Recovering System Metrics from Kernel Trace

Francis Giraldeau
École Polytechnique de Montréal
francis.giraldeau@polymtl.ca

Julien Desfossez
École Polytechnique de Montréal
julien.desfossez@polymtl.ca

David Goulet
École Polytechnique de Montréal
david.goulet@polymtl.ca

Michel Dagenais
École Polytechnique de Montréal
michel.dagenais@polymtl.ca

Mathieu Desnoyers
EfficiOS Inc.
mathieu.desnoyers@efficios.com

Abstract

Important Linux kernel subsystems are statically instrumented with tracepoints, which enables the gathering of detailed information about a running system, such as process scheduling, system calls and memory management. Each time a tracepoint is encountered, an event is generated and can be recorded to disk for offline analysis. Kernel tracing provides system-wide instrumentation that has low performance impact, suitable for tracing online systems in order to debug hard-to-reproduce errors or analyze the performance.

Despite these benefits, a kernel trace may be difficult to analyze due to the large number of events. Moreover, trace events expose low-level behavior of the kernel that requires deep understanding of kernel internals to analyze. In many cases, the meaning of an event may depend on previous events. To get valuable information from a kernel trace, fast and reliable analysis tools are required.

In this paper, we present required trace analysis to provide familiar and meaningful metrics to system administrators and software developers, including CPU, disk, file and network usage. We present an open source prototype implementation that performs these analysis with the LTTng tracer. It leverages kernel traces for performance optimization and debugging.

1 Introduction

Tracing addresses the problem of runtime software observation. A trace is an execution log of a software, that consists essentially of an ordered list of events. An event is generated when a certain path of the code is executed, commonly called tracepoint. Each event consists of a timestamp, a type and some arbitrary payload.

Tracepoints can be embedded statically in software or dynamically inserted. Dynamic tracing allows custom tracepoints to be defined without source code modification. While this approach is flexible, static tracepoints are generally faster. In addition, tracing can be performed at the kernel and user-space level. In this paper, we focus on the static instrumentation of the Linux kernel provided by the Linux Trace Toolkit next generation (LTTng) [1]. Unlike a debugger, tracing a program does not interrupt it. As such, performance of the tracer is critical to minimize disturbance of the running software. LTTng offers this level of performance, allowing to trace the kernel very efficiently.

Runtime information on a system can also be obtained by recording metrics periodically from files under the `/proc` directory. Utilities like `top` and `ps` use this interface, parse their content and format them for display. This technique provides statistics about the system at a sampling frequency based determined by the interface. In contrast, kernel tracing records all events according to time. Instead of pooling metric values, they can be recovered at arbitrary resolution afterwards from the trace.

This paper is organized as follows. The kernel tracing infrastructure is presented in section 2. This presentation applies to the latest stable release of LTTng 0.249, which is used throughout the paper. In section 3, we present available tracepoints in the Linux kernel, their meaning and how we can recover metrics by processing them. A prototype that performs such analysis is presented in section 4. Finally, future work for LTTng 2.0 is discussed in section 5.

2 LTTng kernel tracer

The tracer is based on static tracepoints in the kernel source code. Each time a tracepoint is encountered and is enabled, an event is added to an in-memory ring buffer. There are three operating modes for subsequent data processing.

The *normal mode* is suitable for offline analysis. When a buffer is full, a signal is sent to a transport daemon, which then syncs buffers to disk before they get overwritten. On average, disk throughput must be higher than event output. If all buffers are full, which can happen if disk bandwidth is lower than event generation throughput, then events are dropped. The number of such lost events is kept in the trace. Lost events can compromise further trace analysis. To avoid lost events, buffer size can be increased at trace start, but not while tracing, because buffers are allocated at trace setup. Hence, enough space must be reserved according to disk speed and maximum expected event throughput.

In cases where high throughput is expected and only the most recent data is desired, the *flight recorder* mode is well suited. In this mode, the ring buffer is overwritten until a condition occurs, and then the most recent events are written to disk. No event will be lost in this mode, but the actual trace duration depends on the buffer size.

