# Scalable Tools for Non-Intrusive Performance Debugging of Parallel Linux Workloads

Robert Schöne* Joseph Schuchart* Thomas Ilsche* Daniel Hackenberg*

*ZIH, Technische Universität Dresden
{robert.schoene|joseph.schuchart|thomas.ilsche|daniel.hackenberg}@tu-dresden.de

## Abstract

There are a variety of tools to measure the performance of Linux systems and the applications running on them. However, the resulting performance data is often presented in plain text format or only with a very basic user interface. For large systems with many cores and concurrent threads, it is increasingly difficult to present the data in a clear way for analysis. Moreover, certain performance analysis and debugging tasks require the use of a high-resolution time-line based approach, again entailing data visualization challenges. Tools in the area of High Performance Computing (HPC) have long been able to scale to hundreds or thousands of parallel threads and are able to help find performance anomalies. We therefore present a solution to gather performance data using Linux performance monitoring interfaces. A combination of sampling and careful instrumentation allows us to obtain detailed performance traces with manageable overhead. We then convert the resulting output to the Open Trace Format (OTF) to bridge the gap between the recording infrastructure and HPC analysis tools. We explore ways to visualize the data by using the graphical tool Vampir. The combination of established Linux and HPC tools allows us to create an interface for easy navigation through time-ordered performance data grouped by thread or CPU and to help users find opportunities for performance optimizations.

## 1 Introduction and Motivation

GNU/Linux has become one of the most widely used operating systems, ranging from mobile devices, to laptop, desktop, and server systems to large high-performance computing (HPC) installations. Performance is a crucial topic on all these platforms, e.g., for extending battery life in mobile devices or to ensure maximum ROI of servers in production environments. However, performance tuning is still a complex task that often requires specialized tools to gain insight into the behavior of applications. Today there are only a small number of tools available to developers for understanding the run-time performance characteristics of their code, both on the kernel and the user land side. Moreover, the increasing parallelism of modern multi- and many-core processors creates an additional challenge since scalability is usually not a major focus of standard performance analysis tools. In contrast, scalability of applications and performance analysis tools have long been topics in the High Performance Computing (HPC) community. Nowadays, 96.4 % of the 500 fastest HPC installations run a Linux OS, as compared to 39.6 % in 2003[1]. Thus, the HPC community could benefit from better integration of Linux specific performance monitoring interfaces in their tools, as these are currently targeting parallel programs and rely on instrumenting calls to parallelization libraries such as the Message Passing Interface (MPI) and OpenMP. On the other hand, the Linux community could benefit from more scalable tools. We are therefore convinced that the topic of performance analysis should be mutually solved by bringing together the expertise of both communities.

In this paper, we present an approach towards scalable performance analysis for Linux using the perf infrastructure, which has been introduced with Linux 2.6.31 [8] and has undergone intensive development since then. This infrastructure allows users to access hardware performance counters, kernel-specific events, and information about the state of running applications. Additionally, we present a new visualization method for ftrace-based kernel instrumentation.

[1] Based on November 2003 and November 2013 statistics on http://top500.org

| Measurement Type | Kernel Interface | Common Userspace Tools and Libraries |
|---|---|---|
| Instrumentation | ptrace | gdb, strace, ltrace |
| | ftrace | trace-cmd, kernelshark, ktap |
| | kernel tracepoints | LTTng, SystemTap, ktap, perf userspace tools |
| | dynamic probes | SystemTap, ktap, perf userspace tools |
| Sampling | perf events | perf userspace tools, PAPI |
| | OProfile (kernel module) | OProfile daemon and tools |

Table 1: Common Linux Performance Analysis Interfaces and Tools

The remainder of this paper is structured as follows: Section 2 presents an overview of existing Linux performance analysis tools. Section 3 outlines the process of acquiring and processing performance data from the perf and ftrace infrastructures followed by the presentation of different use-cases in Section 4.

## 2 Linux Performance Monitoring Interfaces and Established Tools

Several interfaces are available in the Linux kernel to enable the monitoring of processes and the kernel itself. Based on these interfaces, well-established userspace tools and libraries are available to developers for various monitoring tasks (see Table 1). The *ptrace* [15] interface can be used to attach to processes but is not suitable for gaining information about the performance impact of kernel functions. *ftrace* [7] is a built-in instrumentation feature of the Linux kernel that enables kernel function tracing. It uses the `-pg` option of `gcc` to call a special function from every function in a kernel call. This special function usually executes NOPs. An API, which is located in the Debugfs, can be used to replace the NOPs with a tracing function. `trace-cmd` [25] is a command line tool that provides comfortable access to the ftrace functionality. *KernelShark* [26] is a GUI for `trace-cmd`, which is able to display trace information about calls within the Linux kernel based on ftrace events. This allows users to understand the system behavior, e.g., which processes trigger kernel functions and how tasks are scheduled. However, the KernelShark GUI is not scalable to large numbers of CPU cores and does not provide integration of sampling data, e.g., to present context information about application call-paths. Nevertheless, support for ftrace is currently being merged into the perf userspace tools [16]. Kernel *tracepoints* [3] are instrumentation points in different kernel modules that provide event-specific information, e.g., which process is scheduled to which CPU for a scheduling event or what hints have been used when allocating pages. *kprobes* are dynamic tracepoints that can be added to the kernel at run-time [12] by using the `perf probe` command. Such probes can also be inserted in userspace programs and libraries (*uprobes*). The *perf_event* infrastructure can handle kprobes and uprobes as well as tracepoint events. This allows the perf userspace tools to record the occurrences of these events and to integrate them into traces. The *Linux Trace Toolkit next generation* (*LTTng*) [10, 11] is a tracing tool that allows users to measure and analyze user space and kernel space and is scalable to large core counts. It writes traces in the *Common Trace Format* which is supported by several analysis tools. However, these tools do not scale well to traces with large event counts. *SystemTap* [28] provides a scripting interface to access and react on kernel probes and ftrace points. Even though it is possible to write a generic (kernel) tracing tool with `stap` scripts, it is not intended for such a purpose. *ktap* is similar to SystemTap with the focus on kernel tracing. It supports tracepoints, dynamic probes, ftrace, and others.

