Titre: Title:	Runtime latency detection and analysis
Auteurs: Authors:	Julien Desfossez, Mathieu Desnoyers et Michel R. Dagenais
Date:	2016
Type:	Article de revue / Journal article
Référence: Citation:	Desfossez, J., Desnoyers, M. & Dagenais, M. R. (2016). Runtime latency detection and analysis. *Software: Practice and Experience*, *46*(10), p. 1397-1409. doi:10.1002/spe.2389

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/2989/
Version:	Version finale avant publication / Accepted version Révisé par les pairs / Refereed
Conditions d'utilisation: Terms of Use:	Tous droits réservés / All rights reserved

Titre de la revue: Journal Title:	Software: Practice and Experience
Maison d'édition: Publisher:	Wiley
URL officiel: Official URL:	https://doi.org/10.1002/spe.2389
Mention légale: Legal notice:	This is the peer reviewed version of the following article: Desfossez, J., Desnoyers, M. & Dagenais, M. R. (2016). Runtime latency detection and analysis. *Software: Practice and Experience*, *46*(10), p. 1397-1409. doi:10.1002/spe.2389, which has been published in final form at https://doi.org/10.1002/spe.2389 . This article may be used for non-commercial purposes in accordance with Wiley Terms and Conditions for Self-Archiving.

Runtime latency detection and analysis

J. Desfossez[a,*], M. Desnoyers[b], M.R. Dagenais[a]

[a] *École Polytechnique de Montréal, 2900 Boulevard Édouard-Montpetit, Montréal, Québec H3T 1J4*
[b] *Efficios, Inc., 4200 Boulevard St-Laurent, suite 680, Montréal, Québec, H2W 2R2*

Abstract

Detecting latency-related problems in production environment is usually done at the application level with custom instrumentation. This is enough to detect high latencies in instrumented applications but does not provide all the information required to understand the source of the latency and is dependent on the manually deployed instrumentation. The abnormal latencies usually start in the kernel due to contention on physical resources or locks. Hence finding the root cause of a latency may require a kernel trace. This trace can easily represent hundreds of thousands events per second. In this paper, we propose a methodology to identify and analyze latency problems that occur at the kernel level. We introduce a new kernel-based approach that enables kernel developers to track latency problems with a cost compatible with production requirements and trigger actions based on these events to allow them to understand and fix this class of problems.

Keywords: Latency, Tracing, Cloud Computing, Real-Time

1. Introduction

Monitoring resources usage is a mandatory practice in production environments. The tools commonly deployed extract metrics from the servers to graph the usage of the resources over time and eventually generate alerts. The resources monitored include processor, memory, network and disk usage. The load average is also a metric commonly used to detect if a server is saturated. When the servers are in a controlled state with predictable resources usage, this monitoring is enough to detect the major problems.

The latency-specific monitoring is usually performed in production-critical application. For example, a web server can compute the time difference between a request is received and served to provide metrics and alerts based on this value. This process implies that the application developers add this instrumentation in their code and that the monitoring tools interface with this information source. Adding instrumentration in an application adds some cost so it is usually done at high level (FIXME: ref web server metrics) or is only enabled one fraction of the time (FIXME: ref zipkin) to provide an overview and eventually a breakdown of where the time is mostly spent. This allows to sample the latencies without impacting too much the production and it can orient the research of the root cause for high usage services.

But there are cases where a latency is caused by an uninstrumented application, by a resource contention, by a scheduling problem, by an external dependency (for example: interrupt handlers and virtualization). Covering all of these cases with custom instrumentation increases significantly the amount of instrumentation and software maintenance to perform at each deployment and update.

To circumvent this problem, some systems deployment include a phase of micro-benchmarking (FIXME: ref RT, Netflix) where the operators verify that the machine corresponds to the requirements before starting the production on it. But these benchmarks cannot be run again during the production, so if the conditions change during the production, they can only rely on the existing monitoring information to detect the change and react. In a virtualized environment such as cloud computing (FIXME: ref), the contention on the physical resources can change because of other virtual machines spawned on the same physical machine.

Finally, in a production environment, the most interesting latencies are the highest ones, the outliers, the ones that are very rare but cause serious problems on the end-user applications and can reveal serious unhandled problems. The probability of detecting these problems with a sampling based technique is low and they are completely invisible in average-based monitoring. FIXME: definition outliers, 6-sigma, etc.