The final supported mode is *live reading*. In this mode, buffers are flushed to disk regularly, before each read, to ensure consistency between different trace streams within a bounded delay (e.g. 1 second). This applies even for buffers that are only partially filled. The flush guarantees the consistency of the trace, avoiding the possibility of reception of older events out-of-order, which would otherwise appear to the analysis module out of chronological order.

LTTng uses per-CPU buffers to avoid data access synchronization between CPUs in multi-core architectures, and allows scalable tracing for large number of cores. Events are grouped in channels. Each channel represents an event stream and has its own buffers. Hence, the total number of allocated buffers is $(P \times C)$, where P is the number of processors and C the number of channels.

The Linux kernel is instrumented with over 150 tracepoints at key sites within each subsystem. Each tracepoint is compiled conditionally by the `CONFIG_MARKERS` configuration option. This option depends on `CONFIG_TRACEPOINTS`, which enables other kernel built-in instrumentation. Once tracepoints are compiled, they can be later activated to record a trace. A tracepoint compiled in, but not activated, reduces to a `no-op` instruction, hence the performance impact is undetectable. Kernel tracing is highly optimized, but the overall performance impact is proportional to the number of events generated per unit of time. Benchmarks show that each event requires 119 ns to process on 2 GHz Intel Xeon processor in cache-hot condition [3].

Another aspect to observe is the impact of compiled tracepoints on the kernel size. For each tracepoint, a new function and static data is added, and thus increases the kernel size. For kernel 2.6.38, compiling tracepoints results in an increase of about 122 kB of the vmlinuz image, or 1%. This includes LTTng tracepoints and other built-in kernel tracepoints.

2.1 Trace format

An event is composed of a timestamp, an event type and an arbitrary payload. The timestamp is mandatory to sort events according to time. The event type is used to determine the format of the payload.

The timestamp uses the hardware cycle counter and is converted to nanoseconds since the boot of the system. Special care is taken to guarantee that the time always increases monotonically between cores. The time is represented with 27 bit time delta, while the event id is five bits wide, for a total event header size of 32 bits. If no event occurs and the time delta overflows, which occur in the order of 100 ms on recent CPUs, an extended header is written. The timestamp is extended to 64 bits, while 16 bits are reserved for the event type.

The payload is an ordered set of fields, where the size and format are defined by the event type. Fields can be any standard C basic type, as well as variable size strings. The size of a field may differ from the actual type used in the code in order to compress the data. For example, an integer enum value can be recorded in the trace as a byte if the actual value is always less than 255, thus saving space.

2.2 Trace reading

The library liblttvtraceread is provided to read events from a trace for further processing. The library opens all files from the trace at the same time and returns events in total order. It provides a merged view of the trace, which abstracts the complexity of handling per-CPU and per-channel files. The library provides convenient functions to seek at a particular time in the trace performed by a binary search. The library handles endianness transformations automatically if necessary. It parses each event in the trace and returns the timestamp, event type and an array of parsed fields.

3 Recovering metrics

System metrics include CPU usage, memory allocation, file operations and I/O operations, such as network and block devices. The trace contains a comprehensive set of data from which system metrics can be measured and accounted to a process. This section presents events and algorithms used to perform these computations.

3.1 Trace metadata

The Trace metadata channel is used to declare the trace event types. It contains two pre-defined events types, core_marker_format and core_marker_id that are implicit and fixed. The purpose of core_marker_format is to list available channels and event types in the trace, along with the format of each field in their payload. Meanwhile core_marker_id lists event IDs and their corresponding channel and event name. This event inventory enables selection of the correct format to parse the payload of a particular event.

3.2 System state dump

The state dump consists of information about the system as it was at the beginning of the trace. The available information in the state dump is listed in Table 1. Events in the state dump can be split in two categories, static and variable.

Static information is invariant for the duration of the trace. This information may be referenced by other events. For example, the system call table enables system call IDs to be resolved to a meaningful name. This is necessary because system call IDs may differ from one system to another.

Variable information is the initial inventory of resources, which may be modified by later events. For example, the state dump contains the list of active processes at the beginning of the trace. This list is modified by fork and exit events, that add and remove processes respectively.