In addition to the instrumentation infrastructure support in the kernel, measurement points can also be triggered by sampling. The *perf_event* infrastructure provides access to hardware-based sampling that is implemented on x86 processors with performance monitoring units (PMUs) that trigger APIC interrupts [5, 14]. On such an interrupt, the call-graph can be captured and written to a trace, which is usually done with the `perf record` command-line tool but can also be achieved with low-level access to a memory-mapped buffer that is shared with the kernel. In a post-mortem step, tools like `perf script` and `perf report` use debugging symbols to map the resulting events in a trace file recorded by `perf record` to function names. *PAPI* [6, 23] is the de facto standard library for reading performance counter information and is supported by most HPC tools. On current

Linux systems, PAPI uses the perf_event interface via the libpfm4 library. In addition to performance counter access, PAPI is also able to use this interface for sampling purposes. The OProfile kernel module [19] is updated regularly to support new processor architectures. It provides access to hardware PMUs that can be used for sampling, e.g., by the OProfile daemon [20].

However, none of the Linux performance analysis tools are capable of processing very large amounts of trace data, and none feature scalable visualization interfaces. Scalable HPC performance analysis tools such as Vampir [24], Score-P [18], HPCToolkit [1], and TAU [27], on the other hand, usually lack the close integration with the Linux kernel's performance and debugging interfaces.

## 3 Performance Data Acquisition and Conversion

In this section, we discuss our approach to obtaining performance data using standard tools and interfaces and how we further process the data to make it available to scalable analysis tools.

### 3.1 Data Acquisition with perf and ftrace

We use `perf record` to capture hardware-counter-based samples and selected tracepoints. In more detail, we use the following event sources:

**`cpu-cycles`**
: This event is used as a sampling timer. Unlike typical alerts or timers, the cpu-cycles counter does not increase when the CPU is idle. Information about idling CPUs or tasks is not crucial for performance analysis and a lower interrupt rate in such scenarios minimizes the sampling overhead.

**`sched_process_{fork|exec|exit}`**
: These tracepoint events are used to track the creation and termination of processes.

**`sched_switch`**
: This tracepoint event is used to track processes on the CPUs. It provides knowledge about when which task was scheduled onto which CPU. The state of the task that is scheduled away is associated to the event in order to distinguish between voluntary sleep (state S), un-interruptible sleep (state D, usually I/O), or preemption (state R)[2].

**`instructions|cache-misses|...`**
: Other performance counters can be included in the timeline to get a better understanding of the efficiency of the running code. For example, the *instruction* counter allows us to determine the instruction per cycle (IPC) value for tasks and CPUs and adding *cache-misses* provides insights into the memory usage.

As an alternative to sampling, instrumentation can provide fine-grained information regarding the order and context of function calls. For debugging purposes, sampling is not a viable option. Thus, we use the kernel tracing infrastructure *ftrace* to analyze kernel internal behavior. One alternative would be a combination of *trace-cmd* and *KernelShark*. However, KernelShark is limited in terms of scalability and visualization of important information. Instead of `trace-cmd` we use a shell script to start and stop the kernel monitoring infrastructure ftrace. The script allows us to specify the size of the internal buffer for events and filters that can be passed to ftrace in the respective `debug fs` files. To create the trace, we enable the *function_graph* tracer and set the options to display overruns, the CPU, the process, the duration, and the absolute time. The script then starts the recording by enabling ftrace and stops it when the recording time expires.

### 3.2 Conversion to Scalable Data Formats

The `perf record` tool and its underlying file format are designed to induce only minimal overhead during measurement. It therefore simply dumps data from the kernel buffer directly into a single file without any distinction between process IDs or CPUs. This file can be used in a follow-up step to create a profile based on the recorded trace data using `perf report`. External tools can be used with `perf script` to analyze the trace data. However, the simple file structure resulting from the low-overhead recording process has negative side effects on the scalability of the data format. Parallel parsing of a single file is impeded by the variable length of single trace entries and the mixture of management information (e.g., task creation and termination) with performance event information from sampling.

