Wait Analysis of Distributed Systems Using Kernel Tracing

Francis Giraldeau, *Member, IEEE* and Michel Dagenais, *Member, IEEE*

Abstract—We propose a new class of profiler for distributed and heterogeneous systems. In these systems, a task may wait for the result of another task, either locally or remotely. Such wait dependencies are invisible to instruction profilers. We propose a host-based, precise method to recover recursively wait causes across machines, using blocking as the fundamental mechanism to detect changes in the control flow. It relies solely on operating system events, namely scheduling, interrupts and network events. It is therefore capable of observing kernel threads interactions and achieves user-space runtime independence. Given a task, the algorithm computes its active path from the trace, which is presented in an interactive viewer for inspection. We validated our new method with workloads representing major architecture and operating conditions found in distributed programs. We then used our method to analyze the execution behavior of five different distributed systems. We found that the worst case tracing overhead for a distributed application is 18 percent and that the typical average overhead is about 5 percent. The analysis implementation has linear runtime according to the trace size.

Index Terms—Performance measurement, operating systems, tracing, reverse engineering

1 Introduction

A distributed and heterogeneous system is a set of threads running on multiple computers, and implemented in various programming languages. The hidden nature of the processing and the incompatibilities between runtime environments make the task of performance profiling and debugging more difficult. Our goal is understanding the elapsed time of a computation in such systems to improve the response time and to diagnose performance problems.

Popular profilers based on hardware counters sampling [1], [2], dynamic binary translation [3] and call-graph elapsed time are useful to identify code hot spots, but are limited in two ways. Firstly, instruction profilers do not take into account the time spent waiting. Secondly, they are restricted to the local host, which limits their use for distributed applications. Hence, the performance of each component must be analyzed independently.

Previous work considered instrumentation of libraries and middleware to monitor performance of specific distributed systems [4], [5], [6], [7], [8], [9], [10]. The resulting instrumentation provides straightforward performance measures, but is domain dependent and tied to a runtime environment. Considering the large number of languages, components and frameworks, the cost of instrumenting each of them is high. System call tracing was proposed as a less invasive instrumentation technique [11], [12], [13]. The request processing path of a distributed application is recovered by recording send and receive operations. These system calls are not sufficient in general because communication can occur from any kernel code, such as other system calls, interrupt contexts, and kernel threads. A technique based on recording network traffic was proposed to characterize the elapsed time between the client, the server and the network [14]. However, internal processing of endpoint machines is not visible using network events only.

Kernel tracing allows to observe wait occurring between threads. It works with unmodified executables and is system-wide, two properties important for actual heterogeneous distributed systems. By carefully choosing the instrumentation, low overhead and disturbance can be achieved. This paper aims to study methods providing meaningful representation of a broad range of actual distributed applications execution using kernel traces. The contributions are as follows :

- The design of the kernel instrumentation required for the analysis. The implementation is available as Linux loadable modules and works with an unmodified Linux kernel.
- A graph model of the system execution generated from the trace and the corresponding algorithm to extract the active path of a given task.
- Experiments on actual software to study the program behavior with regards to wait, according to host type, software architecture, and network conditions.
- Evaluation of the analysis cost of the runtime overhead and the trace processing.
- Two practical optimizations improving the analysis performance for actual traces.

Note that our goal is not to recover the flow of request in specific application protocols. This knowledge is outside the domain of the operating system, and we consider the data payload in network packets as a black-box. Also, because the method recovers the active path at the system level, if the runtime environment process multiple requests

- *The authors are with the Department of Computer Engineering, Polytechnique Montreal, Montreal, Québec, Canada. E-mail: {francis.giraldeau, michel.dagenais}@polymtl.ca.*

Manuscript received 17 Dec. 2014; revised 30 Sept. 2015; accepted 2 Oct. 2015. Date of publication 8 Oct. 2015; date of current version 20 July 2016. Recommended for acceptance by F. Mueller.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2015.2488629

simultaneously using the same task (such as with user-space thread), additional user-space instrumentation is required to identify processing of a specific request. The next section describes in detail the instrumentation and the analysis algorithms.

2 Analysis Architecture

The analysis is performed according to the following steps :

1) Start tracing on each relevant host;
2) Run the program of interest;
3) Stop tracing;
4) Gather the trace files on one machine;
5) Synchronize the traces;
6) Build the execution graph;
7) Compute the active path of the program of interest;
8) Display the result in the interactive viewer.

The method requires tracing simultaneously each host involved in the distributed processing to observe. Otherwise, the result of the active path may be incomplete if dependencies between events cannot be established.

We first review the essential operating system principles on which the analysis is based. Then, we discuss the trace synchronization method we use. Finally, we define and present the algorithm of the execution graph and the active path and present their corresponding algorithm.

2.1 Operating System Trace of a Distributed System

For our discussion, we consider that a task (or a thread) on a computer can be in four canonical states, namely `running` on a given CPU, `preempted` when ready but not executing, `interrupted` when an interrupt handler nests over the application code, and `blocked` when the task passively waits for an event and yields the CPU. All states other than `running` prevent the program from making progress and should be reduced whenever possible.

Interrupts can be trivially monitored by recording handler entry and exit. Simple statistics can be computed from these events, such as frequency and duration. Tracking interrupts allows to identify if an event is emitted from task or interrupt context.

Preemption mostly occurs when processors must be shared between tasks. The scheduler switches the running task on a given CPU when its quantum expires. The cause of the preemption can be assigned to the tasks running on the corresponding CPU because they affect the completion time of the preempted task. Preemption also occurs between the time a task becomes ready, after blocking, and the time it effectively executes.

Unlike other types of waits, `blocking` changes the control flow of the program. The behavior depends on the structure of the application. A task going to the blocked state is moved from the run queue to the wait queue and then the scheduler is invoked to yield the CPU. The key idea is that the event unblocking the task (thereafter referred as the wake-up event) indicates the cause of the wait, which is unknown *a priori* and is non-deterministic in general. We distinguish two types of blockings, that is when the wake-up occurs from another task in kernel mode, or from an interrupt. Task wake-up examples are contention on a mutex, empty or full pipe conditions, and other interprocess communication. By improving the performance of the subtask, the wait of the main task is reduced. In contrast, when the wake-up comes from interrupt context, the interrupt vector indicates the device after which the task was waiting, for instance, a timer or a disk. In particular, programs waiting for an incoming network packet are generally woken up from a network interrupt. By tracking the source of the packet related to the interruption, we can identify the emitter task. The reasoning is that if the network was faster or the task sent the packet earlier, then the blocking time of the receiver would have been reduced.

Consider for example the system call `select()`, that returns either because a file descriptor is ready or the specified timeout occurs. If data is written to the file by a local task, the wake-up source indicates which thread made the write. If the wake-up comes instead from the timer interrupt, it indicates that the timeout occurred. If the wake-up comes from the network interrupt, it means a remote task sent a message over a socket. Therefore, this mechanism is independent of the system call, its parameters or its return value.

One limitation of this approach is related to active waiting, such as spinlock and polling used in low-latency applications. Busy wait in user-space is not visible from the operating system. For distributed application, the network delays are usually an order of magnitude greater than the CPU speed, therefore it is reasonable to assume that most applications are blocking during the processing for efficiency.

We review briefly the behavior of the Linux operating system regarding the task states, the interruption context, and the network exchanges, which are the foundation for the active path analysis. Packet transmission and reception always occur in kernel mode, either from a system call (such as `send()` or `write()`), or a deferred interruption (thereafter referred to as softirq). The reception is done asynchronously inside softirq, and then any task waiting for the data is awakened. Packet transmission also occurs from softirq context. We observed that TCP control packets are sent immediately after packet reception and that TCP retransmissions are sent from the timer softirq. To prevent user-space starvation due to high-frequency softirq, the processing is deferred to the `ksoftirqd` kernel thread.