In addition, the special event statedump_end indicates the end of the state dump and metadata.

3.3 Process recovery

As presented in Section 3.2, the state dump includes the initial process list. This list includes the process id, thread id, thread group id, parent process id and a flag to indicate a kernel thread. While tracing, the event kernel.process_fork indicates a new process, while kernel.process_exit means a process terminated. The executable name can be deduced from the field filename of the fs.exec event. Until this event occurs, the executable name is the same as the parent process or the previously known name.

An additional step is required to link an event to a process. Since the process id is not saved on a per-event basis, this information must be recovered from the trace itself. The corresponding CPU on which an event occurs is known from the trace file id. The relation with the process id running on a given CPU can be established from the kernel.sched_schedule events. This event type contains two integers, prev_pid and next_pid. Each event occurring on a given CPU following a scheduling event can be related to the process next_pid, assuming no scheduling event is dropped.

3.4 CPU usage

CPU usage is a basic system metric for understanding the behavior of a system. To recover CPU usage per process on the system, we use scheduling events. A process has only two states, which are *scheduled* or *idle*. If no process runs on a given CPU, the kernel schedules the special thread swapper and we consider the CPU as idle. The total CPU time used by a process is the sum of intervals for which it was scheduled.

To chart the CPU usage according to time, we first divide the trace into fixed intervals to match the desired

State channel	Type	Description
`fd_state.file_descriptor`	Variable	File name and PID of opened fd
`irq_state.idt_table`	Static	Interrupts descriptor table for processor exceptions
`irq_state.interrupt`	Variable	Hardware IRQ ids, names and addresses
`module_state.list_module`	Variable	List of loaded kernel modules
`netif_state.network_ipv4_interface`	Variable	List of IPv4 interface names and IP addresses
`netif_state.network_ipv6_interface`	Variable	Same as IPv4, but for IPv6 network interfaces
`softirq_state.softirq_vec`	Static	List of SoftIRQ ids, names and addresses
`swap_state.statedump_swap_files`	Variable	Swap block devices
`syscall_state.sys_call_table`	Static	List of system call ids, names and addresses
`task_state.process_state`	Variable	List of active processes information
`vm_state.vm_map`	Variable	List of all memory mappings by process

Table 1: Available events in state dump

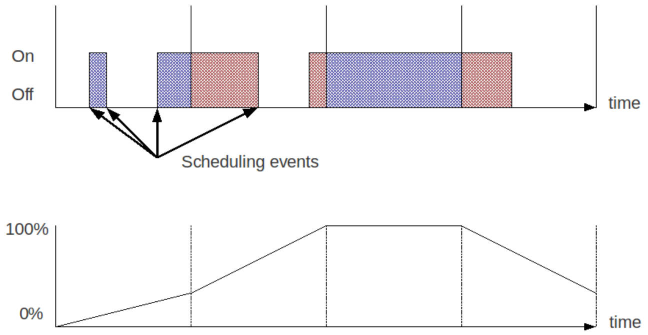


Figure 1: Average CPU usage recovery

resolution. Then the CPU usage of one interval is obtained by summing all overlapping scheduled intervals, as shown in Figure 1.

3.5 Memory usage

Tracepoints related to memory management provide information at the page level. The events `page_alloc` and `page_free` are encountered when a memory page related to a process is allocated or freed at the kernel level. The system memory usage at the beginning of the trace is given by the event `vm_state.vm_map` in metadata. Separate tracepoints report huge page operations. Thus the system memory usage on the system at a given time can be obtained by replaying subsequent page allocation and free events.

Note that the physical memory usage observed at the kernel level can differ from memory operations performed by the application. This behavior is intended to reduce the number of system calls that shrink or grow the heap and the stack of a process. The GNU libc may not release memory to the operating system after a call to `free()` to speedup subsequent allocations.

3.6 File operations

File operations are performed, for instance, through `open`, `read`, `write` and `close` system calls by user-space applications. These system calls are instrumented to compute file usage statistics, such as the number of bytes read or written, and to track opened files by a process. These events occur only if the system call succeeds. File operation events are recorded for files accessed through a file system as well as via network sockets.

