. However, our goal is to provide per-execution statistics and we don't know beforehand to which execution a thread segment belongs to. Furthermore, during our off-line analysis, we need to be able to correlate kernel events to user space stacks and to combine multiple user space stacks occuring simultaneously on different threads. For that reason, a timestamped LTTng-UST event is emitted every time a stack is captured. Stacks captured while user space code is running produce a cpu_stack event while stacks captured during system calls produce a syscall_stack event.

To get the stacks of a process captured, the lttng-profile dynamic library must be preloaded into it. The library uses the same technique as Google Perftools to generate a SIGPROF signal at regular intervals, from which cpu_stack events are generated.

The SIGPROF signal must also be sent to capture the user space stacks associated with system calls. Unfortunately, experimentations show that handling a signal can take longer than the duration of some fast system calls. Therefore, sending a signal and capturing the stack for each system call can add a prohibitive overhead to programs making a lot of short system calls. The problem is that, unlike statistical CPU profiling, which generates a fixed number of events per time unit and per thread, the overhead of this instrumentation is proportional to the number of system calls. To solve that, we decided to capture the user space stack only for system calls with a duration above a threshold. This effectively sets a reasonable upper limit on the rate of events generated by each thread. Because performance is generally not affected by very short system calls, this doesn't undermine the accuracy of our analysis. We found that a 100 μs threshold was a good compromise between precision of the analysis and low overhead. Note that while tracers such as Event Tracing for Windows (ETW) and perf can also capture the call stack of system calls, they don't provide a way to limit this operation to long system calls.

The duration of system calls is tracked by a kernel module. When it is loaded, the lttng-profile library registers the process with the kernel module through an ioctl interface. The kernel module uses the TRACE_EVENT mecanism [30] to register callbacks that are executed at the entry and exit of system calls. On system call entry, the kernel module checks whether the issuing process is registered. If so, the current timestamp is saved in a *read-copy update* (RCU) hash table with the source thread id as the key. On system call exit, the duration of the system call is computed, if the start time is available in the hash table. If the duration is higher than a threshold, a SIGPROF signal is sent to the source

thread. `lttng-profile` catches the signal and captures the call stack, as soon as the control goes back to user space.

The first time that a thread does a system call, an entry is added to the hash table and it is reused until the thread exits. Therefore, most of the time, tracking the duration of a system call simply requires one update and one lookup in the hash table, two very efficient operations.

Our `SIGPROF` signal handler uses `libunwind` to capture call stacks. This library is the state of the art to capture call stacks when the frame pointer is absent. We applied two important optimizations to `libunwind` in order to accelerate the backtrace operation even more. First, we disabled system calls that blocked signals while shared data structures were accessed using the `-enable-block-signals=false` build flag. Disabling signals is necessary to guarantee the reentrancy of `libunwind`. However, since we always invoke the library within `SIGPROF` signal handlers, and since Linux doesn't allow nested signals of the same type unless the `SA_NODEFER` option is set when the signal handler is installed,[4] it is valid to remove that protection in our case. Second, we replaced a loop that restored the value of all registers by more efficient code that restores only the registers that are really needed.[5]

A trade-off of CPU stack sampling, compared to instrumenting each function entry and exit, is that we can miss call stacks that have a short duration. Also, we cannot estimate the time spent in a given call stack with a better precision than the sampling period. Fortunately, for call stacks that have a long duration, the relative error is small. Since the goal of TraceCompare is to diagnose significant performance variations, we don't find the trade-offs of CPU stack sampling to be problematic.

The call stack events generated by our library contain sequences of addresses, instead or human-readable function prototypes. This reduces the tracing overhead and the size of the trace. However, for analysis purposes, full symbols are required. Therefore, we record events that provide the base address of dynamic libraries. Those allow us to transform captured addresses into library offsets. Then, we parse ELF binaries offline, to retrieve function prototypes.

3.3 Building a Task Executions Database

We described how the data required by our analysis can be collected through tracing. We now explain how to process this data to build a database of task executions. The ECCT data structure introduced in Section 3.1 is used to represent the performance characteristics of each task execution in the database.

The TraceCompare builder reads the events of a trace in chronological order. When it encounters an event associated with the beginning of an execution, the current timestamp and current thread are saved in a list of pending executions. Then, when the event marking the end of the execution is encountered, the following steps are performed:

1) Compute the critical path of the execution.
2) Generate an ECCT for the execution.
3) Compute global execution metrics.
4) Insert the execution's ECCT and metrics in a database.

4. http://man7.org/linux/man-pages/man2/sigaction.2.html
5. https://github.com/fdoray/libunwind/tree/minimal_regs

3.3.1 Critical Path

TraceCompare computes the critical path of an execution using the algorithm proposed by our group. The inputs of the algorithm are start and end points, both (*thread id*, *timestamp*) pairs, and a graph of dependencies between threads. The output is a list of segments belonging to the critical path of the execution, each characterized by a thread id, a thread status, a start timestamp and an end timestamp. The thread status takes one of the following values: running, preempted, interrupted, waiting for another thread or waiting for the operating system (block device, network, timer or user input).