Fig. 1a shows the execution according to time of `netcat` transmitting a short string. The elapsed time is adjusted for proper display. The client establishes a TCP connection to the server, sends a short string and closes the connection. Eight messages are exchanged between the client and the server. The server blocks in `accept()` for an incoming connection. The client sends the synchronize packet and the server host acknowledges it directly from the interrupt handler, without the intervention of the server task. The client blocks in `select()` for the connection to be established. The server is then awakened when the handshake is completed. The client writes the data to the socket, closes the connection, and waits for the connection termination in `poll()`. The server blocks while the data is transmitted in `read()`, and finally closes the connection, which unblocks and terminates the client.

The execution path affecting the completion time is identified by following backward the source of the wake-up. The result wrt the `netcat` client is shown in (b). The last

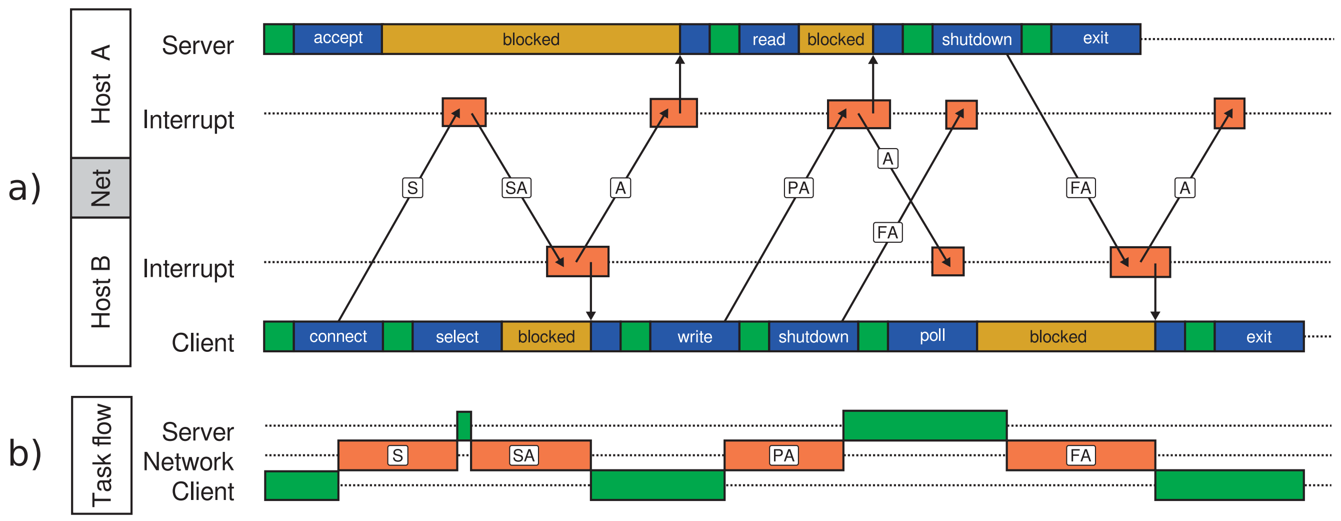

Fig. 1. Task state and TCP packet transmission.

`poll()` is unblocked by the `FIN-ACK` packet from the server, sent when the server calls `shutdown()`. When reaching the blocking in `read()`, the `SYN-ACK` is followed, and the corresponding `write()` of the client is attained. Finally, the blocking in `select()` is resolved by traversing the `SYN-ACK` and `SYN` packets.

The events required for the analysis and the instrumentation method are shown in Table 1. The instrumentation is implemented as loadable kernel modules. We use three instrumentation methods. The first method consists in adding a probe to an existing static tracepoint in the kernel. All scheduler and interrupt events are recorded in this way, except for the wake-up event. To reduce the wake-up latency, the default tracepoint `sched_wakeup` is executed on the destination CPU inside inter-processor interrupt (IPI). This has the effect of losing the source of the wake-up. We therefore define the event `sched_ttwu` using a kprobe hook to the function `try_to_wake_up()`. This function is called before sending the IPI at the actual wake-up call site. The third mechanism uses netfilter to register a hook for recording TCP packet headers.

For the prototype implementation on Linux, we use the Linux Tracing Toolkit next generation (LTTng) as the tracing facility [15]. This tracer provides efficient kernel tracing using a modern architecture, namely per-CPU trace buffers, monotonic timestamps with nanosecond precision and direct write to disk without an intermediate copy. In addition, its binary format is described using metadata, which simplifies defining new event types. Technically, other kernel tracers with precise and monotonic timestamps, such as Perf [1], could be used instead. The choice of LTTng is an implementation detail, and does not reduce the generality of the approach.

TABLE 1
Kernel Events Required for the Analysis

category	event	method
scheduler	sched_ttwu	kprobe
scheduler	sched_switch	tracepoint
interrupt	hrtimer_expire_entry	tracepoint
interrupt	hrtimer_expire_exit	tracepoint
interrupt	irq_handler_entry	tracepoint
interrupt	irq_handler_exit	tracepoint
interrupt	softirq_entry	tracepoint
interrupt	softirq_exit	tracepoint
network	inet_sock_local_in	netfilter
network	inet_sock_local_out	netfilter

2.2 Trace Synchronization

One challenge of distributed event analysis is the absence of a global clock. The analysis is sensitive to message inversion and requires partial order on network events, also called the clock condition. In addition, local interval durations must be preserved to accurately report the cause of delays. We detail in this section the rationale for the synchronization method.

The Network Time Protocol (NTP) is used to synchronize clocks according to a time server [16]. This protocol offers accuracy in the millisecond range. The synchronization error should be significantly lower than message latency to avoid message inversion. Therefore, NTP accuracy is insufficient to guarantee the clock condition of Ethernet network events, where the message latency is in the microsecond range. Furthermore, we do not want to interfere unduly with the system operation by requiring a specific time synchronization service or hardware support.

The Lamport's logical clock [17] is a classical method to order messages based on the happen-before relation. As its name suggests, the result is a logical clock for relative event order but does not provide a global, absolute elapsed time, as required for our analysis. Moreover, sending clock values with all messages is invasive.

The Controlled Logical Clock is an extension of the logical clock providing global time [18]. Unlike the logical clock, it uses existing message timestamps from the trace and is non-invasive. Inverted messages are shifted according to the minimum transfer delay. The events near inversion are also shifted using an amortization function. It can be used for lengthy traces with non-linear clock drift. The algorithm requires prior synchronization.

The convex hull synchronization algorithm [19] produces a linear relation (timestamps transform) between two clocks, that models the offset and the drift. Handling long traces can

$$\begin{tikzcd} q_0 \arrow[r, shift left, "f"] & q_1 \arrow[l, shift left, "f^{-1}"] \arrow[r, shift left, "g"] & q_2 \arrow[l, shift left, "g^{-1}"] \end{tikzcd}$$

Fig. 2. Example of a synchronization graph between three hosts.

be achieved by dividing in sub-intervals, each modeled using a linear function. This method was used to synchronize kernel traces with network packets [20]. To synchronize a group of nodes, the transform for a given clock is composed relative to a reference host. We chose this method because it preserves the local timings, it is non-invasive, and the clock corrections are more precise than with the Controlled Logical Clock, because the relation is computed over a large number of send-receive points (forming the convex hull) instead of locally at each packet reception.

The convex hull algorithm works as follows. For each pair of hosts, an x-y plane is built with two sets of points, either incoming and outgoing, wrt a reference host. The convex-hull of these two sets is approximated using minimum and maximum slopes dividing the two regions. The bisector of the two slopes is taken as the final approximation. The minimum convex hull requires at least two points in each set. The algorithm fails if a slope intersects a convex hull. This situation can occur for lengthy traces (e.g., longer than one hour), because of the physical non-linear properties of crystal clock oscillators, in which case dividing the trace into sub-intervals solves the problem.