The `fs.open` event contains the filename requested by the application and the associated file descriptor. All subsequent operations performed on this file descriptor by `fs.read` and `fs.write` contain the file descriptor and the byte count of the operation.

Tracing network operations on sockets is similar to files on a file system. Socket file descriptors are created upon `net.socket` and `net.accept` events, and byte transfer count is reported by `net.socket_sendmsg` and `net.socket_recvmsg` or `fs.write` and `fs.read`, depending on the system call involved.

3.7 Low-level network events

Low-level network events provide TCP and UDP packets fields, like ports and addresses involved in the communication. These extended network events are not enabled by default. To enable them, `ltt-armall` must be called with the switch `-n`.

Various metrics can be recovered directly from the payload of these events. The information provided by extended network events is a subset of what is available with tcpdump. Recording this information at the kernel level has the advantage of precise timestamps relative to system events. A network packet is always sent before being received, thus providing a convenient way to correlate traces from multiple systems without a common clock source. Average trace synchronization accuracy of 68.8 μs with TCP on standard 100 Mbit/s Ethernet has been achieved by using the Convex Hull algorithm [5].

3.8 Block Input/Output

In addition to file operations, underlying block device activity can be traced. Actions of block I/O scheduler, such as front and back merge requests, are recorded. Once the queue is ready, the event fs_issue_rq is emitted with the related sector involved. Each issue event is followed by the event fs_complete_rq when the request is completed. The delay between the issue and complete event indicates the latency of the disk operation.

Global disk offset can be observed with block level events. Disk offset is the difference between two consecutive sector requests to the device. For mechanical disks, high average offset between requests degrade throughput. Such bad performance is sometimes difficult to understand when it results from multiple processes performing independant operations on physically distant areas of the disk. The disk offset can be computed from fs_issue_rq event. The sector number from this event corresponds to the hardware sector. To relate this request to an inode, the bio_remap event is required. It contains the inode, the target device, the block requested at the file system level and the result of block remap, which is the hardware sector global to the device. Disk offset can thus be accounted on per inode basis.

4 LTTng kernel trace analyzer

We implemented a prototype of CPU usage computation to demonstrate the usefulness of the analysis. The software is coded in Java and uses the liblttvtraceread JNI interface. The system traced has two cores and runs Linux 2.6.36 with LTTng 0.249. The system is loaded with three cpuburn processes, started with one second interval between each. Figure 2 shows the trace loaded in the graphical interface.

The interface is divided in two parts. The top half contains the chart view of CPU usage according to time. The bottom half lists processes sorted according to total CPU usage. The chart has interactive features. Selecting an interval on the chart updates the process list statistics, while selecting a specific process displays its CPU usage. In Figure 2, the first cpuburn process is selected, and we can observe the CPU saturation occurring in this workload once these processes are started.

5 Future work

The goal of the LTTng project is to provide the community with the best tracing environment for Linux. We first present upcoming enhancements to the tracing infrastructure and future development of analysis tools.

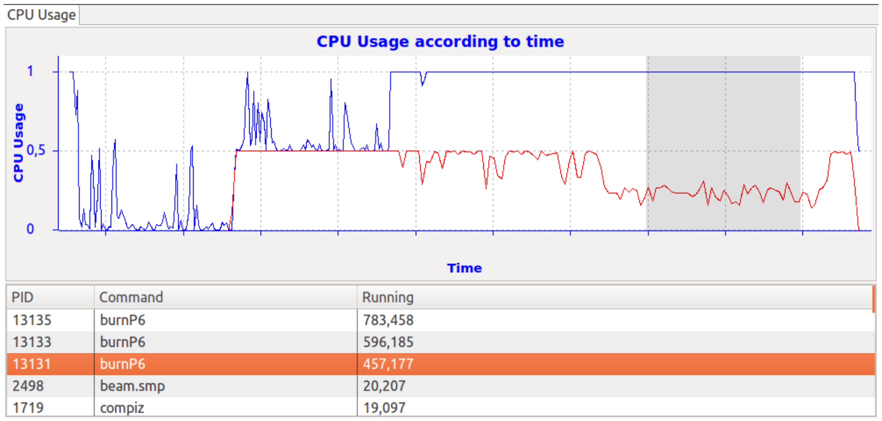
5.1 Tracing infrastructure

The current stable version of LTTng requires the kernel to be patched. Those patches are being converted to modules to work with vanilla kernel. The mainline kernel is already well instrumented. By default, those tracepoints will be available. Using stock kernels reduces the complexity of distributing LTTng, and helps towards making tracing ubiquitous.