The graph of dependencies between threads is built incrementally while the trace is read. Only nodes generated by events that occurred during an execution are required to compute its critical path. Therefore, the critical path of an execution can be computed as soon as its end event is encountered (before the full trace has been read).

3.3.2 ECCT

The next step of the analysis is to generate the ECCT of the execution from the stacks that appear on each segment of its critical path. Backtracking to reread events of the trace, every time the end of an execution is encountered, would be highly inefficient. Instead of that, we accumulate information in a state history while we read the trace. When the time comes to generate the ECCT of an execution, the required information is obtained by querying the state history. Efficient State History Trees were proposed by our group and described in Section 2.2. Using them for comparison of executions is an original contribution. The evolution of the following attributes is tracked in our state history:

- `threads/[tid]/cpu`: The CPU on which the thread `tid` is running.
- `threads/[tid]/stack`: The call stack of the thread `tid`.
- `cpus/[cpuid]/thread`: The thread running on the CPU `cpuid`.
- `blockdevice/threads`: Threads waiting for a block device.

The algorithm presented in Fig. 5 and Fig. 6 generates the ECCT of an execution from its critical path and a state history. Previous work has explored the idea of exploiting wake up events to retrieve call stacks that are on the critical path of a task execution [10]. However, this algorithm is the first to also take advantage of CPU and block device usage events to paint a more complete picture of the interferences between multiple threads. Furthermore, concatenating call stacks from multiple threads to describe precisely the context of each source of latency in a task execution is a novel approach.

3.3.3 Execution Metrics

It is convenient to be able to filter executions based on a wide range of metrics when performing a comparison. For example, we might want to compare executions that generated a lot of page faults against those that generated a few, or executions that allocated a lot of memory against those that allocated a few.

Depending on the events that are present in the source trace, we can keep track of the evolution of different metrics

Input: critical path C, state history H, stack of threads T, ECCT E
      $\triangleright$ T and E are empty in the first recursive call to this function.
1: **for all** $e \in C$ **do**
2:    UPDATETHREADSTACK(H, T, $e.tid$, $e.start$)
3:    **if** $e.status =$ running **then**
4:       $stacks \leftarrow$ Query H for stacks of thread tid in
           time range $[e.start, e.end]$.
5:       **for all** $s \in stacks$ **do**
6:          **if** T.SIZE > 2 **then**
7:             $s.val \leftarrow$ CONCATENATE($T[-2].stack$, thread, $s.val$)
8:          **end if**
9:          E.INSERT($s.val$, $s.duration$)
10:       **end for**
11:    **else if** $e.status =$ preempted **then**
12:       $cpuId \leftarrow$ Query H for CPU of thread tid at time $e.start$.
13:       $threads \leftarrow$ Query H for threads running on CPU $cpuId$ in
           time range $[e.start, e.end]$.
14:       **for all** $t \in threads$ **do**
15:          $e_{preempt} \leftarrow$ ($t.val$, running, $t.start$, $t.end$)
16:          GENERATEECCT($\{e_{preempt}\}$, H, T, E)
17:       **end for**
18:    **else if** $e.status =$ blockdevice **then**
19:       $threads \leftarrow$ Query H for threads using a block device in
           time range $[e.start, e.end]$.
20:       **for all** $t \in threads$ **do**
21:          $s \leftarrow$ Query H for stack of thread $t.val$ at time $t.start$.
22:          $s \leftarrow$ CONCATENATE($T[-1].stack$, block device, s)
23:          E.INSERT(s, $e.duration/threads.size$)
24:       **end for**
25:    **else**
26:       $s \leftarrow$ CONCATENATE($T[-1].stack$, $e.status$)
27:       E.INSERT(s, $e.duration$)
28:    **end if**
29:    UPDATETHREADSTACK(H, T, $e.tid$, $e.end$)
30: **end for**

Fig. 5. Function GENERATEECCT: Generates the ECCT of an execution.

Input: state history H, stack of threads T, tid, ts
1: **for** $i \leftarrow 0$ to T.SIZE $- 1$ **do**
2:    **if** $T[i].tid = tid$ **then**
3:       T.RESIZE(i)
4:    **end if**
5: **end for**
6: $s \leftarrow$ Query H for stack of thread tid at time ts.
7: **if** T.SIZE > 0 **then**
8:    $s.val \leftarrow$ CONCATENATE($threads[-1].stack$, thread, $s.val$)
9: **end if**
10: T.PUSH((tid, $s.val$))

Fig. 6. Function UPDATETHREADSTACK: Puts the enhanced stack of a thread at the top of a stack of threads.

in the state history. Then, when we process an execution, we compute metrics for each segment of its critical path. The values of each segment are then combined into a global metric, on which filters can be applied in our comparison view.

3.3.4 Database

Once the ECCT and the global metrics of an execution have been computed, they are added to the executions database. An interesting observation is that there is no need to duplicate the structure of the ECCT for each execution. A generic ECCT can be formed by the union of all the other ECCTs. Its structure along with the function prototype of each node are stored once in the database. Each execution record can then refer to the nodes of the generic ECCT to specify metrics. Because the ECCTs issued from different executions of the same task normally have a similar structure, this original technique greatly reduces the size of the database.