We define the following relations for the linear timestamp transformation.

- function : $f(t) = mt + b$
- inverse : $f^{-1}(t) = t/m - b$
- compose : $f(g(t)) = f(t) \circ g(t) = m_1 m_2 t + m_1 b_2 + b_1$ with $f(t) = m_1 t + b_1, g(t) = m_2 t + b_2$
- identity : $f(f^{-1}(t)) = I(t) = 1t + 0 = t$

When more than two traces are synchronized, there must be a transitive transform to a reference host. We compute such transform using a directed graph where vertices represent hosts and where edges represent timestamp transforms. If the graph is connected, then a global time can be recovered. The graph is built by adding two edges for each transform, namely a forward edge representing the computed transform and a reverse edge with the inverse transform. The transitive transform is obtained by composing transforms for the path from the reference host to the peer host. A composed transform does not guard from inversion, and the partial order is not guaranteed. In practice, the composed error margin is lower than the network transmission and no inversion occurs.

As an example, consider the synchronization of traces from three computers q_0, q_1 and q_2, where packet exchanges occurred between (q_0, q_1) and (q_1, q_2) and the transform $f(t)$ and $g(t)$ respectively for these two pairs of hosts, obtained using the convex hull method. The resulting graph and each transform according to the reference computer are shown in Figs. 2 and 3 respectively.

2.3 Trace Analysis

The recovery of wait causes needs efficient navigation between related events. We achieve this with a directed acyclic graph (DAG) built from the synchronized traces. Then, the wait cause is recovered by traversing the graph backward.

Reference	Transform
q_0	$q_0 : I(t)$ $q_0 \rightarrow q_1 : f(t)$ $q_0 \rightarrow q_2 : f(t) \circ g(t)$
q_1	$q_1 : I(t)$ $q_1 \rightarrow q_0 : f^{-1}(t)$ $q_1 \rightarrow q_2 : g(t)$
q_2	$q_2 : I(t)$ $q_2 \rightarrow q_1 : g^{-1}(t)$ $q_2 \rightarrow q_0 : g^{-1}(t) \circ f^{-1}(t)$

Fig. 3. Resulting timestamp transforms according to the reference host for the synchronization graph example of Fig. 2.

More formally, we define an *execution graph* data structure as a two-dimensional doubly linked list, where horizontal edges are labeled with task states, and where vertical edges are signals between tasks (either a wake-up or a network packet). A vertex v represents an event and has a timestamp t, representing causality, such that every edge $v \rightarrow v'$ must satisfy $t \leq t'$. The Algorithm 1 details the trace to graph transformation. For simplicity, error handling is not shown and a data structure representing the machine state is implicitly defined per-host.

Algorithm 1. Execution Graph Construction

```
Input: synchronized trace $T \leftarrow \{T_1, T_2, \dots T_n\}$
Output: execution graph $G$
 1: $TASK \leftarrow \{initial\ tasks\}$
 2: $CPU \leftarrow \{p_0, p_1, \dots p_n\}$
 3: $IRQ \leftarrow \{interrupt\ stub\ tasks\}$
 4: $PKT \leftarrow \emptyset$
 5: for all event e ∈ T do                          ▷ Main procedure
 6:     now ← e.timestamp
 7:     if e is sched_switch then
 8:         LINK_HORIZONTAL(prev task, now, running)
 9:         LINK_HORIZONTAL(next task, now, preempted)
10:         set current task on $CPU$
11:     else if e is sched_ttwu then
12:         target ← e.tid
13:         source ← CURRENT_TASK()
14:         $v_1$ ← LINK_HORIZONTAL(target, now blocked)
15:         $v_2$ ← LINK_HORIZONTAL(source, now, running)
16:         link_vertical($v_1$, $v_2$, wake-up)
17:     else if e is interrupt_entry then
18:         push interrupt
19:         LINK_HORIZONTAL(CURRENT_TASK(), none)
20:     else if e is interrupt_exit then
21:         pop interrupt
22:     else if e is inet_sock_local_out then
23:         $tx$ ← LINK_HORIZONTAL(CURRENT_TASK(),
24:                 now, running)
25:         add (packet, $tx$) to $PKT$
26:     else if e is inet_sock_local_in then
27:         if packet match found then
28:             (packet, $tx$) ← remove match in $PKT$
29:             $rx$ ← LINK_HORIZONTAL(CURRENT_TASK(),
30:                     now, running)
```

```
31:         LINK_VERTICAL($tx$, $rx$, network)
32:       end if
33:     end if
34: end for
35: function CURRENT_TASK()                              ▷ Utilities
36:     $cpu$ ← smp_processor_id()
37:     if in_interrupt($cpu$) then
38:         return peek interrupt task of $cpu$
39:     else
40:         return task of $cpu$
41:     end if
42: end function
43: functionlink_horizontal ($task, ts, l$ )
44:     $tail$ ← last vertex of $task$ from $G$
45:     Create vertex $v$ with timestamp $ts$
46:     Create edge $tail[right] \rightarrow v[left]$ with label $l$
47:     return $v$
48: end function
49: function LINK_VERTICAL $from, to, l$
50:     Create edge $from[up] \rightarrow to[down]$ with label $l$
51: end function
```

The algorithm iterates over trace events and processes them according to their type. The `sched_switch` event adds two new edges to the graph, one for the previous task that was running, and the other for the next task that was preempted prior to run. The `sched_ttwu` event adds an edge to represent the blocking state of the target task and the task emitting the signal is necessarily running. Notice that the source may be either a thread or a per-CPU placeholder thread representing the interrupt context. Finally, a vertical edge is added from the source to the target with the wake-up label. The events `interrupt_entry` and `interrupt_exit` are managing the corresponding placeholder thread stack to account for nested interrupts. The network events are processed as follows. On transmission, a new vertex is added to the emitter task. This new vertex and its packet are added to the unmatched packet set. When the corresponding packet is found, a new vertex is created on the receiver, and a vertical edge with the label network is created from transmission to reception vertices. The transmit vertex is added immediately to the graph and before the receive counterpart is seen to ensures linear complexity. If both vertices were added when a packet match occurs, it would require a graph seek to insert the transmission vertex and this would increase the computational complexity.

We define the *active path of execution* as the execution path where all blocking edges are substituted by their corresponding subtask. The algorithm is shown in Algorithm 2 and works as follow. The states of the main task are iterated forward, and visited edges are appended to the active path. If a blocked state is found, the incoming wake-up edge is followed, and the backward iteration starts. In the backward direction, the visited edges are prepended to a local path. If an incoming packet is found, the source is followed backward. If a blocking edge is found while iterating backward, this procedure is repeated recursively. The backward iteration stops when the beginning of the blocking interval is reached, the accumulated path is appended to the result and the forward iteration resumes.

Algorithm 2. Active Path Computation

```
Input: execution graph $G$, task $T$, $v_s, v_e$
Output: path $P$
 1: $v \leftarrow v_s$
 2: while $v$ is not $v_e$ do                          ▷ Forward iteration
 3:     $E \leftarrow v.right$
 4:     append PROCESS($E$) to $P$
 5:     $v \leftarrow E.to$
 6: end while
 7: function PROCESS ($edge$)
 8:     if $edge.label$ is $blocking$) and
 9:             $edge.to$ has incoming vertical edge then
10:         return RESOLVE($edge$)
11:     else
12:         return $E$
13:     end if
14: end function
15: function RESOLVE ($edge$)
16:     $TMP \leftarrow \emptyset$
17:     $v \leftarrow edge.from$                            ▷ Follow incoming
18:     while $v.ts > edge.from.ts$ do                   ▷ Backward iteration
19:         $E \leftarrow v.left$
20:         if $v.down.label$ is $network$ then
21:             prepend RESOLVE($E$) to $TMP$
22:         else
23:             prepend PROCESS($E$) to $TMP$
24:         end if
25:         $v \leftarrow E.from$
26:     end while
27:     return $TMP$
28: end function
```

It is immediately apparent from the graph construction algorithm that the runtime complexity is $O(n)$, n being the number of events in the trace. The same observation applies to the active path computation, where only the connected components of the graph are traversed, and only once. We conclude that the sequential execution of both algorithms is linear according to the number of events. This property is verified experimentally using the actual implementation and is presented in Section 3.5.