When trying to understand this class of sporadic low-level high latency problems that are difficult to reproduce outside of production condition, system administrators face a problem in their problem-solving methodology. They can usually identify what is the problematic subsystem by running some of the usual diagnostics tools on the server (FIXME: ref Brendan Gregg book), but after that, they are either bound to do a trial and error test on solution ideas (update the kernel, other softwares, etc), or try to capture more information on the faulty subsystem and hope for the problem to reappear even though they changed the experiment conditions (FIXME: heisenbug).

At this stage, tracing is the class of tools used. It can be at the process-level (*strace*, *perf* [7]), at the network layer (*tcpdump* [9]), and the kernel-level (*ftrace* [6], *perf*, *LTTng* [15], *SystemTap* [17]), or completely *ad-hoc* added just for the occasion in the target application (with *printf*(), *logger*, etc). All of these tools have their own share of advantages and drawbacks, but one common factor for all these tools is the additional work on the target system caused by the instrumentation and the trace extraction. This is enough to change the resources usage, the scheduling, the locking, and change completely the experiment conditions. When all of this is in place, the problem has to be detected again while tracing. Depending on how long it takes for the problem to reappear, the traces can be huge and require an expert to understand them (FIXME: needle in the haystack).

In this paper, we are trying to find the best way to detect latency outliers and extract background informations to understand the root cause. This study serves as the basis for developping *latency_tracker*. This module aims at measuring and executing actions on high latencies while providing these guaranties:

- Provide a flexible framework for instrumenting various parts of the kernel.
- Work in every kernel execution contexts (including *NMI* and *MCE* handlers).
- Scale on concurrent workloads with a small and predictable overhead.
- Work in production environment as background monitoring.

The result is a scalable approach to generic runtime latency detection.

We start by surveying the existing related work in Section 2, we then present the proposed architecture in Section 3, we present some analyses implemented with this new mechanism in 4 and we measure the overhead in Section 5. Finally, we discuss the results and ideas for the future in Section 6.

2. Related Work

2.1. Latency analysis

With the amount of large online services growing up, a lot of research is already in progress to eliminate as much latency as possible throughout the stack. The kernel itself is known to be a major factor in the latency of network applications. The intent here is to serve the more concurrent clients as possible with the smallest number of server. Applications like Chronos [10] remove the kernel and the network stack from the critical path of network applications and reduce the latency of certain applications by a factor of twenty. A complete breakdown of the latency of all the components involved in a network communication is presented in [11] and it clearly shows that most of the time is spent in the network stack.

Identifying latencies in specific subsystems is useful to fix one particular bottleneck, but when operating real services, the load usually relies on the performance of multiple subsystems and it can be difficult to get an overview of where the time is spent. In [12], Leverich et al. classify the latencies of real-world co-located applications in three categories: queueing delay, scheduling delay and thread load imbalance. In addition, they explain most of the delays and it gives insights on the parameters to take into account when trying to optimize the usage of servers while still keeping a fast response time. In the case studied, they manage to co-locate computing services (such as analytics) on *memcached* cache servers in a architecture inspired by the Facebook use-case.

Under usage of computing resources is a common problem, in this 29-days trace of a Google cluster [18], we see that the main factor is a lack of accurate estimate of resources request, but in more interactive applications, we see that engineers and cloud load balancer tend to leave a lot of headroom because of the unknown high-impact latencies that might occur. For example, in a study on server utiilization in public clouds ([13]), we see that the estimated server utilization on Amazon EC2 is below 20% because we don't have a reliable way to guarantee a certain quality of service in terms of responsiveness.

Maximising the usage of servers while maintaining a low latency is also of interest for the energy efficiency research. For example, in [14], researchers at Google, identified patterns of usage, estimated real world Service Level Objectives and built dynamic controllers to dispatch the work while respecting the expectations of the users in terms of latency.

We can see that a lot of research is currently done in optimizing the high-impact delays in the production chain, but we did not find any paper or project working to actually detect and explain the latency outliers in real world production environments.