[2] cf. `man top`

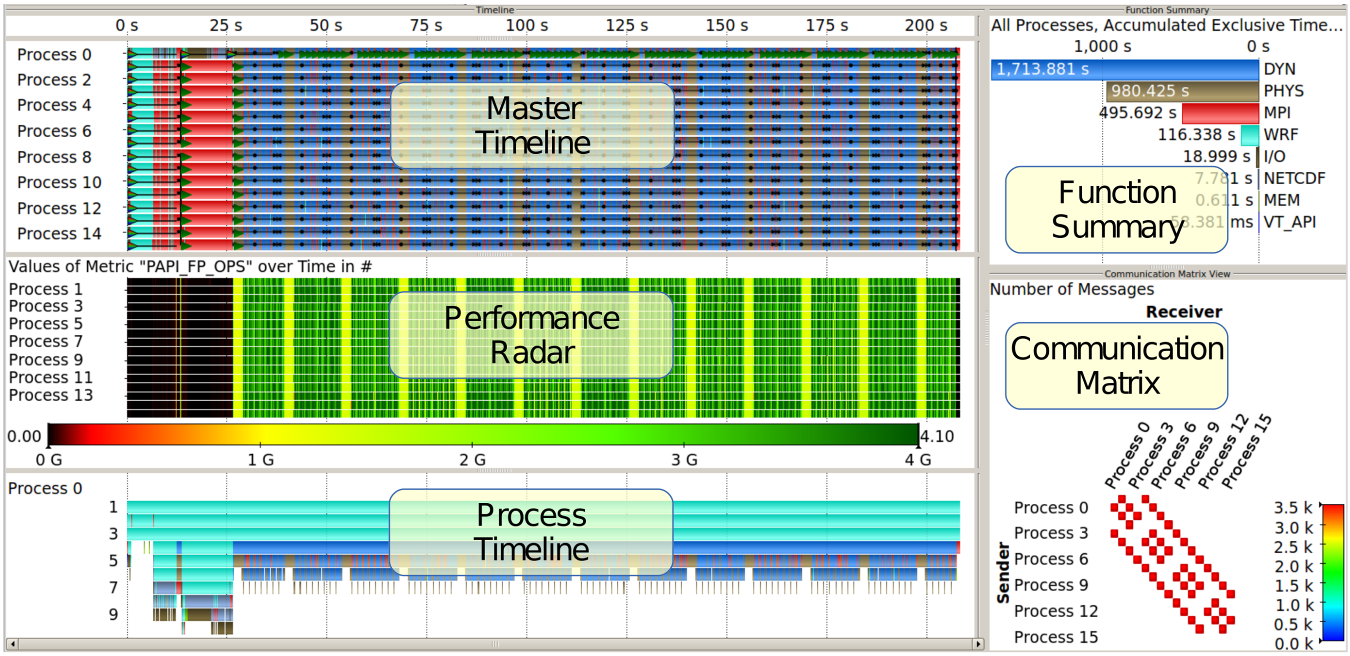


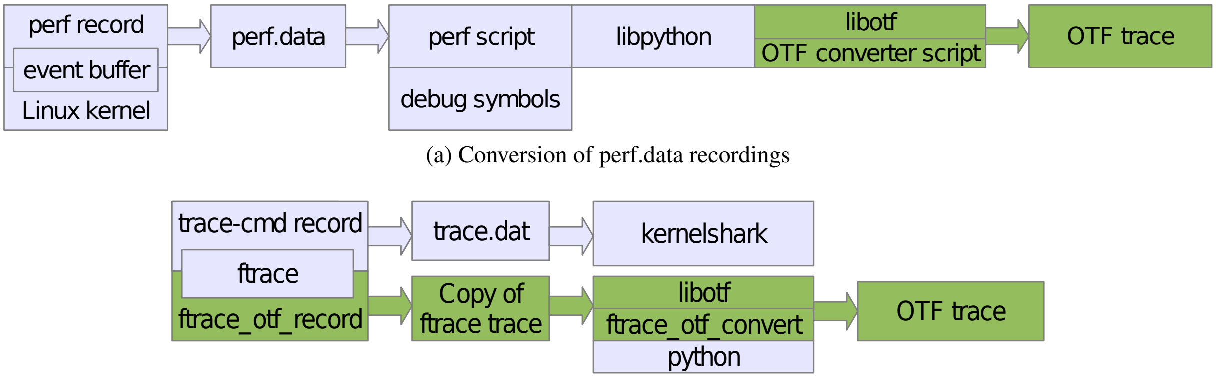
Figure 1: Vampir visualization of a trace of the HPC application WRF, including the *Master Timeline* showing the parallel process activity (different function calls in different colors, MPI messages as black lines aggregated in bursts), the *Performance Radar* depicting performance metrics such as hardware counter readings, and the *Process Timeline* with the call-stack of one process. The right side contains the *Function Summary* that provides a function profile and a *Communication Matrix* depicting a profile about the communication between the parallel processes. The trace is available for download at http://vampir.eu.

### 3.2.1 Scalable Trace Formats

Scalable performance analysis tools commonly used by the HPC community make use of scalable formats such as OTF [17], OTF2 [9], CTF [4], and HPCTRACE [1]. The *Open Trace Format* (OTF) was designed for use with VampirTrace [24] and enables parallel reading and writing of trace files. The format is built around the concept of event streams, which can hold trace information of one or more parallel execution entities (processes, threads, GPU streams). Event properties, such as names of processes and functions as well as grouping information, can be defined locally for one stream or globally for all streams. This separation of different event streams as well as meta-data is important for efficiently reading and writing event traces in parallel, which has already been demonstrated on a massively parallel scale with more than 200,000 event streams [13]. The data itself is encoded in ASCII format and can be compressed transparently. The successor of this trace format is OTF2 [9]. It has a similar structure but allows for more efficient (binary) encoding and processing. OTF2 is part of the Score-P performance measurement environment [18].

We use Vampir for the visualization of the generated OTF files. Figure 1 shows the visualization of a trace of a typical MPI application recorded using VampirTrace. Vampir is designed to display the temporal relation between parallel processes as well as the behavior of individual processes, to present performance metrics, e.g., hardware counters, MPI communication and synchronization events. Additionally, Vampir derives profiling information from the trace, including a function summary, a communication matrix, and I/O statistics. Starting from an overall view on the trace data, Vampir enables the user to interactively browse through the trace data to find performance anomalies. By providing the capability of filtering the trace data, Vampir helps users to cope with potentially large amounts of trace data that has been recorded by the measurement infrastructure. Moreover, it provides a client-server based infrastructure using a parallel analysis server that can run on multiple nodes to interactively browse through the large amounts of trace data.

In general, the trace files can be written and read through an open source library to enable users to analyze the traces with custom tools. VampirTrace and OTF are



(a) Conversion of perf.data recordings

(b) Conversion of ftrace recordings

Figure 2: Toolchains for recording and converting performance data of Linux performance monitoring tools.

bundled with command-line tools for analyzing and processing OTF traces. Since the focus of these tools has been instrumentation based recording, there is no dedicated call-path sample record type in OTF or any of the other formats supported by Vampir so far. Therefore, the call-path sample information from perf.data is mapped to enter- and leave-function events typically obtained through instrumentation. Introducing support for sampled events into the full tool chain is currently work in progress.

### 3.2.2 Conversion of Trace Data

To convert the perf.data information into a scalable file format, we use the python interface provided by `perf script` and the python-bindings of OTF. Additionally, we patched perf script to pass dynamic symbol object information to the conversion script[3]. Based on the PID and CPU information within every sample, we are able to create two different traces: a task-centric and a CPU-centric trace. The conversion process depicted in Figure 2a is still sequential due to the limitations of the perf.data file format. For a CPU-centric view, this limitation could be overcome with multiple perf data files – one per CPU – which would be feasible with the existing tool infrastructure. However, task migration activities and their event presentation do pose a major challenge for a task-centric view since information on individual tasks would be scattered among multiple data files.

Note that perf.data information that is gathered in a task-specific context does not provide information about the CPU that issued a specific event. Thus, we can only create task-centric traces in this case. Information that is gathered in a CPU-specific context allows us to create both CPU-centric and task-centric traces.

Processing information provided by ftrace is straightforward as the exact enter and exit events are captured. Thus, we use the OTF python bindings to write events whenever a function is entered or exited. We concurrently generate two traces – a CPU-centric trace and a process-centric trace. If a function has been filtered out or the process has been unscheduled in between, enter events are written to match the current stack depth. One challenge for the trace generation is the timer resolution of ftrace events, which is currently in microseconds. This leads to a lower temporal accuracy within the traces as function call timer resolution is nanoseconds. The difference of these timer sources adds uncertainty. However, the order and context of the calls stay correct, thereby allowing enthusiasts to understand causal relations of function calls within the kernel. The full toolchain overview is depicted in Figure 2b.