3 Evaluation

We evaluated the system in three steps. First, we studied a single blocking call according to various operating conditions. We compare the execution on the local host to the execution in virtual machines and on physical machines. We show how the result changes according to the level of asynchronous processing of the application. We observe the effect of network latency and bandwidth on the result.

The second evaluation step focuses on analyzing five distributed systems under typical operating conditions. Use-cases were selected to represent the diversity of runtime environments used in the industry and includes C/C++, Java, Python, and Erlang.

The last step consists of evaluating the analysis cost. We measured the worst-case and average runtime overhead. We also studied the scalability of the analysis implementation.

For all experiments, the operating system is Ubuntu 14.04, running Linux 3.13 and LTTng 2.4. The machine used for local and virtual machine experiments is an Intel i7-4770, with 16 GB of RAM and an 1 TB SSD. The cluster used to run bare-metal experiments has four nodes, where each node is a dual-core AMD Opteron 246 processor, with 4 GB of RAM and 100 GB hard drive, communicating through a dedicated Gigabit Ethernet subnet. The analyzer is implemented in Java as Eclipse plug-ins. The code to reproduce the experiments is freely available on GitHub [21].

3.1 Effect of Host Type

We studied the effect of the host type on the analysis results. We compare three type of hosts configuration: a single host, two virtual machines, and two distinct computers. We compare the execution of an RPC request for each host configuration. We implemented `wk-rpc`, a minimal RPC implementation in C to control precisely the system calls performed. The remote procedure computes the amount of time specified by the client. The client connects to the server, writes the command and the parameter to the socket, and then calls read to retrieve the return value of the command. When the server-side operation is completed, it sends the return value to the client, which causes the read call to return.

When the client and the server are running on the same host, the operating system transmits network packets on the loopback interface using a softirq. It produces the same events structure as a physical interface. The synchronization stage is not necessary because only one clock is involved.

We executed the client and the server processes each in their own virtual machine running on the same host. We used Kernel Virtual Machine (KVM) as the hypervisor. In this case, traces must be synchronized, because each virtual machine scales the Time Stamp Counter (TSC) to nanoseconds independently.

The third experiment consists in executing the client and the server, each on their own physical computer. Compared to the virtual machine experiment, the communication is done through physical network interfaces.

The results of the three experiments are producing the same structural result as described in Fig. 1. The communication mechanism on the localhost interface works the same way as a remote socket for the analysis. We conclude that the abstraction we propose works for local and remote sockets and is independent of the host type.

We observed a difference between local and remote executions for large data transfers. When the client and the server run locally, the client may be preempted inside `sendto()` by the server executing `recvfrom()`. When run remotely, the client blocks in `sendto()` instead. The local preemption is however immediate from the trace.

Another case involves a user-space program detecting if its peer processes are on the same computer. The program may use shared memory instead of sockets for efficiency, which changes the local behavior as compared to the distributed execution. The OpenMPI library has this capability and is discussed in Section 3.4.5. If the wait related to shared memory synchronization is done through blocking system calls such as `futex()`, the wait dependencies between threads is taken into account by the proposed method, without the need to trace accesses to shared memory itself.

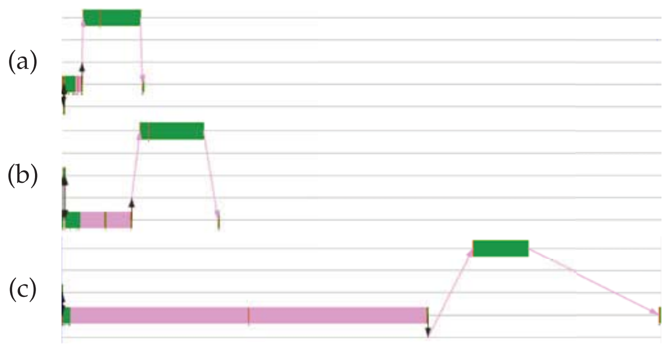

Fig. 4. Active path of `wk-rpc` according to the network latency. The execution in (a) has natural network latency, in (b) the latency is set to 10 ms and in (c), the latency is set to 100 ms.

3.2 Effect of Network Conditions

We used traffic shaper tool `tc` to increase packet transmission latency for the `wk-rpc` synchronous remote procedure call. The client and the server are running inside virtual machines, and the traffic shaping is applied to the virtual network interface on the host operating system. Fig. 4 shows the three executions for natural latency in (a), a latency of 10 ms in (b) and 100 ms in (c). For each execution, the server task is above the client.

In the graphical representation, the green intervals indicate CPU processing, the pink intervals are unresolved network wait, and pink edges are matched network packets between traced hosts. In addition, back edges are used for local processes wait and clipped intervals. Each execution begins with two network intervals in pink, representing the DNS resolution prior to the connection. These intervals are not resolved because the UDP packets and the DNS server are not traced. If the data was available, the DNS timing could be recovered, but it is left as future work. The second part of the execution shows, as expected, that the proportion of the network transmission increases according to latency.

We also observed the effect of available bandwidth on the behavior of a large network transfer. This experiment involves the `apache` web server and the `wget` client, where traffic shaping is used to limit the bandwidth. We observed that due to the TCP window and the fact that the processing time of the server is low, the transmission delay is greater than the associated blocking window, even when the bandwidth is not limited. The analysis accurately reports that almost all the wait time is caused by the network delay.

3.3 Effect of Asynchronous Processing

Asynchronous processing is a computation occurring simultaneously with input and output [22]. We simulate asynchronous processing in the client of `wk-rpc` using a busy-loop between sending the command and receiving the result. The transmission of such a small message does not block, allowing the busy-loop to proceed. The effect is to reduce the blocking window of the subsequent read call.

Fig. 5 shows the active path of the client according to time (server process above client process). Asynchronous levels are 0, 50 and 100 percent respectively in (a), (b) and (c). When synchronous processing is used, the blocking

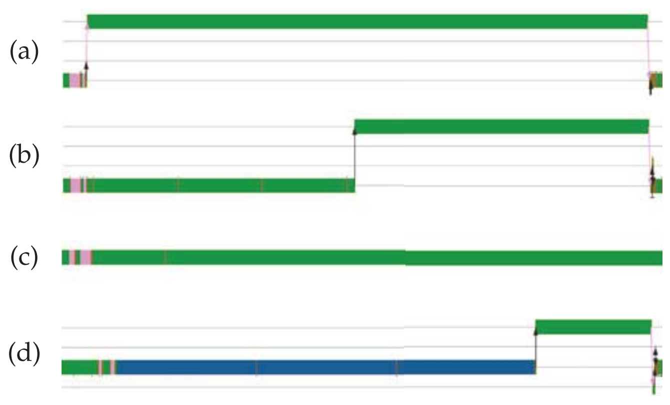

Fig. 5. Active path of `wk-rpc` according to asynchronous processing level of 0 percent in (a), 50 percent in (b) and 100 percent in (c), and asynchronous processing based on event loop in (d).

window reveals the entire server processing related to the request. This window is reduced proportionally to the amount of asynchronous processing, until the point where the process does not block. In this situation, no change in the control flow occurs in the active path.