3.3.5 Garbage Collection

Two data structures are built incrementally while the trace is read: a graph of dependencies between threads and a state history tree. The size of both these data structures is proportional to the number of events read. For huge traces, storage of all this data might be a concern. Fortunately, the nodes added to the graph of dependencies, before the first event of an execution is read, are not needed to compute its critical path. Similarly, the intervals of the state history tree, that end before the beginning of an execution, are not needed to compute its ECCT or global metrics. Therefore, whenever the available memory is getting low, it is possible to discard the parts of the data structures that won't be needed to

process the pending executions. In addition to reducing memory usage, this strategy improves the time complexity of queries in both data structures. They become logarithmic with respect to the average size of an execution, instead of logarithmic with respect to the size of the analyzed trace.

3.4 Comparing Task Executions

The database of executions is used to populate a web-based view that allows interactive comparisons between groups of executions. The view is divided in three parts: the filters, the flame graph and the list of executions. An obvious benefit of providing a web-based view is that it facilitates collaboration. Anybody can easily share a link to its findings in a bug report, to help others fix the problem quickly. This is much more convenient than sharing huge trace files, let alone asking others to reproduce a complex bug themselves.

3.4.1 Filters

The aim of the filters is to allow the user to define two groups of executions to compare. They are presented as two columns of histograms. Each column is associated with a group of executions and each row is associated with a metric. Hence, each histogram shows the distribution of a metric within a given group. Initially, both columns are identical because all executions are included in both groups.

To filter the executions of a group, the user selects a region on an histogram.[6] For example, in Fig. 7, the left group contains executions with a total duration under 20 ms while the right group contains executions with a total duration above 50 ms. It is possible to set as many filters as desired.

Whenever a filter is added to a metric, the histograms of the other metrics are updated to only take into account the executions that belong to their group. This can reveal interesting correlations between metrics. For example, Fig. 7 shows a case where the CPU time of an execution is not correlated with its total duration, but is correlated with the number of bytes read from the disk.

3.4.2 Flame Graph

Differential flame graphs were introduced in Section 2.3 as a way to compare two CPU profiles. The second part of our comparion view reuses this visualization tool, but with some adaptations.

First, our differential flame graphs are built from ECCTs rather than from from CPU profiles. That means that they can reveal latencies caused by waiting threads in addition to

---

6. The implementation is inspired from http://square.github.io/crossfilter/.
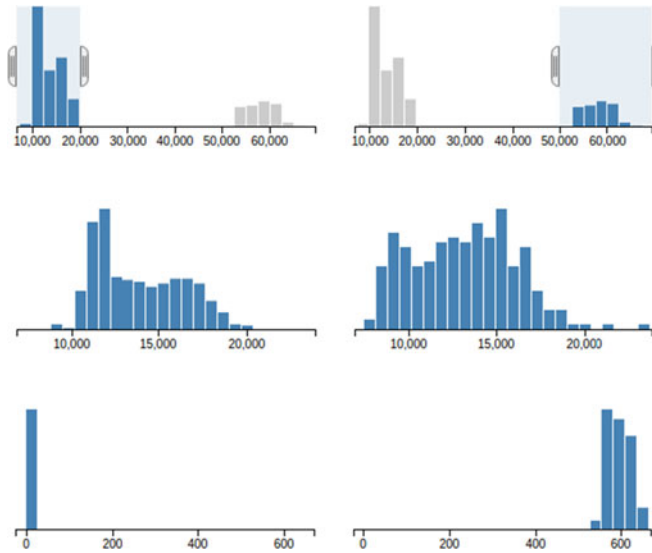
Fig. 7. Comparison filters.

Fig. 8. Differential flame graph.

latencies caused by CPU consumption. It also means that, as illustrated on Fig. 8, they can show call stacks for every thread on the chain of dependencies connecting a latency to its cause. To our knowledge, TraceCompare is the first tool to combine call stacks from multiple threads to produce a visual representation of a chain of dependencies between threads. Call stacks are important because they help application developers to find the source code responsible for the observed latencies.

Second, our differential flame graphs are used to compare two groups of executions instead of two individual executions. The width of a box is proportional to the mean time spent in a calling context for executions of the right group. The hue of a box is determined by the *t-score* of the difference between mean times spent in a calling context in the two compared groups. The *t-score* is computed as follows:

$$t = \frac{\bar{x}_2 - \bar{x}_1}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}}, \qquad (1)$$

where $\bar{x}_i$ is the mean time spent in a calling context for group i, s_i is the standard deviation of the time spent in a calling context for group i and n_i is the number of ECCTs in group i. Using a *t-score* instead of a simple difference between two values prevents colors to be applied to boxes in the presence of normal variations, which is a limitation of the original differential flame graphs.

Finding comparable nodes between multiple ECCTs is trivial: the paths connecting comparable nodes to the root of their ECCTs have to be identical.

If an ECCT doesn't include a calling context, we use zero as the time spent in that calling context.

Our differential flame graph is dynamically updated when filters are applied in the top part of the view.