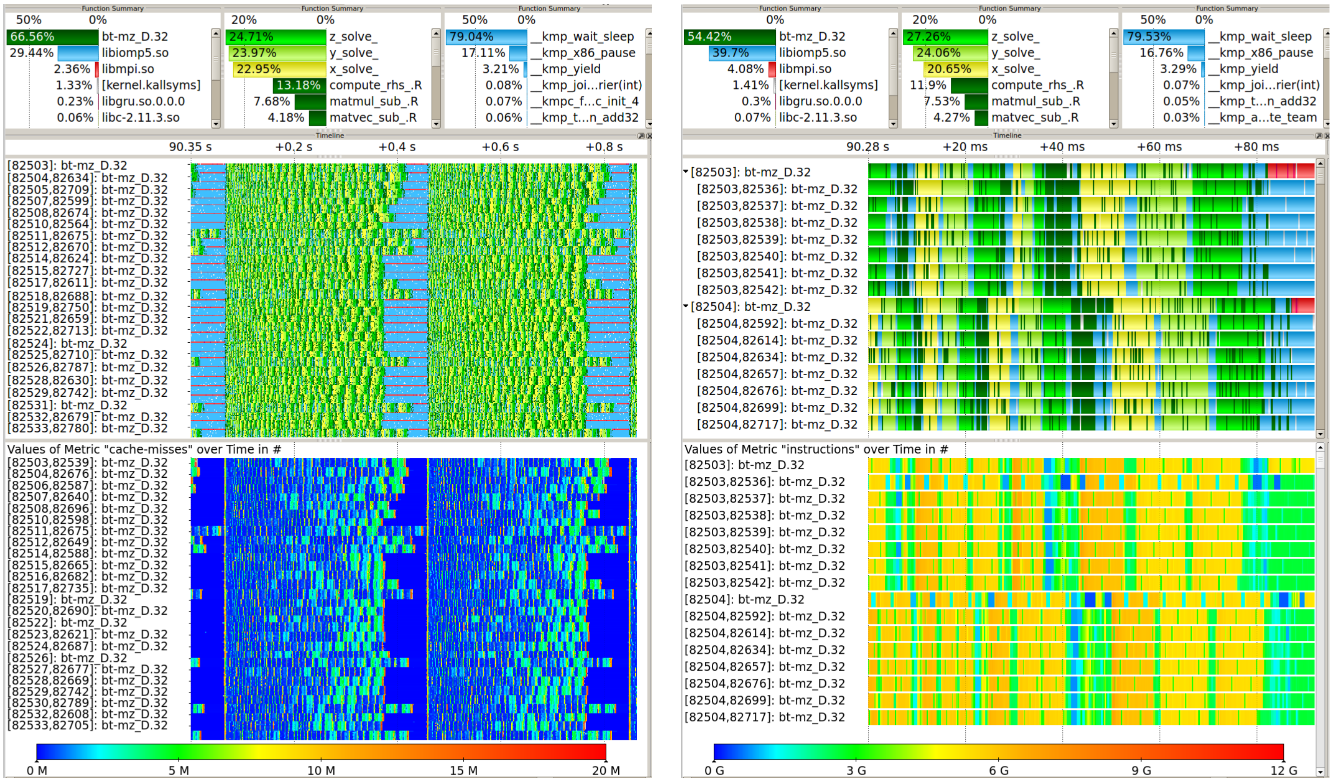
# 4 Examples and Results

## 4.1 Analyzing Parallel Scientific Applications

This example demonstrates the scalability of our approach. We use the perf-based tool infrastructure presented in Section 3 to analyze a hybrid parallel application. The target application is bt-mz of the NAS parallel benchmark suite [2] (Class D, 32 MPI processes with 8 OpenMP threads each).

We run and post-process this workload on a NUMA shared memory system with a total of 512 cores and

[3] See https://lkml.org/lkml/2014/2/18/57



Function Summary (a)

| 50% 0% | |
|---|---|
| 66.56% | bt-mz_D.32 |
| 29.44% | libiomp5.so |
| 2.36% | libmpi.so |
| 1.33% | [kernel.kallsyms] |
| 0.23% | libgru.so.0.0.0 |
| 0.06% | libc-2.11.3.so |

| 20% 0% | |
|---|---|
| 24.71% | z_solve_ |
| 23.97% | y_solve_ |
| 22.95% | x_solve_ |
| 13.18% | compute_rhs_.R |
| 7.68% | matmul_sub_.R |
| 4.18% | matvec_sub_.R |

| 50% 0% | |
|---|---|
| 79.04% | __kmp_wait_sleep |
| 17.11% | __kmp_x86_pause |
| 3.21% | __kmp_yield |
| 0.08% | __kmp_joi...rier(int) |
| 0.07% | __kmpc_f...c_init_4 |
| 0.06% | __kmp_t...n_add32 |

Function Summary (b)

| 0% | |
|---|---|
| 54.42% | bt-mz_D.32 |
| 39.7% | libiomp5.so |
| 4.08% | libmpi.so |
| 1.41% | [kernel.kallsyms] |
| 0.3% | libgru.so.0.0.0 |
| 0.07% | libc-2.11.3.so |

| 20% 0% | |
|---|---|
| 27.26% | z_solve_ |
| 24.06% | y_solve_ |
| 20.65% | x_solve_ |
| 11.9% | compute_rhs_.R |
| 7.53% | matmul_sub_.R |
| 4.27% | matvec_sub_.R |

| 50% 0% | |
|---|---|
| 79.53% | __kmp_wait_sleep |
| 16.76% | __kmp_x86_pause |
| 3.29% | __kmp_yield |
| 0.07% | __kmp_joi...rier(int) |
| 0.05% | __kmp_t...n_add32 |
| 0.03% | __kmp_a...te_team |