Another type of asynchronous processing consists in an event loop to keep a single thread responsive, despite long blocking waits. A typical example of an event loop is shown in Fig. 5d. The event loop is implemented with the `poll()` system call, blocking for a resource to become ready up to a maximum timeout. The timeout period is set to 16 ms, corresponding to the screen vertical sync of 60 Hz and simulating the periodic refresh of a graphical user interface. The execution contains four intervals, where the first three (blue) are timeouts and where the last one is unblocked by the server's reply. The resulting active path associates wait time to the server only for the last blocking interval of the event loop. Because a large blocking window is decomposed into multiple arbitrary small timeouts, the control flow of the active path changes only for the blocking window related to the completion of the background request. Assuming uniform probability of event inter arrival while blocking, the resulting blocking window will not reflect the actual wait for the resource. Better handling of this execution pattern may be a direction for future work. Nonetheless, the analysis shows the cause of the delay at the system level in the case of a missed deadline.

3.4 Use-Cases

3.4.1 Java RMI

The Java Remote Method Invocation (RMI) is a framework to access objects on different computers over the network. We traced a classical example of RMI, where a client invokes a method on a remote server to compute the value of π with a given decimal place [23]. When the server starts, it registers the compute engine object to the `rmiregistry`. The client contacts the registry and obtains a reference to a proxy to access the remote object. The active path of the client is shown in Fig. 6. The client waits two times for the registry and three times for the compute engine server. The last interval represents the actual computation of π. Java RMI is synchronous by nature, and the method produces accurate results in this condition.

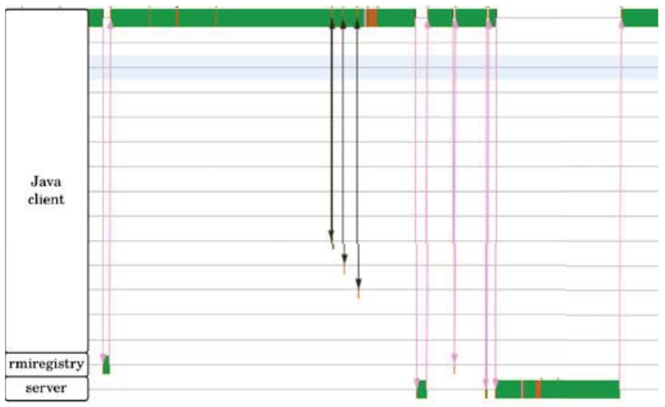

Fig. 6. Example of Java RMI compute engine execution.

3.4.2 Network Share

A remote file system is a storage device accessed through the network. The operating system handles transparently the I/O for the applications, either on a local drive or on a remote server. This experiment is about evaluating whether the wait for the file system is correctly recovered by our method. The experiment consists of a Samba server, providing a CIFS network share, and a client mounting and accessing files on this share.

Fig. 7 shows the execution of the remote directory listing, where the processing of the `smbd` server is visible. The program `ls` waits twice for the file server. The first wait is in `newstat()` to get the file attributes and the second is in the `getdents()` system call returning the directory entries. These two system calls are actually sending network packets, because of the virtual file system implementation. In other words, even the most trivial program `ls` can indirectly be a distributed program. The correct active path is obtained because no assumption is made about which system call could send network packets. Another interesting finding concerning the inner working of the kernel is that the `cifsd` daemon receives the answer from the server and then wakes up the main client for both calls. This observation is only possible because the tracing is system wide.

3.4.3 Web Application

We observed the result of the analysis for a simple, yet actual and unmodified web application. We used the Poll application of the Django project [24], a typical and popular Web framework written in Python. The user's vote is simulated with a non-interactive script using the python library `mechanize`. The voting process has three steps. First, the client performs a get request to download the form, then transmits the data using post, and finally the client is redirected to the poll result page. The application is deployed as WSGI using Apache HTTP server and PostgreSQL database.

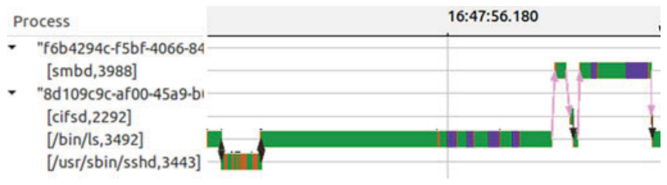

Fig. 7. Example of a CIFS remote directory listing.

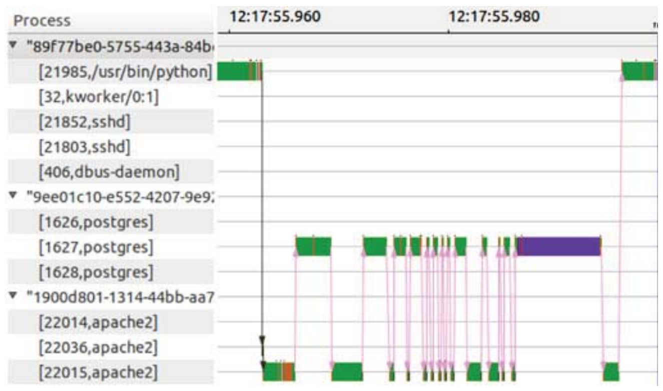

Fig. 8. Execution of the post HTTP request across the client, the web server and the database.

Each tier runs on its own KVM instance and is traced while the vote occurs.

The result of the post is shown in Fig. 8. The client connects to the HTTP server, sends the POST data, and waits for the reply. The server immediately dispatches the request to a worker thread. The control flow changes frequently between the apache worker thread and the database. Near the end of the request, the postgres process performs a call to `fdatasync()` (purple), blocking until all dirty pages are flushed to permanent memory and is related to the SQL update statement.

3.4.4 Erlang Service

We verified that the proposed method works for the Erlang runtime. We implemented a small echo server and its corresponding client in Erlang. The system was deployed in two virtual machines. The experiment consists of ten round-trips between the client and the server. The result is shown in Fig. 9. The client blocks for the answer from the server and the active path works for Erlang distributed processes. In this execution instance, the last round-trip does not extend to the server, unlike the nine previous ones. This is due to a smaller blocking window that occurred because the client was preempted by another task just prior to blocking.

3.4.5 MPI Computation

We tested our method for an MPI program using an imbalanced parallel computation. The function `MPI_Barrier()` is called at the end of a cycle to force the synchronization between distributed threads. We found that the OpenMPI barrier is implemented with a busy wait and appears as normal processing from the kernel perspective. Busy-wait reduces the latency by avoiding to invoke the scheduler, but at the cost of reduced resource efficiency. This design decision is justified where resources are dedicated and power consumption is a secondary concern, such as in scientific computing.

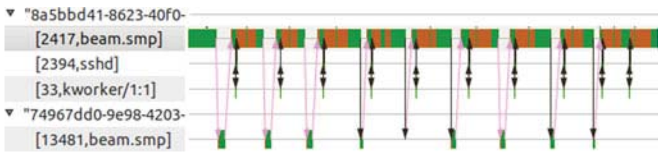

Fig. 9. Execution of the echo Erlang example.

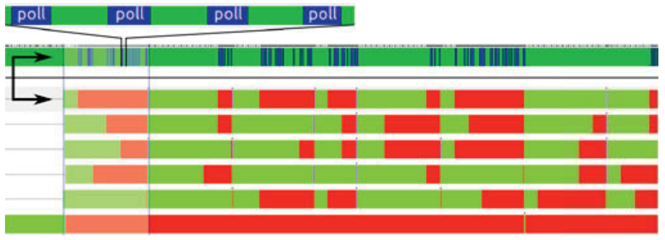

Fig. 10. MPI imbalanced computation execution. Below: the user-space trace displays the running (green) and waiting (red) of each thread. Above: the kernel trace corresponding to the first MPI thread. The top interval shows a zoomed in view of the active wait section. The first cycle of the computation is highlighted.