3.4.3 List of Executions

The bottom of the view presents two tables with the source trace and timestamp of a few executions sampled from the two compared groups. This data enables the user to find executions of interest in another viewer such as TraceCompass to analyze the precise ordering of events.

3.4.4 Filtering Algorithm

TraceCompare uses a map-reduce algorithm to update histograms when filters are modified. "Map" functions are applied to all executions that are either added to or removed from a group following a filter modification. The functions determine in which bucket, of each histogram, affected executions belong. "Reduce" functions compute the new value of each histogram bucket.

A sorted index is kept for all metrics on which filters can be applied. Also, a filter is always specified as a range of accepted values for a given metric. Therefore, to update a filter, we simply find the first affected execution in the index and we traverse subsequent executions until we find the last affected one. This operation is performed in $O(logN + M)$ time, where N is the total number of executions and M the number of affected executions.

The user updates filters by dragging a handle on a histogram. Therefore, M is typically small at each mouse event and the views are updated in a few milliseconds. The update takes a little bit more time when a lot of executions have a similar value for the filtered metric and are affected simultaneously. To keep the views responsive in that case, we could precompute the result of the reduction for segments of fixed size on each dimension.

TraceCompare uses the Crossfilter[7] implementation of this map-reduce algorithm. The algorithm has already been used to analyze multi-dimensional data sets of payment, meteorological and transportation data, but never to compare software performance metrics.

4 CASE STUDIES

This section presents four performance problems, encountered in real applications, along with a description of how TraceCompare can diagnose them. The first two case studies were made on test applications that we built to closely reproduce problems found in enterprise software. The two subsequent case studies show how TraceCompare was used to find *real* performance problems in MongoDB,[8] an open-source database software.

4.1 CPU Contention in a Real-Time Application

4.1.1 Problem Summary

A realtime task is scheduled to be executed at a frequency of 100 Hz. Another task is scheduled to be executed at a frequency of 30 Hz on the same CPU. The priority set for the second task (using `nice`) is higher than the priority set for the first task. However, the first task should be set to the highest priority. At regular interval, the deadline of the second task occurs while the first task is running, increasing its execution time.

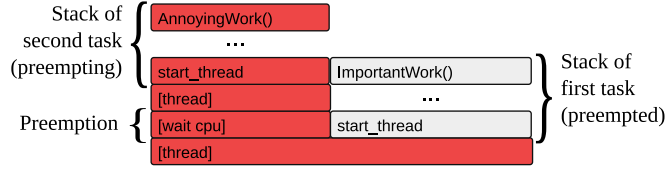7. https://github.com/square/crossfilter
8. https://www.mongodb.org

Fig. 9. Differential flame graph showing CPU contention in a real-time application.

4.1.2   Diagnosis

TraceCompare was used to compare slow executions of the first task against fast ones. The flame graph, partially reproduced in Fig. 9, immediatly revealed that the first task was preempted by the second during the slow executions. More interestingly, it showed call stacks from the second task. With these call stacks, it was clear where to look in the code to fix the problem. Without call stack events, the analysis tool could only have shown the name of the task that stole the CPU from the first task. Since threads are named after their parent, unless efforts are made to give them meaningful names, further investigation would have been needed to find out what was the purpose of the second task.

The histogram showing the "timestamp" metric of slow executions also provides useful information: it reveals that the problem is occurring at regular intervals.

A simple profiler could not have been used to diagnose this problem. Indeed, despite the fact that the second task causes the first task to miss its deadline, it does not account for a significant proportion of the total CPU consumption of the application. Being able to isolate slow executions to compare them against normal executions was indispensable in this case. Also, existing performance comparison tools (e.g., IntroPerf) are not able to identify preemption relationships between threads. Therefore, they could not have revealed what task was responsible for the slowdown observed.

4.2   Disk Contention in a Server Application

4.2.1   Problem Summary

A server has to read data from the disk to fulfill requests. At regular intervals, a request is about five times slower than usual.

4.2.2   Diagnosis

TraceCompare revealed that the additional latency found in the slow requests came from abnormaly long `read()` system calls. With a tool whose scope is limited to a single thread, it would have been hard to pursue the analysis further. Fortunately, TraceCompare was able to leverage the information from a trace of the whole system to show precisely which other threads used the disk during these long system calls and what were their call stacks.

TraceCompare highlighted a single thread that always used the disk during the slow executions, but never during the fast executions. That thread was performing an `fsync()` system call after having written to a log file. With this information, fixing the problem was straightforward.