Timeline
90.35 s
+0.2 s
+0.4 s
+0.6 s
+0.8 s
[82503]: bt-mz_D.32
[82504,82634]: bt-mz_D.32
[82505,82709]: bt-mz_D.32
[82507,82599]: bt-mz_D.32
[82508,82674]: bt-mz_D.32
[82510,82564]: bt-mz_D.32
[82511,82675]: bt-mz_D.32
[82512,82670]: bt-mz_D.32
[82514,82624]: bt-mz_D.32
[82515,82727]: bt-mz_D.32
[82517,82611]: bt-mz_D.32
[82518,82688]: bt-mz_D.32
[82519,82750]: bt-mz_D.32
[82521,82659]: bt-mz_D.32
[82522,82713]: bt-mz_D.32
[82524]: bt-mz_D.32
[82525,82710]: bt-mz_D.32
[82526,82787]: bt-mz_D.32
[82528,82630]: bt-mz_D.32
[82529,82742]: bt-mz_D.32
[82531]: bt-mz_D.32
[82532,82679]: bt-mz_D.32
[82533,82780]: bt-mz_D.32
Values of Metric "cache-misses" over Time in #
[82503,82539]: bt-mz_D.32
[82504,82676]: bt-mz_D.32
[82506,82587]: bt-mz_D.32
[82507,82640]: bt-mz_D.32
[82508,82696]: bt-mz_D.32
[82510,82598]: bt-mz_D.32
[82511,82675]: bt-mz_D.32
[82512,82649]: bt-mz_D.32
[82514,82588]: bt-mz_D.32
[82515,82665]: bt-mz_D.32
[82516,82682]: bt-mz_D.32
[82517,82735]: bt-mz_D.32
[82519]: bt-mz_D.32
[82520,82690]: bt-mz_D.32
[82522]: bt-mz_D.32
[82523,82621]: bt-mz_D.32
[82524,82687]: bt-mz_D.32
[82526]: bt-mz_D.32
[82527,82677]: bt-mz_D.32
[82528,82669]: bt-mz_D.32
[82529,82742]: bt-mz_D.32
[82530,82789]: bt-mz_D.32
[82532,82608]: bt-mz_D.32
[82533,82705]: bt-mz_D.32
0 M
5 M
10 M
15 M
20 M
Timeline
90.28 s
+20 ms
+40 ms
+60 ms
+80 ms
[82503]: bt-mz_D.32
[82503,82536]: bt-mz_D.32
[82503,82537]: bt-mz_D.32
[82503,82538]: bt-mz_D.32
[82503,82539]: bt-mz_D.32
[82503,82540]: bt-mz_D.32
[82503,82541]: bt-mz_D.32
[82503,82542]: bt-mz_D.32
[82504]: bt-mz_D.32
[82504,82592]: bt-mz_D.32
[82504,82614]: bt-mz_D.32
[82504,82634]: bt-mz_D.32
[82504,82657]: bt-mz_D.32
[82504,82676]: bt-mz_D.32
[82504,82699]: bt-mz_D.32
[82504,82717]: bt-mz_D.32
Values of Metric "instructions" over Time in #
[82503]: bt-mz_D.32
[82503,82536]: bt-mz_D.32
[82503,82537]: bt-mz_D.32
[82503,82538]: bt-mz_D.32
[82503,82539]: bt-mz_D.32
[82503,82540]: bt-mz_D.32
[82503,82541]: bt-mz_D.32
[82503,82542]: bt-mz_D.32
[82504]: bt-mz_D.32
[82504,82592]: bt-mz_D.32
[82504,82614]: bt-mz_D.32
[82504,82634]: bt-mz_D.32
[82504,82657]: bt-mz_D.32
[82504,82676]: bt-mz_D.32
[82504,82699]: bt-mz_D.32
[82504,82717]: bt-mz_D.32
0 G
3 G
6 G
9 G
12 G

(a) All processes, two iterations with cache-misses per second

(b) Two processes, close-up inside one iteration which instructions per second

Figure 3: Trace of a parallel application using OpenMP and MPI. Program execution is colored green/yellow. Thread synchronization via OpenMP is colored blue. Process synchronization via MPI is colored red.

8 TiB main memory[4]. To generate the trace, we only use features that are available for unprivileged users in standard Linux environments. We utilize `perf record` with default settings for the cycles, instructions, and cache-misses hardware events and enabled call-graph tracing.

Additional cores are reserved for perf to reduce the perturbation of the application due to the measurement. The recording operates at the limit of the system I/O capacity, so that a number of chunks are lost. According to internal measurements of the application, its execution time increases from 70.4 s to 95.8 s when comparing a regular and a measured execution. Considering the scale of the application and three hardware counters with a relatively high recording frequency, the overhead is acceptable. The resulting perf.data file contains 166 million events in 16 GiB. After the conversion process, the resulting compressed OTF trace has a size of 2.1 GiB.

Figure 3a visualizes an excerpt of approx. 1 second of the application execution in Vampir. For a more concise visualization, we filter the shepherd threads of the OpenMP run-time library as well as the `mpirun` and helper processes. These tasks monitor the the OpenMP threads and the MPI environment for failures. They are recorded along the other tasks but do not show any regular activity during the execution. The figure contains three summaries of function activity: the fraction of time spent in each dso, the time share of functions in the binary, and the time share of functions in the OpenMP library. It also contains a timeline with the current function and a heat-map of cache-misses for all processes respectively. The visualization contains two iterations of the application execution. After each iteration, a global synchronization (red) between all MPI ranks is performed. The computation threads also synchronize (light blue) with their respective master threads. At the very beginning of each iteration, there is a short phase with a high cache miss rate after which the miss rate drops. Towards the end of each iteration, the cache miss

[4] SGI UV2000 with 64 socket Intel Sandy Bridge E5-4650L @ 2.6 GHz

rate also increases and so does the run-time of the repeated x/y/z_solve functions. A closer look inside an iteration is shown in Figure 3b, which is focused on two processes (16 compute threads total). Within each process, the x/y/z_solve and a few other functions are repeatedly executed with OpenMP synchronizations in between. Note that there is some sampling noise of other function calls within the x/y/z_solve that cannot be filtered due to imperfect call-path information. The performance radar shows that the functions x/y/z_solve have different typical instruction rates. Two threads (82536 and 82504) show regular drops in the instruction rate and similar drops in the cycles rate (not shown in the picture). This is likely due to them being preempted in favor of another task. As a consequence, the synchronization slows down the entire thread groups. Moreover, there is a regular diagonal pattern of short drops in the instruction rate. This is likely a result of OS-noise similar to the effects that we analyze in Section 4.4.