In this experiment, we measured the wait using a user-space instrumentation to record events before and after the barrier, in addition to the kernel trace. The instrumentation semantic is equivalent to the MPI Parallel Environment (MPE) monitoring library [25], and we verified that we obtain the same result with the Jumpshot viewer. The recorded data is displayed as time intervals and serves as an overlay to highlight the underlying kernel trace corresponding to the wait. Both user-space and kernel traces are using the same clock and timestamp transform and are therefore synchronized on a per-host basis. Fig. 10 shows the result for the execution of the MPI program involving two compute nodes. It allowed to pinpoint that the MPI barrier repeatedly performs non-blocking calls to `poll()` while waiting. We also studied the runtime parameter `mpi_yield_when_idle`, indicating to yield the processor when waiting. The parameter has the effect of adding a call to `sched_yield` in the busy loop rather than passively wait for the event.

Busy-wait is also observed in the implementation of OpenMP barriers. However, the default behavior can be modified by setting the environment variable `OMP_WAIT_POLICY`. When it is set to `PASSIVE`, the program blocks while waiting. This allows our method to highlight correctly the imbalance between threads without user-space instrumentation. The OpenMPI library could be extended similarly to support passive waiting.

3.5 Analysis Cost

We present the results of the analysis cost, both in terms of the tracing overhead and the implementation of the analysis algorithm.

The tracing cost includes the tracepoint instructions in the code path, and the execution of the trace consumer daemon, which is responsible for writing the event buffers to disk. This daemon is working as a background thread and does not increase latency of the workload if no preemption occurs between them.

The total tracing cost is proportional to the number of events produced and is about 200 ns per event. However, the overhead ratio is proportional to the event production rate. The first part of the experiment attempts to produce the highest possible frequency for a distributed workload, in order to measure the upper bound of the overhead. The second part focuses on the average overhead for a typical use case. Experiments were performed on the cluster hardware described in Section 3. The scheduler governor was set to `performance` instead of `ondemand` to reduce variations caused by processor frequency changes.

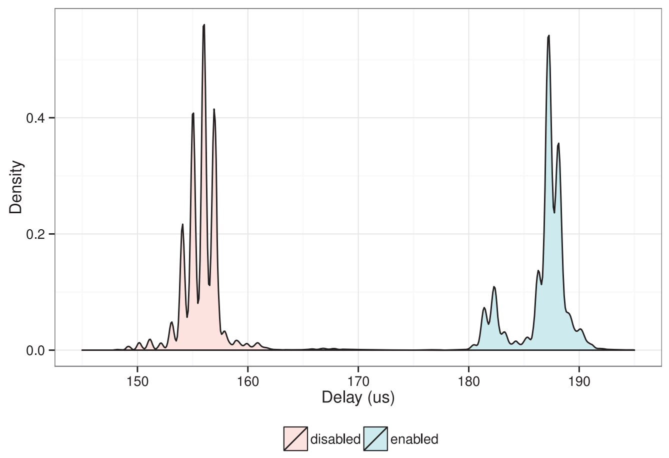

Fig. 11. Effect of tracing on request latency density for the wk-rpc benchmark. Tracing causes the distribution to shift to the right.

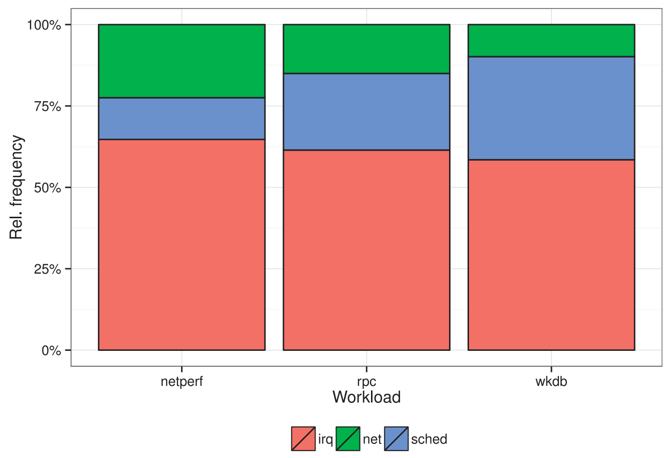

Fig. 12. Relative event frequency according to workload. Interrupts are the most frequent event type for all workloads.

The first experiment uses `netperf` to measures the effect on network throughput. Messages of size 16 kio are sent from the client to the server. The throughput measured is 424 Mbits/s with and without tracing. Tracing has no significant impact on the network throughput for this experiment. This result could be explained by the fact that the probe latency is hidden by the TCP transmission window.

The second experiment uses `wk-rpc` performing requests in a tight loop. The objective is to make round-trip queries at the fastest possible rate. The messages exchanged are only 32 bytes in size. We measured that tracing increased request latency by 18.3 percent, from 155.9 to $190.8\,\mu s$ (mean difference in change, 34.9 [95 percent CI, 34.7 to 35.1]; $p < 0.01$). Ideally, tracing should add a constant delay to requests. Fig. 11 shows the effect of tracing on the request delay density. The delay density without tracing is multi-modal, but its envelope is uni-modal and nearly normal. The main effect of the tracing is a nearly constant offset. A slight second mode appears at about $5\,\mu s$ below the main mode. A hypothesis to explain this phenomenon may be that different code paths are executed depending on runtime conditions. Further analysis is necessary to verify this hypothesis.

The third experiment uses `wkdb`, the Django Poll web application. The test measures the latency for loading and submitting the voting form. The request latency increased by 5.1 percent, from $116.3\,ms$ to $122.5\,ms$ (mean difference in change, 6.3 [95 percent CI, 4.8 to 7.7]; $p < 0.01$).

We measured the CPU usage of the tracing daemon to evaluate its relative impact on the system. We used the scheduling events from the trace to recover the average CPU usage for a window of one second during the peak workload activity. We found that the average CPU usage of the trace consumer daemon is comprised between 0.8 and 8.2 percent of one CPU, and is proportional to the event production rate ($R^2 = 0.91$).

To further categorize the cause of the overhead, we computed the event proportions according to their category, namely scheduler, network, and interrupt. The results for the three experiments are shown in Fig. 12. For all experiments, the most frequent events are interrupts. Reducing the tracing overhead for the active path analysis should therefore target interrupt events.

We studied the scalability of the Java implementation of the graph construction and path extraction algorithms. The input is traces of HTTP requests of the `wkdb` application, where the number of requests increases by powers of two, up to 2^{10} or 1,024. The largest trace has a size of about 500 MB. The measurements are performed with the trace loaded in the page cache to exclude the I/O time. From the chart of Fig. 13, we confirm that both algorithms have linear runtime according to the number of traced requests. A key question is the proportion of the trace reading n the graph construction. To establish this proportion, we measured separately the time to read the trace without event processing. This experiment indicates that nearly half the time to build the graph is related to reading the trace. The graph construction is by far the most expensive step of the analysis, being two orders of magnitude longer than the active path extraction. Once the graph is computed, the active path for the largest trace is returned under 0.5 s, which is practical to interactively explore the performance of different tasks.

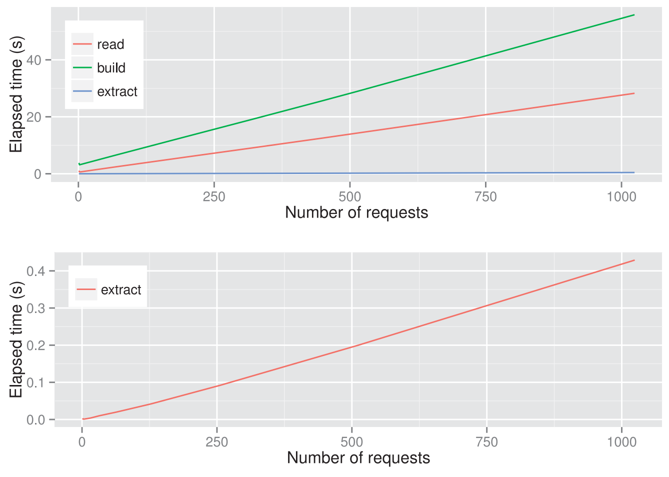

Fig. 13. Above: analysis time according to the number of traced requests for the different steps, namely the time to read the trace, build the graph and extract the active path. Below: the active path extraction is shown separately because of the difference in scale. Both algorithms have linear scalability and reading the trace accounts for roughly half the cost to build the graph.