Existing tools rely either on "wake-up" events from the operating system kernel [10], [31] or on custom instrumentation [11] to identify explicit dependencies between threads. However, this doesn't work to identify implicit
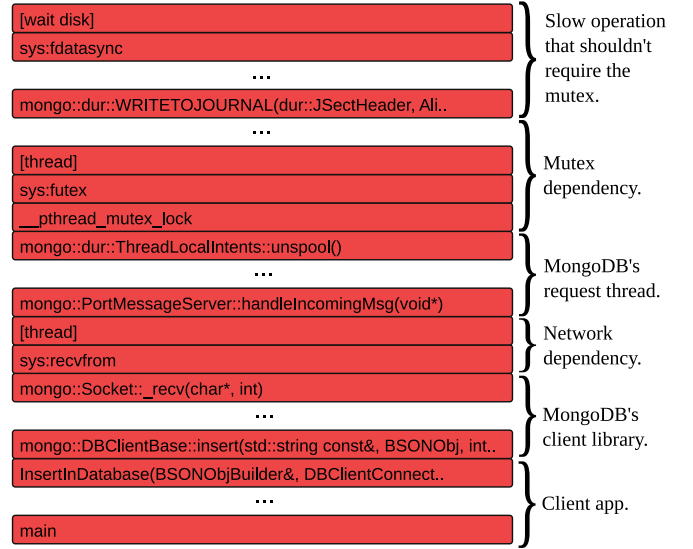

Fig. 10. Differential flame graph showing a lock contention in MongoDB.

dependencies, such as the disk contention described in this case study. By querying disk usage data from a state history, TraceCompare was able to show an interference between threads that no other tool could have identified.

4.3   Lock Contention in MongoDB

4.3.1   Problem Summary

A client application generates data and inserts it into a MongoDB database (version 2.5.4). Most of the time, the whole operation takes around 10 ms. However, a fraction of the time, the operation takes more than 100 ms. Data is generated at less than 1 MB/s.

4.3.2   Diagnosis

The database insertions were performed serially, so it was unlikely that the performance variation was due to CPU or lock contention. Also, the MongoDB's documentation states that insert operations don't wait for data to be committed on disk before returning (a dedicated command is available to wait for that). Therefore, we didn't expect disk latency to affect the performance of our application. Note that if our application had produced data at a higher rate, it would have been understandable that MongoDB throttled new insertions to flush its memory buffers.

The differential flame graph of TraceCompare, partially reproduced in Fig. 10, revealed that slow executions had waited for a mutex protecting access to a list of pending changes. During these wait times, the mutex was held by a journalization thread. This information alone doesn't lead to an obvious fix and could have been found by lock analysis with a tool such as Intel VTune.[9] However, TraceCompare sets itself apart from other tools by also revealing a function running on the journalization thread for almost all of the wait time. We analyzed the code of this function and discovered that it was not using the list protected by the mutex. Therefore, we wrote a simple patch to release the mutex before calling the costly function. We verified that the patch correctly fixed the serious performance

9. https://software.intel.com/intel-vtune-amplifier-xe

problem that we had uncovered and we submitted it to the MongoDB developers. All this was done without any prior knowledge of the MongoDB codebase.

This diagnosis was made with an unmodified MongoDB binary. We only inserted probes in our client application to delimit the task to analyze. To track the latency, from our client application to the journalization thread of MongoDB, TraceCompare had to match TCP packets exchanged between both applications, and to analyze wake-up events within the Linux kernel.

In this case study, the ability of TraceCompare to show distributed call stacks was critical to correctly identify the offending function on the journalization thread and to understand how it was related to the client application. In Section 2.2, we have seen that existing tools struggle to visually summarize the thread dependencies found in a large number of task executions.

4.4 Sleep in MongoDB

4.4.1 Problem Summary

Batch insert commands are sent to a MongoDB server (version 3.0.0 rc 10, WiredTiger storage engine). The commands are run in less than 700 μs most of the time. However, about 1 in 10 000 commands takes between 3 and 5 seconds to complete.

4.4.2 Diagnosis

TraceCompare made it easy to compare a group of fast executions against a group of slow executions. The comparison revealed that there was in fact two distinct sources of latency in our test scenario. Some executions were waiting for the disk within `pwrite64()` system calls while other were blocked by a timer. Since disk contention can be expected when a lot of data is inserted in a database, we decided to create a second filter to focus on the more suspicious timer contention.

The flame graph showed that the timer contention was caused by a sleep within a function responsible to obtain a hazard pointer to a page of data. We located the sleep call in the source code. It was accompanied by a comment explaining that if another thread was trying to evict the page from the cache at the same time, it was necessary to wait a while before retrying to obtain the pointer.

TraceCompare didn't show any dependant work being done during the long sleeps. In fact, we were able to confirm that both the CPU and disk activity were very low during the sleeps, by using the information provided in the tables at the bottom of the view to locate the slow executions in another trace analysis tool. At that point, we were confident that we had found a synchronization bug in MongoDB, along with a precise diagnostic.

Despite the fact that the sleeps caused huge delays, they didn't account for a significant proportion of the trace. They wouldn't have caught our attention if we had used a profiler, which only shows global statistics. Having the ability to create groups of executions using multiple custom filters was a key feature to diagnose this problem. Also, having a trace of timestamped events allowed us to obtain resource usage statistics for specific time ranges.

5 PERFORMANCE ANALYSIS

We conducted experiments to evaluate the tracing overhead of our solution as well as the time required to analyze a trace in order to build a database of executions. It is crucial to be able to record all the events required by TraceCompare with low overhead. Indeed, it is unlikely that our solution would be used on production systems if it decreased performance noticeably. Rare bugs that occur under very specific conditions would therefore be out of its reach. Also, having a low overhead ensures that the behavior captured in the trace is similar to that of the uninstrumented system. On the other hand, a fast analysis time helps system administrators respond to problems quickly.