2.2. Latency measurements

In order to measure the time spent in serving a request in production, we need an efficient request tracking mechanism. SystemTap [17] is an efficient tool for dynamically hooking analyses in the critical path of a subsystem in the kernel. The language allows to quickly create a custom analysis probe. It is often used for statistics because it is designed to avoid perturbating the system too much (a lot of work is done in the compiled code to account the time spent in each probe and return if it is higher than a specified threshold). We experimented with it for our use-case and identified that its generic design makes it difficult to scale an analysis for concurrent applications. Since we are tracking entry and exit events that can occur randomly on any processor, the safe approach used in SystemTap to lock the associate arrays imposes a high impact on our workload. Moreover, the scripting environment is self-contained which makes it difficult to emit tracepoints and be called directly from an external module or the kernel itself. The results from our experiments in scaling is available in Section 5.

To measure the entries and exits of various subsystems in the kernel, we could use the tracepoint infrastructure with one the kernel tracers (perf, ftrace, LTTng) and compute the metrics offline, but the amount of data to extract can be in the order of megabytes/sec which is too much for only identifying outliers.

The other methods to measure the performance of a system rely on polling the *proc* filesystem, but it usually consist of usage metrics that are used to show the average usage of the system. Most of the common production monitoring tools such as Ganglia [16], SMILE [21], or Parmon [2] rely on local agents or the SNMP protocol to read these counters.

Ftrace has latency analyses targetted specifically for real-time systems [19]. They are hardcoded in the kernel and provide an interesting view of what caused the interrupts or the preemption to be disabled for some time, with a trace leading to this event. This kind of analysis is exactly what we are trying to achieve but for a more general purpose that hard realtime systems.

The other realtime analyses we know of are microbenchmarks ([3], [1]) aimed at certifying that a hardware architecture satisfies the response-time needs for a certain application. Tools like *hackbench* and *cyclictest*, part of the $rt - test$ tools suite, are often used for this purpose, but cannot be run with production data, since they are benchmarking tools.

3. Architecture

In this paper, we propose a generic approach to track latency events in production environments. Our main goal is to provide an efficient way for developers and advanced users to measure the delay between two or more related events and act on this measurement from the critical path. So in this section, we are trying to find the best way to keep track of thousands of concurrent events and provide enough flexibility and background information to allow the users to understand the source of the latency.

This work aims to find the appropriate balance between generating too much data for offline post-processing and intruding too much on the critical path.

This high-level architecture of our work is presented in Figure 1.

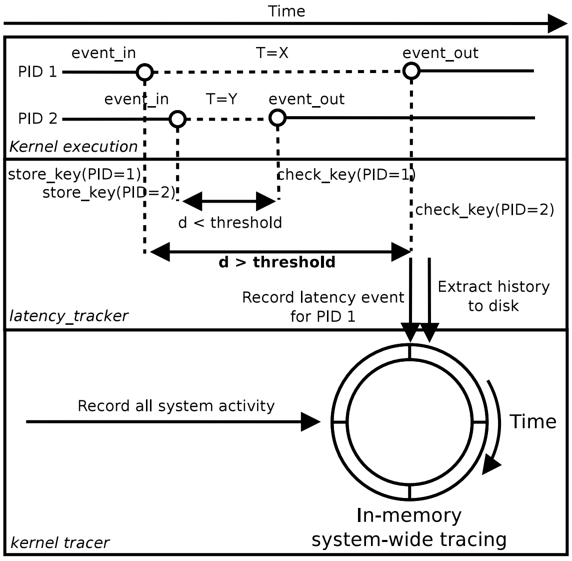

Figure 1: Latency tracker architecture

As shown in Figure 1, our approach is split in three parts: the collection of data in the critical path (*Kernel execution*), the detection of high latency (*latency_tracker*) and the extraction of the kernel trace (*kernel tracer*). We first detail the collection and extraction of data and then focus on the algorithms and the design decisions that consitute the core of the tracking mechanism.

3.1. In-memory data collection

In order to collection the background history efficiently, we use the LTTng tracer configured in "flight-recorder" mode, but the overall solution is not dependant on any particular tracer. LTTng since version 2.3 has the ability to record the trace only in memory in a ring-buffer. This process removes completely the cost of the I/O operations required to extract the trace on disk or on the network. When an external factor detects that an interesting condition happened (the user manually or a coredump handler for now), we can trigger a "snapshot" which takes all the content of the ring-buffer and extracts it in trace files (local or remote). Then the user can run analyzes (manual or automatic) on a trace that contains only a relatively small window of data around the interesting event. The analyzes can be simple statistics such as (FIXME: ref python analysis scripts) to more advanced critical path analysis (FIXME: ref fgiraldeau).