3.6 Analysis Optimizations

We present two additional contributions improving the efficiency of trace synchronization for actual traces.

We observed a high peak memory consumption when synchronizing traces, caused by the packet matching algorithm. Packets are added to a hash map and removed once a match is found. However, traces may be highly skewed such that, in the worst case, all events from a trace are read before the other. This has the effect of keeping all packets in memory. Old unmatched packets cannot be removed because a match may be found later, and it increases the memory consumption for the processing. This problem does not occur if the traces are synchronized because matches would be found in a continuous manner and removed from the map.

To resolve this bootstrap problem, we take advantage of the fact that traces are started almost at the same time. The procedure starts by shifting traces to the origin of the reference trace by applying the timestamp transform

$$f(t) = t - (t_s - t_0),$$

where t_s is the start time of the trace and t_0 is the start time of the reference trace. Then, we perform a coarse synchronization using the convex hull algorithm on traces aligned at the origin. The coarse synchronization is stopped when the graph is connected with timestamp transform of precision greater than 1 percent. This condition is evaluated efficiently by using the weighted quick-union find [26]. When the precision threshold is reached, then a link between the two hosts is added. The coarse synchronization ends if the number of partitions equals one. Finally, the normal synchronization procedure can be performed.

In the coarse synchronization stage, memory usage is reduced by removing packets still unmatched after a long delay. These packets do not contribute to increase the precision of the synchronization and are discarded by the convex hull algorithm. There is no upper bound to the expiration delay, but the memory reduction is greater if the delay is smaller. The expiration delay of unmatched packets must be high enough to keep accurate packets. The minimal expiration delay is a function of the network transmission delay plus the delay to start all traces. The network delays can be estimated by the upper bound of the confidence interval at a given confidence level, and the actual delay to start all traces can be measured.

We evaluated the effect of the two-step synchronization with packet expiration using a trace of a three-tier web application. The trace size is 126 MB. For this experiment, the two-step synchronization peak memory usage was 319 times lower than performing the synchronization in one pass, while achieving the same precision. This drastic memory usage reduction is at the expense of reading twice a small proportion of the trace, which increases slightly the processing time (mean difference in change, 16.8 percent [95 percent CI, 14.5 to 19.1 percent]; $t(10) = 14.1$, $p < 0.01$).

The other enhancement is related to the computation of the transform. The transform slope is typically close to one because all timestamps are in nanosecond (the TSC scaling is already performed by the operating system). The timestamp in nanosecond from the epoch is in the order of 10^{18}. When multiplying these two numbers using double precision floating point arithmetic, the rounding produces non-monotonic timestamps. For this reason, an expensive 128-bit arithmetic is used for all events, specifically Java BigDecimal.

We designed a fast timestamp transform using integer arithmetic that guarantees monotonic time without overflow. The details are shown in Algorithm 3. It is based on the following equivalent rewriting of the linear function

$$f(t) = \frac{mc(t - t_0)}{c} + (mt_0 + b),$$

where the first term is the dynamic part of the timestamp, and the second term is constant over a period of time. The floating point slope is scaled by $c = 2^{30}$ to capture the nanosecond precision and then converted to an integer. The effect of the factorization is to reduce the width of the timestamp to the 32-bit range. The multiplication result of the scaled slope and time difference fits into standard 64-bit registers. The division itself is implemented using bit shift for efficiency. Overflow is avoided by recomputing the constant factor mt_0 if $t - t_0 > 2^{30}$ using large decimal, but it occurs only once per second of elapsed time in the trace. As for the offset b, the decimal part is simply dropped because the number is already in the nanosecond range, which is the highest precision of timestamps in the analysis.

Algorithm 3. Fast Timestamp Transform

```
Input: slope $m$, offset $b$, timestamp $t$
Output: transformed timestamp $t_x$
 1: $t_0 \leftarrow 0$                                  ▷ initialization
 2: $c \leftarrow (1 \ll 30)$
 3: $cst \leftarrow 0$
 4: $m_{int} \leftarrow (int)(m \times c)$
 5: $b_{int} \leftarrow (int)b$
 6: function FAST_TRANSFORM $t$
 7:     if ABS$(t - t_0) > c$ then                    ▷ Rescale
 8:         $cst \leftarrow t \times m + b_{int}$
 9:         $t_0 \leftarrow t$
10:     end if
11:     $t_{tmp} \leftarrow (m_{int} \times$ ABS$(ts - t_0) \gg 30$     ▷ Transform
12:     if $ts < start$ then                          ▷ Rectify sign
13:         $t_{tmp} \leftarrow -t_{tmp}$
14:     end if
15:     return $t_{tmp} + cst$
16: end function
```

We ran a micro-benchmark that computes 2^{25} consecutive timestamps with $200\,ns$ increments, a delay simulating the highest event frequency. Using the same machine as the previous test, the baseline transform function took $10.1\,\mathrm{s}$ to complete, compared to $65\,ms$ for the fast transform, representing an average speed-up of 155 times. We performed a benchmark to evaluate the overall effect of the fast timestamp transform on trace reading. Using the same trace as above, the fast transform reduces trace reading time significantly (mean difference in change, -20.8 percent [95 percent CI, -16.4 to -25.3 percent]; $t(10) = 9.2$, $p < 0.01$).

4 Future Work

We showed that, in order to reduce the tracing overhead of the analysis, optimizing the selection of interrupt events to be traced would yield the greater improvements. Entry and exit from interrupts are recorded to know if a given event occurs from interrupt and its vector. Tracing interrupts could be replaced by a per-event context carrying this information. It would reduce the number of events generated, but on the other hand some event sizes would increase slightly. Further study is required to determine the net impact of such optimization.

Concerning the analysis, the current approach has limited memory scalability. The graph size is proportional to the number of state changing events. Working in constant memory is required to handle traces larger than available memory. One solution would consist in computing a tree of blocking intervals incrementally in a bottom-up manner, and deleting unused vertices of the graph once a blocking is resolved. This method could work because a future event does not invalidate past computations.

During our experiments, we evaluated the effect of page cache conditions on the analysis time for traces stored on SSD. A cache cold run was only 2 percent slower than when the trace is in the page cache. Because the process is CPU bound, parallel processing of the trace may speed-up the analysis.

Another area for improvement concerns the ability to use the active path and relate it to the source code. The active path could be annotated using user-space tracing, call stack sampling and performance counters. Then, the developer would be able to relate the source code to the underlying system-level execution.

5 Related Work

User-space and domain dependent instrumentation was proposed to record request flow and thread interactions in a distributed system. The techniques differ in the instrumentation method and semantics.

In [10], the critical path of a distributed processing system is computed by propagating the CPU usage along the communication edges between processes. The path using the most CPU time limits the completion time. This technique works for CPU bound processing but does not account for the I/O wait time affecting the completion time.

Panappticon [4] combines user-space and kernel instrumentation to monitor the responsiveness of UI on Android. The execution model recovers asynchronous processing and I/O activity correctly. Its scope is limited to the client-side processing.

The tool `tcpeval` computes the critical path of TCP transactions [14]. TCP packets between two hosts are recorded. The relation between sent and received packets is established to build a packet dependence graph. The critical path is computed by traversing the graph backward from the last node to the root. The elapsed delay can be accounted to the server, the client or the network. The method is intrinsically limited to one communication flow between two hosts. The synchronization method used is NTP and packet inversion may occur in practice. The actual communicating tasks cannot be identified solely using a network trace. The internal processing of endpoints is not available, such that the elapsed time cannot be characterized between CPU, timer, I/O and processing in subtasks.