### 4.2 Analyzing the Behavior of a Web Server

In addition to analyzing one (parallel) application, perf can also be used for system analyses. To demonstrate these capabilities, we ran perf as a privileged user on a virtual machine running a private ownCloud[5] installation using the Apache2 webserver and a MySQL database. The virtual machine is hosted on a VMware installation and is provided with 2 cores and 4 GB of memory. The recording was done using the -a flag to enable system-wide recording in addition to call-graph sampling. The visualization of the resulting trace is shown in Figure 4. The recorded workload consisted of six WebDAV clients downloading 135 image files with a total size of 500 MB per client.

The parallel access of the clients is handled through the Apache2 mpm_prefork module, which maintains a pool of server processes and distributes requests to these workers. This is meant to ensure scalable request handling with a high level of separation between the workers and is recommended for PHP applications. The process pool can be configured with a minimum and maximum number of server processes based on the number of expected clients. However, the high load from the clients downloading files in parallel in conjunction with the small number of available cores leads to an overload that manifests itself through the parallel server processes spending much time in the idle(R) state in which processes are run-able and represented in the kernel's task queue but not actually running, e.g., they are not waiting for I/O operations to complete. These involuntary context switches are distinctive for overload situations and are also reflected by the high number of context switches, as can be seen in the display in the middle of the figure.

The MySQL database is involved in the processing as it stores information about the files and directories stored on the server. Every web-server instance queries the database multiple times for each client request. Since the run-times of the database threads between voluntary context switches (waiting for requests) are relatively short, the threads are not subject to involuntary switches.

In addition to the run-time behavior of the processes and their scheduling, we have also captured information about the network communication of the server. This is depicted in the lower three displays of Figure 4. To accomplish this, two additional events have been selected during the recording: net:net_dev_xmit reflecting the size of the socket buffers handed to the network device for transmission to clients and net:netif_receive_skb for received socket buffers. Note that this information does not necessarily reflect the exact data rate on the network but can provide a good estimate of the network load and how it can be attributed to different processes.

### 4.3 Analyzing Parallel Make Jobs

In addition to analyzing the performance of server workloads, perf can also be used to record the behavior of desktop machines. As an example, we use the compilation process of the perf project using the GCC 4.8.0 compiler. As in the previous example, perf has been run as a privileged user in order to capture scheduling and migration events in addition to the cycles and page-faults counter. Figure 5 shows the compilation process in four different configurations, from a serial build to a highly parallel build on a four core desktop machine (Intel Core i7-2620M). The serial compilation is depicted in Figure 5a and reveals that one compilation step requires significantly more time to finish than all other steps. Figure 5b depicts a parallel make to compensate for the wait time (and to better utilize the