5.1 Environment

All experiments were run on a computer with a quad-core Intel Core i7-3770 CPU running at 3.4 GHz, 16 GB of DDR3 memory and a 7200 RPM hard drive. The frequency scaling governor of the CPU was set to "performance" to ensure that it always runs at its maximum frequency. The Linux kernel version was 3.13.0-49 while the LTTng version was 2.6.0.

5.2 Cost of Tracing

We wanted to measure the overhead incurred by each step of the generation of our new `syscall_stack` event. To do so, we first created a microbenchmark that invokes the `getpid()` system call 100 million times and we timed its completion time when different parts of our kernel module were enabled. We repeated each experiment 20 times and always got a standard deviation of less than 0.5 percent. The `getpid()` system call was chosen because of the low variability of its duration.

Registering empty probes to tracepoints located at the entry and exit of system calls incurs a cost of 96 ns per system call. This is because the Linux kernel has to go through a slow path whenever there are probes registered to system call tracepoints. Adding our code to detect long system calls in the probes brings an additional cost of 42 ns. These costs apply to all system calls, even those that turn out to be faster than the threshold for long system calls. When a long system call is detected, a signal is sent to the application so that it captures its call stack. Sending the signal and executing an empty signal handler takes 1.2 μs.

Next, we microbenchmarked the operations executed within the signal handler. We used microbenchmarks that repeat the measured operation 100 million times. We repeated each experiment 20 times and always got a standard deviation of less than 2 percent.

The total time to unwind the stack depends on the number of frames currently on the stack. If it's the first time that a stack frame is encountered, it takes 326 ns to process it. This is 20 percent faster than without our custom optimizations. Because decoded unwinding rules are cached, subsequent processing of the same stack frame takes only 7 ns. There is also a base cost between 30 and 700 ns, depending on whether all stack frames are handled by cached rules. Once a stack has been captured, it is written in an LTTng buffer, an operation that requires 175 ns.

`cpu_stack` events are generated from a signal sent at the expiration of a timer. Enabling the timer has a negligible

TABLE 2
Execution Time of Test Programs (in Seconds)

	prime		babeltrace		find		mongod	
	Mean	Dev.	Mean	Dev.	Mean	Dev.	Mean	Dev.
Base	9.11	0.00	17.57	0.17	14.39	0.18	9.91	0.28
Traditional tracing of system calls	9.11	0.00	17.81	0.14	14.99	0.28	10.72	0.06
Tracing stack events	9.12	0.00	17.74	0.15	14.67	0.35	10.09	0.20
Tracing stack and critical path events	9.13	0.00	17.76	0.15	15.08	0.30	10.76	0.05

overhead. The execution of the signal handler takes the same time as for `syscall_stack` events.

Finally, we wanted to determine how our instrumentation affects typical applications. Table 2 shows the time required to run four programs under different tracing scenarios. The overhead in percentage is reported in Table 3. Each test case was run 100 times.

The "Traditional tracing of system calls" scenario is a baseline to compare with the "Tracing stack events" scenario. Indeed, our custom `syscall_stack` event reveals long system calls with a significatively lower cost than the traditional system call events. TraceCompare requires a trace recorded under the "Tracing stack and critical path events" scenario.

`prime` is an application that computes a list of prime numbers less than 10 000. It does no system calls, so the overhead is mostly due to the statistical CPU profiling. `babeltrace` is a single-threaded program that reads CTF traces. It does a lot of system calls to open trace files and output results, but few of them are long enough to trigger the generation of a stack event. The 1 percent overhead is mostly due to the kernel module tracking the duration of each system call. `find` is a command-line tool that searches files recursively in the file system. It spends most of its time blocked on block device requests. Each request generates interrupts and context switches, which contribute to the 5 percent overhead of tracing critical path events. `mongod` is an open-source database server. It was tested using the client application presented in Section 4.3. The interactions between multiple threads through synchronization primitives and TCP packets explain the 9 percent overhead.

5.3 Cost of the Analysis

The time required to build the database of executions for each of the case studies presented earlier is reported in Table 4. This metric is correlated with the size of the trace rather than its duration. It is always less than three times the time required to read the trace with `babeltrace`[10] (a tool that just decodes the events of a trace).

The table also shows that the size of the generated database is on average 10 percent of the size of the source trace. Yet, it contained enough information to perform the precise diagnosis that we presented earlier. By keeping a database of executions rather than voluminous traces, system administrators could save storage space without sacrificing the convenience of being able to analyze the behavior of a system over a long period of time.

10. http://git.efficios.com/?p=babeltrace.git

6 FUTURE WORK

Our comparison tool has hard-coded rules to identify latencies caused by dependencies between threads, (provided that they use OS level synchronization), or by CPU or disk contention. Its scope could be widened by adding logic to account for GPU contention, CPU contention across virtual machines, or user space synchronization through standard libraries. It is however impossible to handle out of the box the specificities of each application (e.g., custom task queues, custom IPC through shared memory). The declarative language used for trace analysis in [32] could be extended to allow the expression of these specificities.