The total buffer size per-channel is calculated in Equation 1.

$$buf\_size = num\_subbuf * subbuf\_size \qquad (1)$$

The relationship between the number of events in a snapshot and the configuration of the tracer can be approximated by the Equation 2.

$$avg\_nb\_events = buf\_size / avg\_event\_size \qquad (2)$$

Therefore, the period covered by a snapshot is approximated by the Equation 3.

$$snapshot\_period = avg\_nb\_events / event\_rate \qquad (3)$$

For example, if we enable only the *sched_switch* on an idle 8-core server, with the default configuration of 4 sub-buffers of 256kB per-cpu, the snapshots contain around 15000 events which cover a period of 30 seconds (around 60 bytes/event). On the other hand, with the exact same configuration but with a high scheduling load on all cores, the snapshots have the same amount of events but only cover a period of 5 seconds.

For the latency tracking analysis, we use this feature to keep a relatively short history of trace data around the interesting events. The parameters such as the sub-buffer size and enabled events are configured depending on the conditions and the problems we are trying to solve.

3.2. Callbacks: data extraction

The callbacks are the action called when the delay between two events is higher than a predefined threshold, and when the timeout is reached. The callback is a function provided by the user of the module. For the case of high delay, the function is called during the exit event, so it is in the critical path of the tracked system. For that reason, the action executed here need to be as fast as possible to avoid starting a feedback loop of latencies. The advantage of calling it from this site, is that the user gains full control over the execution and can extract accurate information about the current state of the process or the system.

For example, in the use-case we defined earlier, we want the callback to record an event in an active tracing session to mark the exact time where the high latency was detected and then extract the trace buffers that contains this event and as much history as possible. Emitting an event in the context of the target application is fast enough (328-338 ns/event [5]) to be run directly in the callback since it is usually multiple orders of magnitude less than the latency threshold, but collecting the snapshot is a more intrusive operation that requires the communication between two user-space daemons over unix sockets, so we need to delegate another task to perform this operation in the background and give back the control to the kernel as soon as possible.

To handle this problem, we created a special *procfs* file. This file is owned by the tracking module. When a user-space process reads this file, it is put in a wake-up queue. When we detect a high latency, we wake up all the processes in the queue and make the read operation return. When the process returns from the *read* or *poll*, it knows an important event happened and can handle it all in user-space without blocking the kernel. Since the wake up can happen in the context of the scheduler (for example in the wakeup-latency analysis), we have to create an *irq_work* and work from the IRQ context to avoid calling *sched_wakeup* during a *sched_switch* which results in a deadlock.

Since the system is most likely loaded when the callback is activated, there is a risk that the user-space program will not be scheduled soon enough to extract enough background information in the snapshot as possible before it is overwritten by the tracer, so it is important to set a real-time priority for this task if its result is critical.

Finally, to avoid feedback loops, we implemented a simple rate limiter set to one snapshot per second.

3.3. Latency tracker

Once we have a way to collect efficiently a short history around an interesting event, we need to automatically detect the interesting conditions and trigger the recording of the snapshot. This part is the most important original contribution of our work.

3.3.1. Design principles

The objective is to trigger an action whenever a high latency is detected inside the kernel. The action is provided by the user and can be, for example, to extract the tracing buffers, generate alerts, compute advanced statistics, record a stack trace, etc. The cost of this additional tracking must be bounded and low enough to run in production.

In order to do that, we developed a kernel module ([4]) that exposes new functions to the kernel. These functions are designed to be called from anywhere in the kernel: hardcoded in the source code, or from a tracepoint handler, a *kprobe* callback, a *netfilter* hook, etc. Moreover, these functions can be called from any context including system calls, timers and interrupt handlers.

The module uses a common *key* to track the time difference between two punctual events (the entry and the exit) and take actions depending on the delay between the two events. The key depends on the context and can be in any form (string, structure, integer, etc) as long as it is the same in the two events.