[5] See http://owncloud.org/

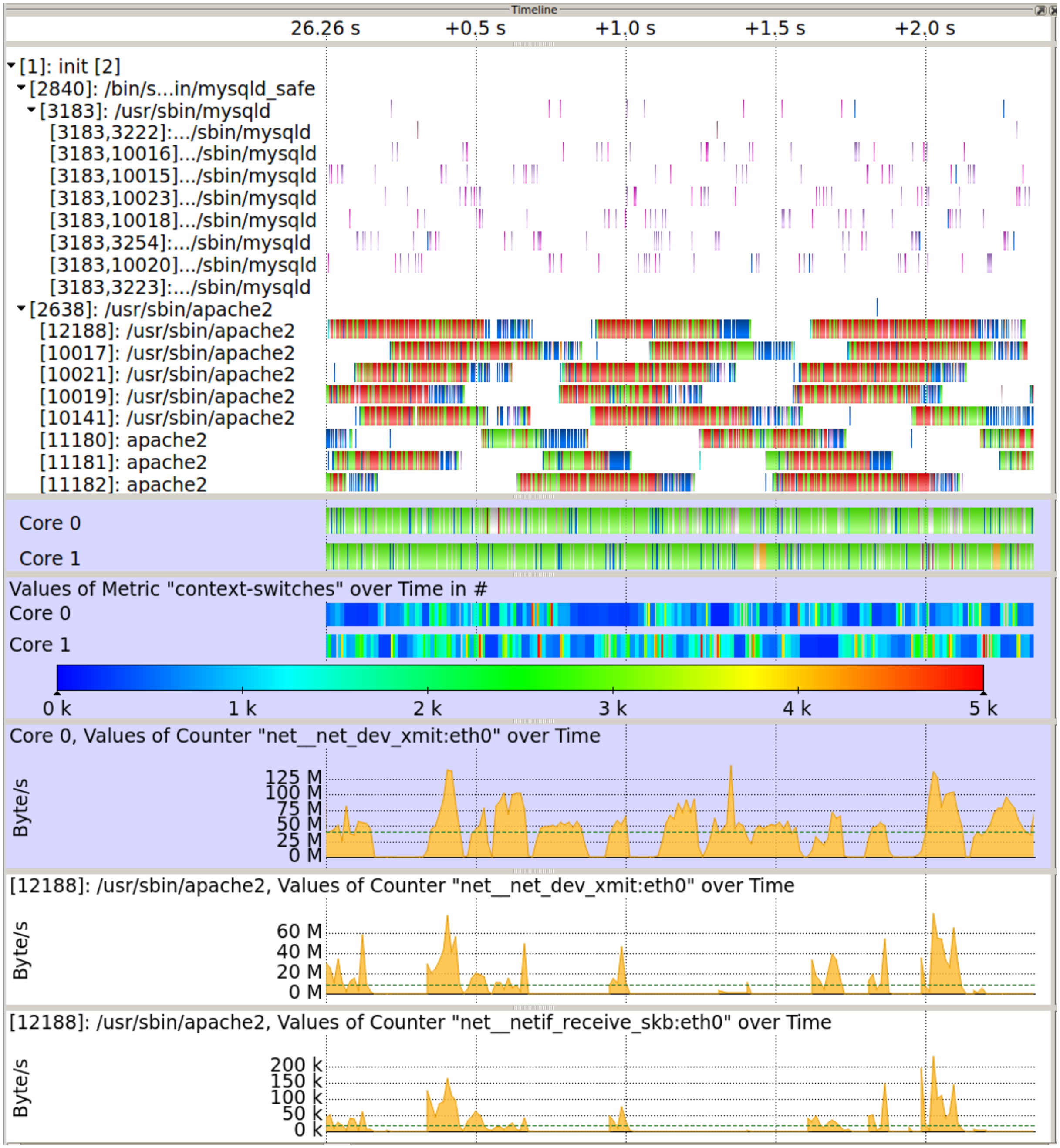
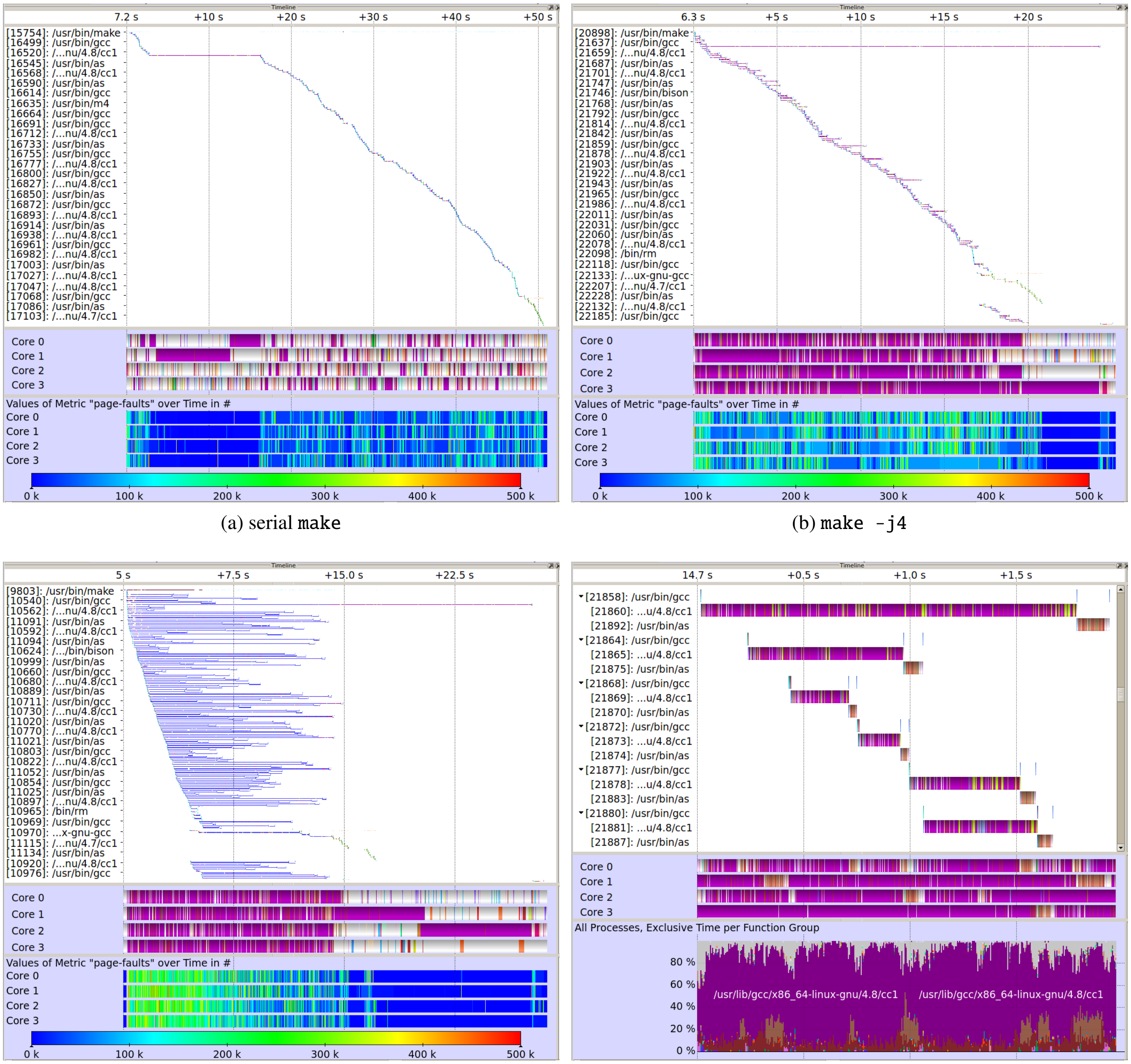

Timeline
26.26 s
+0.5 s
+1.0 s
+1.5 s
+2.0 s
[1]: init [2]
[2840]: /bin/s...in/mysqld_safe
[3183]: /usr/sbin/mysqld
[3183,3222]:.../sbin/mysqld
[3183,10016]:.../sbin/mysqld
[3183,10015]:.../sbin/mysqld
[3183,10023]:.../sbin/mysqld
[3183,10018]:.../sbin/mysqld
[3183,3254]:.../sbin/mysqld
[3183,10020]:.../sbin/mysqld
[3183,3223]:.../sbin/mysqld
[2638]: /usr/sbin/apache2
[12188]: /usr/sbin/apache2
[10017]: /usr/sbin/apache2
[10021]: /usr/sbin/apache2
[10019]: /usr/sbin/apache2
[10141]: /usr/sbin/apache2
[11180]: apache2
[11181]: apache2
[11182]: apache2
Core 0
Core 1
Values of Metric "context-switches" over Time in #
Core 0
Core 1
0 k
1 k
2 k
3 k
4 k
5 k
Core 0, Values of Counter "net__net_dev_xmit:eth0" over Time
Byte/s
125 M
100 M
75 M
50 M
25 M
0 M
[12188]: /usr/sbin/apache2, Values of Counter "net__net_dev_xmit:eth0" over Time
Byte/s
60 M
40 M
20 M
0 M
[12188]: /usr/sbin/apache2, Values of Counter "net__netif_receive_skb:eth0" over Time
Byte/s
200 k
150 k
100 k
50 k
0 k

Figure 4: Vampir trace visualization of a system running an Apache2 web sever and a MySQL database. Some processes were filtered. The top display shows the thread-centric view followed by the CPU-centric view and the number of context switches per core. The lower part of the figure contains the average socket buffer size transmitted (`net_dev_xmit`) per time for core 0 and for one of the Apache2 processes as well as the average socket buffer size received per time by that process. The executed code parts are colored as follows: MySQL in purple, PHP5 in green, and `libc` in blue. For the cores, the function `native_safe_halt` is colored orange and is used on Core 1 when it is not needed toward the end. The `idle(R)` state is colored red.