We used TraceCompare to analyze performance regressions between multiple versions of the same program. Renamed functions can easily be dealt with by allowing the user to provide a file, mapping old function names to new names. However, the biggest challenge comes from major code refactoring, during which functions can be splitted or merged. Static analysis techniques exist to identify cloned syntactic blocks between multiple source files [33]. Techniques also exist to retrieve a mixed stack of function calls and syntactic blocks for a thread [34]. These techniques could be combined to allow a meaningful comparison between traces recorded on different versions of the same program, even after a major refactoring.

Our user space tracing library can only capture the stack for ELF binaries. It should be extended to properly handle programs written in a dynamic language and JIT compiled.

TABLE 3
Overhead of Tracing

	Overhead (%)			
	prime	babeltrace	find	mongod
Traditional tracing of system calls	0.0	1.4	4.1	8.2
Tracing stack events	0.1	1.0	1.9	1.8
Tracing stack and critical path events	0.2	1.1	4.8	8.6

TABLE 4
Cost of the Analysis

	Trace		Database	
	Recording Time	Size (MB)	Build Time	Size (MB)
4.1	13 min 26 s	52	17 s	7
4.2	22 min 42 s	86	20 s	7
4.3	2 min 38 s	31	7 s	3
4.4	5 min 11 s	129	32 s	13

It would also be useful to allow user space processes to associate metrics with the current execution (e.g., number of returned results). Filters on these metrics could then be used to specify the execution groups to compare with more granularity.

Finally, it would be interesting to assess the ability of TraceCompare to unveil security problems. For example, we can imagine that TraceCompare would be helpful to diagnose a performance degradation due to crafted input introducing long collision lists in a hash table.

7 CONCLUSION

In this paper, we presented TraceCompare, a new tool that facilitates the detection and diagnosis of performance variations between multiple executions of the same task. While some tools focus on CPU consumption bugs (profilers and flame graphs) and other focus on OS interaction bugs (OS tracers and IntroPerf), we have shown that TraceCompare works well with a wide variety of performance problems.

A novel technique was proposed to capture the user space stack of programs at key moments. The stacks are recorded along with kernel events using the LTTng tracer. We showed how to combine all these events to generate a database of ECCTs, a new data structure to describe latency that occurs at multiple levels during task executions. Then, we introduced an intuitive GUI that allows a user to specify groups of executions, using custom filters, in order to visualize their differences. Differences are presented using histograms and enhanced differential flame graphs. Enhanced differential flame graphs show all latencies that are on the critical path of the analyzed task executions, along with their full context. The GUI is dynamically updated using an efficient map-reduce algorithm when the user sets new filters.

We detailed how our tool was able to discover performance bugs commonly found in enterprise software and in MongoDB, a popular open-source software.

Finally, we showed that the overhead of tracing the events, required by our comparison analysis, is always under 9 percent, which means that our solution can be used on production systems.

REFERENCES

[1] F. Doray, "Analyse de variations de performance par comparison de traces d'exécution," Master's thesis, École Polytechnique de Montréal, Montréal, QC, Canada, Jul. 2015.

[2] I. Ceaparu, J. Lazar, K. Bessiere, J. Robinson, and B. Shneiderman, "Determining causes and severity of end-user frustration," *Int. J. Human-Comput. Interaction*, vol. 17, no. 3, pp. 333–356, 2004.

[3] E. Nygren, R. K. Sitaraman, and J. Sun, "The Akamai network: A platform for high-performance internet applications," *SIGOPS Operating Syst. Rev.*, vol. 44, no. 3, pp. 2–19, Aug. 2010.

[4] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "HPCToolkit: Tools for performance analysis of optimized parallel programs," *Concurrency Comput.: Practice Experience*, vol. 22, no. 6, pp. 685–701, Apr. 2010.

[5] A. Montplaisir-Gonçalves, N. Ezzati-Jivan, F. Wininger, and M. R. Dagenais, "State history tree: An incremental disk-based data structure for very large interval data," in *Proc. Int. Conf. Soc. Comput.*, Sep. 2013, pp. 716–724.

[6] N. Ezzati-Jivan and M. R. Dagenais, "A framework to compute statistics of system parameters from very large trace files," *SIGOPS Operating Syst. Rev.*, vol. 47, no. 1, pp. 43–54, Jan. 2013.

[7] B. Gregg, *DTrace : Dynamic Tracing Oracle Solaris, Mac OS X, and FreeBSD*. Upper Saddle River, NJ, USA: Prentice-Hall, 2011.

[8] F. Giraldeau and M. Dagenais, "Wait analysis of distributed systems using kernel tracing," *IEEE Trans. Parallel Distrib. Syst.*, p. 1, 2015.

[9] 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 *Proc. ACM Symp. Cloud Comput.*, 2014, pp. 1–13.

[10] S. Han, Y. Dang, S. Ge, D. Zhang, and T. Xie, "Performance debugging in the large via mining millions of stack traces," in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 145–155.