In Algorithm 1, we present the basic usage of the tracker. In this example, we use the static instrumentation already existing in the Linux kernel and connect two probes. These probes only extract the parameters *sector* and *dev* and use these as a key to track the request. The latency tracker module then handles the storing and matching of the key. If the delay between the two events, is higher than *threshold*, then the callback function *cb* is executed.

Algorithm 1 Block latency tracker example

```
function INIT
    $tracker \leftarrow latency\_tracker\_create()$;
    $tracepoint\_probe\_reg(''block\_rq\_issue'')$;
    $tracepoint\_probe\_reg(''block\_rq\_complete'')$;
function PROBE_RQ_ISSUE(e)
    $key.d \leftarrow e.dev$;
    $key.s \leftarrow e.sector$;
    $t \leftarrow theshold$;
    $latency\_tracker\_event\_in(tracker, key, t, cb)$;
function PROBE_RQ_COMPLETE(e)
    $key.d \leftarrow e.dev$;
    $key.s \leftarrow e.sector$;
    $latency\_tracker\_event\_out(tracker, key)$;
```

In addition, the module provides a *timeout* parameter in order to call the callback function if the exit event has not been called before a user-defined expiry time. This function allows to perform off-CPU profiling (FIXME: ref Brendan) and some focused sampling which is particularly useful to detect high latencies while they are happening (before the exit event arrives). It is for example a good place to take a stack trace of the blocked process and sample external counters. The callback function receives a parameter to let it know if it was called for a timeout or a normal high latency, so the user has the control regarding the operation to take depending on the case. Also, the key is not deleted until the exit event happens, so the user callback gets the control two times when a timeout has been reached.

Moreover, at any point in time during the lifetime of a pending event, it is possible to query the latency_tracker to see if a key is currently active. This opens up the possibility of creating "stateful tracing events". For example a probe can be hooked on any kernel function with a *kprobe*, when the callback is called, the probe can check if the current process has an pending latency event and extract additional information specifically related to this condition. Before this active state tracking, we had to extract the data every time the callback was hit and process the result after the fact which resulted in additional noise, overhead and processing time.

3.3.2. Memory allocation

To allow the calls to work in all of these different states, we need to make sure our module does not trigger any page fault which could lead to deadlocks in certain situations. In order to do that, we allocate the memory required to store the keys in the hash table when the module is loaded and ensure that it is ready to be used. The user evaluates the maximum number of keys that should be used concurrently and all the memory is allocated by the module before starting the work. The free memory is organized in a simply linked free list.

In some production cases, the user does not know how much concurrent keys can co-exist and we don't want to miss interesting events because of misconfiguration. To solve this problem, if the impact is tolerable, the free list can be dynamically resized outside of the critical path. In order to do that, we set a special flag when allocating the element stored in the middle of the list. When we start using this element, we set a flag in the tracker. Periodically, a timer handler (also used for garbage collection) checks if this flag is set and if it is, starts a *workqueue* to resize the list (up to a maximum size defined by the user). The algorithm for this process is detailed in Algorithm 2.

We have to use a timer to start the *workqueue* process, even though the *workqueue* is itself an independant task, because when queueing a new work, the Linux kernel informs the scheduler that the task needs to run using a *sched_wakeup* which is a function we are interested in when tracking scheduling wakeup latencies. Even though the code is reentrant, we want to limit the impact we have on the server in the critical path of waking up a task.

Since our workload can be executed in parallel, we need to protect the access to the free list. We experi-

Algorithm 2 Freelist allocation and resize

```
function FREELIST_INIT(tracker, size)
    $list \leftarrow tracker.list$
    for i = 0; i < size; i++ do
        $e \leftarrow alloc(latency\_tracker\_event)$
        if $i == size/2$ then
            $set\_resize\_flag(e)$
        $list\_add(list, e)$
    $tracker.size \leftarrow size$
function FREELIST_GET_NEW(tracker)
    if $list\_empty(tracker.list)$ then
        return NULL
    $e \leftarrow list\_first(tracker.list)$
    if $has\_resize\_flag(e)$ then
        $tracker.need\_resize \leftarrow True$
    return e
function TRACKER_TIMER_HANDLER(tracker)
    if $tracker.need\_resize$ then
        $queue\_work(tracker.resize\_work)$
function TRACKER_RESIZE_WORK(tracker)
    $size \leftarrow tracker.size$
    $size \leftarrow min(size * 2, tracker.max\_size)$
    $freelist\_init(tracker, size)$
```

mented with two linked-list implementations built-in the Linux kernel and three locking strategies (along with the various hash tables algorithms detailed in Section 3.3.3).