Request tagging [5], [6] consists in assigning a unique identifier to incoming requests, in order to identify processing related to it. The instrumentation targets server-side frameworks and is thus transparent to applications using them. The request flow can be augmented with system state. The analysis does not extend to the client.

MPE [7] is an interposition library that can trace calls to MPI. The Jumpshot interactive viewer uses the trace to display thread states and communications. The user can see the dynamics of thread execution according to time. The scope of the analysis is tightly coupled to the MPI domain.

BorderPatrol [11] uses library overloading to intercept calls to the C standard library functions. This instrumentation method is ineffective for statically linked programs. A more robust approach, used in vPath [12], intercepts system calls at the hypervisor level, but requires the workload to run in a virtual machine. Kernel tracing [13] has the same benefit without the virtual machine constraint. All methods based on the system call interface make assumptions regarding the underlying communication that do not hold for the general case.

6 Conclusion

Kernel tracing is a system-wide method to understand the actual latency of programs, independently of their runtime environment. We proposed a method to visualize the execution of distributed systems using scheduling, network and interrupt events. Thread blocking indicates a change in the control flow and the wake-up source identifies the wait cause. We demonstrated that this principle of operation can be applied on traces synchronized using the convex hull algorithm, and that the analysis produces insightful results for a broad range of distributed systems, with a moderate impact on the system.

Acknowledgments

This work was made possible by the financial support of Ericsson, EfficiOS and NSERC. We are grateful to Geneviève Bastien for code quality, Mohamad Gebai, Suchakra Sharma and Naser Ezzati for their comments, Matthew Khouzam and Alexandre Montplaisir for their help regarding Eclipse, and finally Mathieu Desnoyers and Lothar Wengerek for their precious advices.

References

[1] I. M. et al. (2014, Dec.). *Perf wiki* [Online]. Available: https://perf.wiki.kernel.org/

[2] W. E. Cohen, "Tuning programs with OProfile," *Wide Open Mag.*, vol. 1, pp. 53–62, 2004.

[3] N. Nethercote and J. Seward, "Valgrind: A program supervision framework," *Electron. Notes Theor. Comput. Sci.*, vol. 89, no. 2, pp. 44–66, 2003.

[4] L. Zhang, D. R. Bild, R. P. Dick, Z. M. Mao, and P. Dinda, "Panappticon: Event-based tracing to measure mobile application and platform performance," in *Proc. Int. Conf. Hardware/Software Codes. Syst. Synthesis*, 2013, pp. 1–10.

[5] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan, "Magpie: Online modelling and Performance-aware systems," in *Proc. 9th Conf. Hot Topics Operating Syst.*, 2003, Vol. 9, p. 15.

[6] B. Sigelman, L. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a large-scale distributed systems tracing infrastructure," *Google Research*, 2010.

[7] O. Zaki, E. Lusk, W. Gropp, and D. Swider, "Toward scalable performance visualization with Jumpshot," *Int. J. High Perform. Comput. Appl.*, vol. 13, no. 3, pp. 277–288, 1999.
[8] E. Thereska, B. Salmon, J. Strunk, M. Wachs, M. Abd-El-Malek, J. Lopez, and G. Ganger, "Stardust: Tracking activity in a distributed storage system," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 34, pp. 3–14, 2006.
[9] A. Chanda, A. Cox, and W. Zwaenepoel, "Whodunit: Transactional profiling for multi-tier applications," *ACMSIGOPS Operating Syst. Rev.*, vol. 41, pp. 17–30, 2007.
[10] J. Hollingsworth, "An online computation of critical path profiling," in *Proc.SIGMETRICS Symp. Parallel Distrib. Tools*, 1996, pp. 11–20.
[11] E. Koskinen and J. Jannotti, "Borderpatrol: Isolating events for Black-box tracing," in *Proc. 3rd Eur. Conf. Comput. Syst.*, 2008, pp. 191–203.
[12] B. Tak, C. Tang, C. Zhang, S. Govindan, B. Urgaonkar, and R. Chang, "vPath: Precise discovery of request processing paths from Black-box observations of thread and network activities," in *Proc. Conf. USENIX Annu. Tech. Conf.*, 2009, p. 19.
[13] P. Fournier and M. Dagenais, "Analyzing blocking to debug performance problems on multi-core systems," *ACM SIGOPS Operat. Syst. Rev.*, vol. 44, no. 2, pp. 77–87, 2010.
[14] P. Barford and M. Crovella, "Critical path analysis of TCP transactions," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 30, no. 4, pp. 127–138, 2000.
[15] M. Desnoyers, "Low-impact operating system tracing," Ph.D. dissertation, École Polytechnique de Montréal, Montréal, QC, Canada, 2009.
[16] D. L. Mills, "Precision synchronization of computer network clocks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 24, no. 2, pp. 28–43, 1994.
[17] L. Lamport. (1978). Time, clocks, and the ordering of events in a distributed system," *Communications ACM* [Online]. *21(7)*, pp. 558–565. Available: http://dl.acm.org/citation.cfm?id=359563
[18] R. Rabenseifner, "The controlled logical clock-a global time for trace based software monitoring of parallel applications in workstation clusters," in *Proc. 5th EUROMICRO Workshop Parallel Distrib. Process.*, 1997, pp. 477–484.
[19] A. Duda, G. Harrus, Y. Haddad, and G. Bernard, "Estimating global time in distributed systems." in *Proc. 7th Int. Conf. Distrib. Comput. Syst.* 1987, pp. 299–306.
[20] B. Poirier, R. Roy, and M. Dagenais, "Accurate offline synchronization of distributed traces using kernel-level events," *ACM SIGOPS Operat. Syst. Rev.*, vol. 44, no. 3, pp. 75–87, 2010.
[21] F. G. et al. (2014, Nov.). *Workload experiments and active path analysis source code* [Online]. Available: https://github.com/giraldeau/
[22] A. S. Tanenbaum and M. Van Steen, *Distributed Systems*, 2nd ed. New York, NY, USA: Pearson, 2007.
[23] O. A. Inc. (2014, Nov.). Trail: RMI (the Java^{TM} Tutorial) [Online]. Available: https://docs.oracle.com/javase/tutorial/rmi/
[24] D. S. Foundation. (2014, Nov.). Django project tutorial [Online]. Available: https://www.djangoproject.com/
[25] O. Zaki, E. Lusk, W. Gropp, and D. Swider, "Toward scalable performance visualization with Jumpshot," *Int. J. High Perform. Comput. Appl.*, vol. 13, no. 3, pp. 277–288, 1999.
[26] K. Sedgewick, Robert and Wayne, *Algorithms*, 4th ed. Bostan, MA, USA: Addison-Wesley, Mar. 2011.

Francis Giraldeau received the BS degree in electrical engineering and the MS degree in computer science from the University of Sherbrooke, in 2005 and 2010, respectively. He is currently working toward the PhD degree in computer engineering from the Polytechnique Montreal. His current research focuses is on the automatic analysis of operating system traces. He is a member of the IEEE.

Michel Dagenais is a professor at the Ecole Polytechnique de Montreal and co-founder of the Linux-Québec user group. He authored or co-authored over one hundred scientific publications, as well as numerous free documents and free software packages in the fields of operating systems, distributed systems, and multicore systems, particularly in the area of tracing and monitoring Linux systems for performance analysis. In 1995-1996, during a leave of absence, he was the director of software development at the Positron Industries and chief architect for the Power911, object oriented, distributed, fault-tolerant, call management system with integrated telephony, and databases. In 2001-2002, he spent a sabbatical leave at the Ericsson Research Canada, working on the Linux Trace Toolkit, an open source tracing tool for Carrier Grade Linux. The Linux Trace Toolkit next generation is now used throughout the world and is part of several specialised and general purpose Linux distributions. He is a member of the IEEE.