The Common Trace Format (CTF) will be used to record the trace [2]. CTF has the ability to describe arbitrary sequence of binary objects. This new format will make it easier to define custom tracepoints in an extensible fashion.

Kernel tracing requires root privileges for security reasons. Enhancements to tracing tools will allow kernel tracing as normal user based on group membership, simplifying system administration. In addition, the unified git-like utility command-line tool lttng will control both kernel and user-space tracing on per-session basis. The live tracing mode will support reading data directly in memory, without flushing it to disk.

As presented in section 3.3, recovery of the PID of an event requires all scheduling events to be recorded. Omitting the PID from recorded events reduces the trace size, but increases analysis complexity. In some situations, appending a context to each event may be desirable to simplify event analysis, tolerate missing scheduling events, or to get values of performance counters



PID	Command	Running
13135	burnP6	783,458
13133	burnP6	596,185
13131	burnP6	457,177
2498	beam.smp	20,207
1719	compiz	19,097

Figure 2: LTTng trace analyzer prototype

when specific events occur. The next release will allow such event context to be added as a tracing option. Available context information includes, among others, the PID, the nice value, and CPU Performance Monitoring Unit (PMU) counters.

5.2 Trace analyzer

Analysis tools will require updates according to these changes to the tracing infrastructure. One purpose of the existing LTTng kernel patch is to extend the kernel instrumentation. Those tracepoints may not be available anymore with a vanilla kernel and the modules-based LTTng. We plan to propose additional tracepoints upstream to perform useful analysis for system administrators and software developers. For example, system calls are instrumented with entry and exit tracepoints, but arguments are not interpreted. Some of them may be pointers to structures, but saving an address is not useful for system metric recovery. Dereferencing the pointer and saving appropriate fields is required for many tracepoints.

Our current prototype implementation only includes the CPU usage. Our goal is to implement other presented metrics. System metrics computation can be integrated to the Eclipse LTTng plugin, part of the Eclipse Linux Tools project [6]. For this purpose, an open source native CTF Java reading library is being developed. Also, a command line tool similar to `top` is being developed to provide lightweight and live system metrics display from tracing data.

Further trace analysis are being developed to understand global performance behavior. Previous work has been done on critical path analysis of an application [4]. Our goal is to extend this analysis in two ways. First, computing the resource usage allows to get the cost of the critical path of an application. Secondly, analyzing communication paths between processes allows to recover links between distributed process. By combining those two analysis, we could provide a global view of the processing path of a client request made in a distributed application.

6 Conclusion

We showed that tracing can provide highly valuable data on a running system. This data can help to understand system-wide performance behavior. We presented the tracing infrastructure provided by LTTng and techniques to extract system metrics from raw events. Future developments to LTTng demonstrate the commitment to provide a state of the art tracing environment for the Linux community.

References

[1] Linux trace toolkit next generation. http://lttng.org.

[2] Mathieu Desnoyers. Common trace format specification v1.7. git://git.efficios.com/ctf.git.

[3] Mathieu Desnoyers. *Low-Impact Operating System Tracing*. 2009.

[4] P.M. Fournier and M.R. Dagenais. Analyzing blocking to debug performance problems on multi-core systems. *ACM SIGOPS Operating Systems Review*, 44(2):77–87, 2010.

[5] B. Poirier, R. Roy, and M. Dagenais. Accurate offline synchronization of distributed traces using kernel-level events. *ACM SIGOPS Operating Systems Review*, 44(3):75–87, 2010.

[6] D. Toupin. Using tracing to diagnose or monitor systems. *Software, IEEE*, 28(1):87–91, 2011.