We first experimented with a basic linked-list protected by an IRQ-safe *spinlock*, this experiment helped us create the first prototype of the module and identify the problematic concurrency situations. We then took a look at the strategy SystemTap uses to protect their associative arrays and ported this mechanism to protect the free list and the hash table. SystemTap uses a *rwlock* and loops on a $write_trylock$ every 10 micro-seconds until it obtains the lock. We noticed a similar performance than the *spinlock* approach, but the *rwlock* creates less pressure on the CPU than the *spinlock* which is interesting in case of *hyperthreading* cores which share some resources, the down-side is the risk of starvation.

Since the free list is a significant contention point, we experimented with the lock-free linked-list of the Linux kernel (protected by RCU) and improved significantly the overall performance (all results in Section 5).

3.3.3. Hash table

We are looking to compare arbitrary data at entry and exit sites to track the state of any operation that can be expressed as a request. Hence, the core of the tracking mechanism is a generic hash table inside the kernel. To give the maximum performance to users, the *hashing* and *matching* functions can be customized to perform best depending on the type of key used. Because of the possibility of preemption, thread migration, interrupt handling, etc, the table is shared among the processors. Our intent is to find the best locking strategy to limit the overhead we add in the critical path. One important aspect of this workload is that the usage of the hash table is typically symmetric for each key: one insertion, one lookup immediately followed by a removal. But this process can happen in parallel with thousands of active keys, so we can't just use a list of active keys, the lookups need to happen in $\mathcal{O}(\log n)$. In addition, the user must be able to handle the case of duplicated entries.

The Linux kernel already provides multiple choices of locking strategies and hash table implementations, Since our use-case is the worst-case of a hash table and can be called from any context, we studied closely the algorithms of every options available both in term of speed and scalability.

The default hash table of the Linux kernel requires a lock (IRQ-safe in our case) for each operation. We identified that a stategy based on *write_trylock* (from *rwlock*) is more efficient for this use-case than a *spinlock* on hyperthreaded CPU cores. However, the impact is still high for parallel workloads (details in Section 5).

Since 3.17, the relativistic hashtable (*rhashtable* [20]) is included in the mainline Linux kernel. This hashtable has the advantage of being resizable, which is an important feature for the future of our work, but also lock-free for the read-side. Thanks to *RCU*, we only need to lock the table when inserting new elements which is half of our workload. Moreover, from the benchmarks provided by the authors, the performance seems better than the default hash table (FIXME: ref). But after experimenting with it, we identified a high overhead starting at 16 concurrent processors (FIXME: explanation with perf results). We didn't use the resize capability of this table because the resize operation is triggered in the context of usage which can be problematic for our operating conditions.

The last hash table we used comes from the userspace-RCU library (FIXME: ref), that we ported to the kernel. This hashtable is completely lock-free and helped us achieve our best scalability results. This hash table can also be resized and the resize operation can be performed outside in a separation execution context, so it is another advantage for this structure.

3.3.4. Locking

Here is a summary of the various linked-list, hash tables and locking algorihms that we compared for this study to identify the best combination for our use case. The *Id* is used as a legend for the performance graph.

Id	Free list	Hash table	Locking
A	basic list	basic HT	spinlock
B	basic list	basic HT	try lock
C	lockless list	basic HT	spinlock
D	lockless list	basic HT	try lock
E	basic list	rHT	spinlock
F	basic list	rHT	try lock
G	lockless list	rHT	spinlock
H	lockless list	rHT	try lock
I	basic list	urcuht	spinlock
J	basic list	urcuht	try lock
K	lockless list	urcuht	N/A

Table 1: Locking and data structure tested

4. Use-cases implemented

4.1. Off-CPU profiling

Based on this architecture, we created a kernel module to track the time a process spends not running and record its kernel stack when it returns on a CPU. That way, we know at runtime all the processes that are blocked and we can then confirm if it is a normal delay or not. It can help identify locks imbalance, resources shortage and other concurrency issues.