[11] 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," *Google Res. Rep.*, 2010. [Online]. Available: http://research.google.com/pubs/pub36356.html

[12] The Chromium Authors, Performance profiling with the timeline. (2015). [Online]. Available: https://developer.chrome.com/devtools/docs/timeline

[13] F. Rajotte and M. R. Dagenais, "Real-time Linux analysis using low-impact tracer," *Adv. Comput. Eng.*, vol. 2014, p. 8, Jun. 2014, http://www.hindawi.com/journals/aceng/2014/173976/cta/

[14] B. Gregg. (2014, Nov.). Differential flame graphs. [Online]. Available: http://www.brendangregg.com/blog/2014-11-09/differential-flame-graphs.h tml

[15] D. Lee, S. Cha, and A. Lee, "A performance anomaly detection and analysis framework for DBMS development," *IEEE Trans. Knowl. Data Eng.*, vol. 24, no. 8, pp. 1345–1360, Aug. 2012.

[16] N. Joukov, A. Traeger, R. Iyer, C. P. Wright, and E. Zadok, "Operating system profiling via latency analysis," in *Proc. 7th Symp. Operating Syst. Des. Implementation*, 2006, pp. 89–102.

[17] P. E. McKenney, "Differential profiling," in *Proc. 3rd Int. Workshop Model., Anal., Simul. Comput. Telecommun. Syst.*, 1995, pp. 237–241.

[18] 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," *SIGMETRICS Perform. Eval. Rev.*, vol. 42, no. 1, pp. 235–247, Jun. 2014.

[19] J. Trumper, J. Dollner, and A. Telea, "Multiscale visual comparison of execution traces," in *Proc. IEEE 21st Int. Conf. Program Comprehension*, May 2013, pp. 53–62.

[20] M. Weber, R. Brendel, and H. Brunst, "Trace file comparison with a hierarchical sequence alignment algorithm," in *Proc. IEEE 10th Int. Symp. Parallel Distrib. Process. Appl.*, Jul. 2012, pp. 247–254.

[21] A. Hamou-Lhadj, S. S. Murtaza, W. Fadel, A. Mehrabian, M. Couture, and R. Khoury, "Software behaviour correlation in a redundant and diverse environment using the concept of trace abstraction," in *Proc. Res. Adaptive Convergent Syst.*, 2013, pp. 328–335.

[22] S. Ghemawat. (2010, Jun.). Google Perftools CPU profile. [Online]. Available: http://google-perftools.googlecode.com/svn/trunk/doc/cpuprofile.html

[23] kernel.org. (2014, Jun.). perf: Linux profiling with performance counters. [Online]. Available: https://perf.wiki.kernel.org/

[24] J. Oakley and S. Bratus, "Exploiting the hard-working dwarf: Trojan and exploit techniques with no native executable code," in *Proc. 5th USENIX Conf. Offensive Technol.*, 2011, p. 11.

[25] F. Nybäck, "Improving the support for ARM in the IgProf profiler," Master's thesis, Aalto University, Espoo, Finland, Oct. 2014.

[26] J. Olsa. (2012, May). perf: Add backtrace post dwarf unwind. [Online]. Available: http://lwn.net/Articles/499116/

[27] G. Ammons, T. Ball, and J. R. Larus, "Exploiting hardware performance counters with flow and context sensitive profiling," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 1997, pp. 85–96.

[28] X. Zhuang, S. Kim, M. I. Serrano, and J.-D. Choi, "Perfdiff: A framework for performance difference analysis in a virtual machine environment," in *Proc. 6th Annu. IEEE/ACM Int. Symp. Code Generation Optimization*, 2008, pp. 4–13.

[29] M. Desnoyers and M. R. Dagenais, "The LTTng tracer: A low impact performance and behavior monitor for GNU/linux," in *Proc. Ottawa Linux Symp.*, 2006, pp. 209–224.

[30] S. Rostedt, "Using the trace_event() macro," Mar. 2010. [Online]. Available: http://lwn.net/Articles/379903/, consulted on Jan. 26, 2015.

[31] P.-M. Fournier and M. R. Dagenais, "Analyzing blocking to debug performance problems on multi-core systems," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 77–87, Apr. 2010.

[32] F. Wininger, "Conception flexible d'analyses issues d'une trace systéme," Master's thesis, École Polytechnique de Montrèal, Montrèal, QC, Canada, Apr. 2014.

[33] E. Merlo and T. Lavoie, "Computing structural types of clone syntactic blocks," in *Proc. 16th Working Conf. Reverse Eng.*, Oct. 2009, pp. 274–278.

[34] N. R. Tallent, J. M. Mellor-Crummey, and M. W. Fagan, "Binary analysis for measurement and attribution of program performance," *SIGPLAN Notices*, vol. 44, no. 6, pp. 441–452, Jun. 2009.

François Doray received the BEng degree in software engineering and the MS degree in computer engineering from École Polytechnique de Montrèal. He is now a software developer at Google in Montreal, Canada. His research interests include performance analysis tools and parallel systems. He is a student member of the IEEE.

Michel Dagenais is a professor at École Polytechnique de Montreal in the Computer and Software Engineering Department. His research interests include several aspects of multicore distributed systems with emphasis on Linux and open systems. His group has made several original contributions to Linux. He is a senior member of the IEEE.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.