Here is the result with a threshold set at 5 seconds, we see a thread of the *rsyslogd* daemon that was waiting on a *read* system call.

```
offcpu: in:imklog (743) 5395165 us
 [<ffffffff817256d9>] schedule+0x29/0x70
 [<ffffffff810be37a>] do_syslog+0x4fa/0x5c0
 [<ffffffff81232964>] kmsg_read+0x44/0x60
 [<ffffffff812243dd>] proc_reg_read+0x3d/0x80
 [<ffffffff811bdd05>] vfs_read+0x95/0x160
 [<ffffffff811be819>] SyS_read+0x49/0xa0
 [<ffffffff81731d7d>] system_call_fastpath+0x1a/0x1f
```

Using some simple visualisation scripts [8], we can also generate a flamegraph like presented in Figure 2 for the same data.

4.2. Scheduler wake up latency

For real-time systems, it is important to have a boundary on the delay between the time the scheduler decides a process should be running and the time it actually runs. So we used our new architecture to

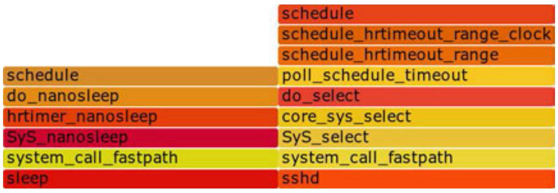

Figure 2: Latency flame graph

compute the time difference between a *sched_wakeup* and a *sched_switch* on the same PID. We used the callback to record a LTTng snapshot when the delay was above a threshold of 5 milliseconds. We let run this analysis for 24 hours on an active web and mail server and identified interesting latency patterns that would have been really hard to understand without this new method. Thanks to the small trace generated when a latency occurs, these new results led us to create new analyses tools that focus on explaining these latencies. The details of these results will be part of another study.

4.3. Syscall calls latency

Another useful module implemented with this new tracking mechanism is a system call latency tracker. We created a module that hooks on system call entries and exits and compute the time difference between the two events. If above a threshold, we generate an event that can record a LTTng snapshot. In addition, we used the feature to check the state of a current process. So at each *sched_switch*, we check if the current process has been blocked in a system call for more than the threshold. If true, we extract its kernel stack. This allows us to see exactly where a system call is blocked inside the kernel every time this process gets some CPU time. FIXME: graph and/or SVG.

5. Measurements

In order to evaluate the performance of our approach and decide if it is suitable for a usage in production, we first measure the overhead introduced by the tracing in memory and by the tracking of latencies by our module under a variety of loads in micro benchmarks targetting specific resources. We then evaluate the whole methodology by deploying our solution in a real world use case, we intend to detect the high latencies and find the optimal parameters to have enough background information to understand the source of the problem and control the memory footprint.

5.1. Flight-recorder mode

- flight-recorder mode alone - flight-recorder parameters tweak (size/events vs snapshot duration)

5.2. Latency tracker overhead

5.2.1. CPU overhead

In this experiment, we evaluated the impact of the hash table and the linked list implementations and locking on a highly-concurrent workload while increasing the number of CPUs available. The test machine is a quad-socket AMD Opteron(TM) Processor 6272, so the kernel sees 64 CPUs, but the AMD Bulldozer architecture is a partial Simultaneous Multithreading (SMT) architecture, so the FPU and L2 cache are shared between pairs of core, but the integer cores are independant. Before running the 64-cores tests, we evaluated that the overhead of running our workload on shared cores is between 10-15% than running on independant CPU cores. Up-to 32 simultaneous CPUs we ran on independant cores to limit this side-effect, but for the test at 48 and 64 cores, we had to use shared cores.

The test consists of running *hackbench* 100 times and compute the statistics about how long it takes to complete. *Hackbench* spawns 10 groups of processes and tries to exchange 100 bytes back and forth between the senders and receivers over 40 different sockets. So it consists of 400 tasks trying to run simultaneously which represents a high concurrency on the scheduler. When increasing the number of CPUs available between 1 and 64, we see a linear speedup. The graph **??** shows the results with all the combinations detailed in Table 1.

5.2.2. I/O overhead

We tried to quantify the overhead imposed when tracking the block-related events like explained in Algorithm 1, but we were unable to find a significant overhead. We used the *fileio* test of *sysbench* in various combinations (sequential read, sequential write, random read and random write) on SSD and rotating drive.

6. Conclusion and Future Work

Conclusion

7. Acknowledgments

This work was made possible by the financial support of Ericsson, EfficiOS and NSERC. We are grateful to Naser Ezzati for the reviews.

8. References

[1] Wolfgang Betz, Marco Cereia, and Ivan Cibrario Bertolotti. Experimental evaluation of the linux rt patch for real-time applications. In *Emerging Technologies & Factory Automation, 2009. ETFA 2009. IEEE Conference on*, pages 1–4. IEEE, 2009.

[2] Rajkumar Buyya. Parmon: a portable and scalable monitoring system for clusters. *Software-Practice and Experience*, 30(7):723–740, 2000.

[3] John M Calandrino, Hennadiy Leontyev, Aaron Block, UC Devi, and James H Anderson. Litmus^ rt: A testbed for empirically comparing real-time multiprocessor schedulers. In *Real-Time Systems Symposium, 2006. RTSS'06. 27th IEEE International*, pages 111–126. IEEE, 2006.

[4] Julien Desfossez. latency-tracker source code. https://github.com/jdesfossez/latency_tracker, 2014.

[5] Mathieu Desnoyers. *Low-impact operating system tracing*. PhD thesis, École Polytechnique de Montréal, 2009.

[6] Jake Edge. A look at ftrace. http://lwn.net/Articles/322666/, 2009.

[7] Jake Edge. Perfcounters added to the mainline. http://lwn.net/Articles/339361/, 2009.

[8] Brendan Gregg. stack trace visualizer. https://github.com/brendangregg/FlameGraph, 2014.

[9] Van Jacobson, Craig Leres, and S McCanne. The tcpdump manual page. *Lawrence Berkeley Laboratory, Berkeley, CA*, 1989.

[10] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M Voelker, and Amin Vahdat. Chronos: predictable low latency for data center applications. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 9. ACM, 2012.

[11] Steen Larsen, Parthasarathy Sarangam, Ram Huggahalli, and Siddharth Kulkarni. Architectural breakdown of end-to-end latency in a tcp/ip network. *International journal of parallel programming*, 37(6):556–571, 2009.

[12] Jacob Leverich and Christos Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of the Ninth European Conference on Computer Systems*, page 4. ACM, 2014.

[13] Huan Liu. A measurement study of server utilization in public clouds. In *Dependable, Autonomic and Secure Computing (DASC), 2011 IEEE Ninth International Conference on*, pages 435–442, Dec 2011.

[14] David Lo, Liqun Cheng, Rama Govindaraju, Luiz André Barroso, and Christos Kozyrakis. Towards energy proportionality for large-scale latency-critical workloads. In *Proceeding of the 41st annual international symposium on Computer architecuture*, pages 301–312. IEEE Press, 2014.

[15] M. R. Dagenais M. Desnoyers. The lttng tracer : a low impact performance and behavior monitor for gnu/linux. In *Proceedings of Ottawa Linux Symposium 2006*, pages 209–223, 2006.

[16] Matthew L Massie, Brent N Chun, and David E Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817–840, 2004.

[17] Vara Prasad, William Cohen, FC Eigler, Martin Hunt, Jim Keniston, and J Chen. Locating system problems using dynamic instrumentation. In *2005 Ottawa Linux Symposium*, pages 49–64. Citeseer, 2005.

[18] Charles Reiss, Alexey Tumanov, Gregory R Ganger, Randy H Katz, and Michael A Kozuch. Towards understanding heterogeneous clouds at scale: Google trace analysis. *Intel Science and Technology Center for Cloud Computing, Tech. Rep*, page 84, 2012.

[19] Steven Rostedt. Finding origins of latencies using ftrace. *Proc. RT Linux WS*, 2009.

[20] Josh Triplett, Paul E McKenney, and Jonathan Walpole. Resizable, scalable, concurrent hash tables via relativistic programming. In *USENIX Annual Technical Conference*, page 11, 2011.

[21] Putchong Uthayopas, Surachai Phaisithbenchapol, and Krisana Chongbarirux. Building a resources monitoring system for smile beowulf cluster. In *Proceedings of HPC Asia98 Conference, Singapore*, 1998.