(a) serial make

(b) make -j4

(c) make -j

(d) make -j4 (close-up)

Figure 5: System traces of a desktop machine compiling perf (as shipped with Linux 3.8.0) in different configurations: (a) serial make; (b) with four parallel make jobs (using -j4); (c) with unlimited number of make jobs (using -j); and (d) a close-up view of the trace shown in (b) showing six make jobs. Figures (a) – (c) each show both the process-centric view (top) and the CPU-centric view (bottom) of the function execution in addition to a display of the page faults that occurred during execution. Figure (d) also shows a summary of all executed processes during the depicted time-frame. The colors are as follows: `cc1` depicted in purple, `idle(R)` in blue, `as` in dark brown, `libc` in light brown, and kernel symbols in light gray. All figures only contain the compilation and linking steps, the preceding (sequential) configuration steps are left out intentionally.

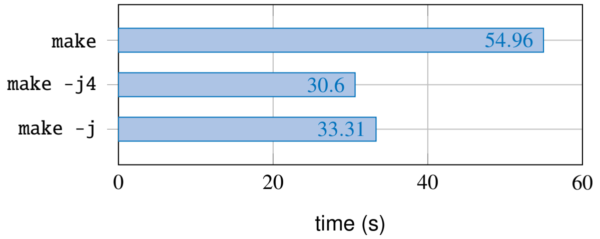


Figure 6: Time required for building the perf project using different configurations for parallel make (1, 4, unlimited).

available four CPU cores). It shows that the compilation proceeds even though the long running compilation step is not finished yet. Only at the very end, the linking step has to be deferred until all make jobs are finished. A subset of the parallel make steps is depicted in Figure 5d to visualize the process structure (`gcc` spawns the processes `cc` and `as` and waits for their execution to finish) and the actual parallel execution. The figure also shows the executed applications and library functions, e.g., `cc1`, `gcc`, `as`, and kernel symbols.

Another attempt to speed up the compilation (and to compensate for possible I/O idle times) is to spawn even more processes. This can be done using `make -j` without specifying the number of parallel jobs. In that case, `make` launches as many jobs as possible with respect to compilation dependencies. This can lead to heavy oversubscription even on multi-core systems, possibly causing a large number of context switches and other performance problems. The behavior of a highly parallel make is depicted in Figure 5c, which also shows an increased number of page faults as a consequence of the high number of context switches. Overall, compiling the perf project with `make -j4` is slightly faster (30.6 s) compared to using `make -j` (33.31 s), as shown in Figure 6.

### 4.4 Analyzing OS Behaviour with ftrace

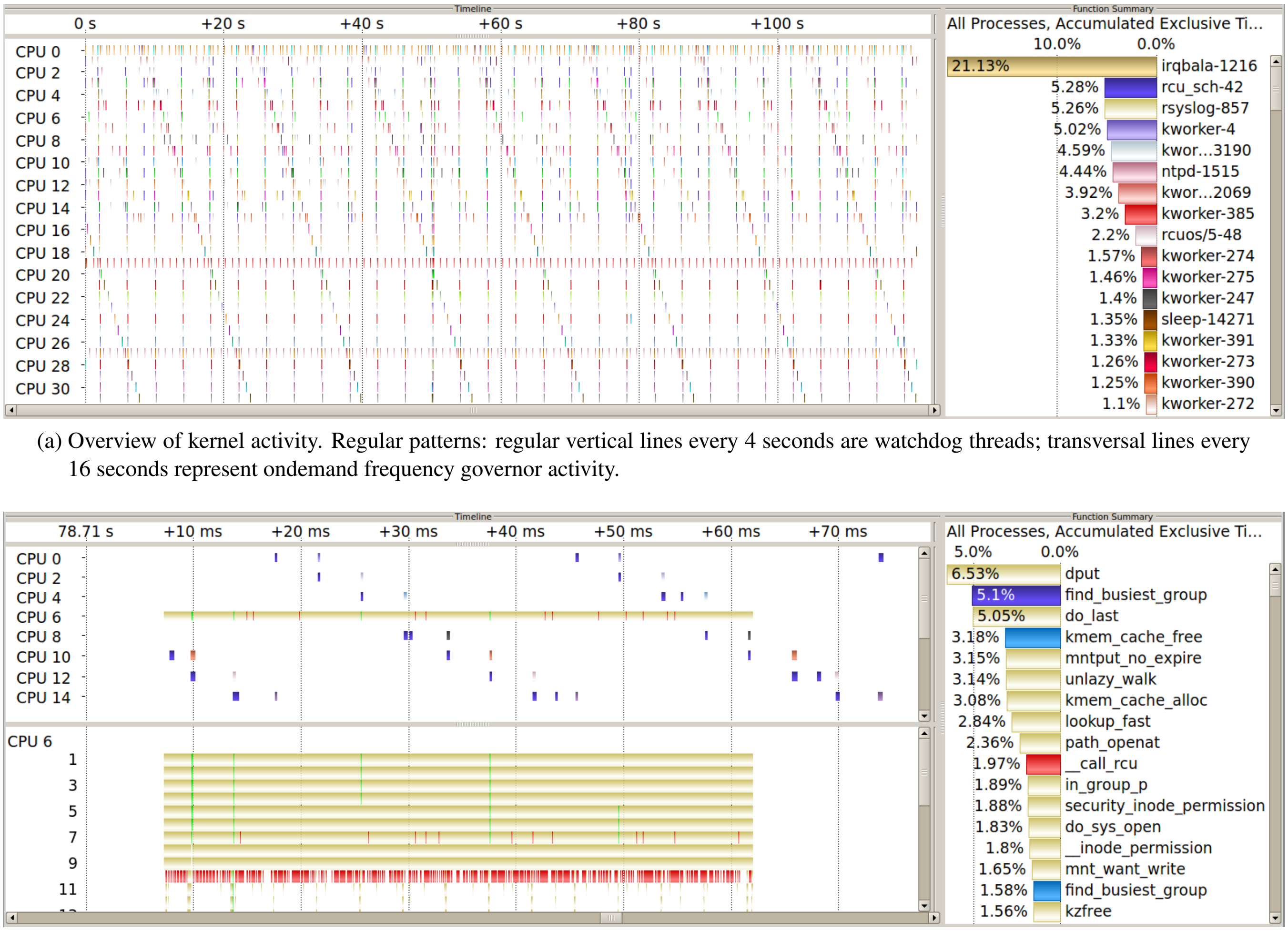
Figure 7a shows a trace of an idle dual socket system running Ubuntu Server 13.10. With eight cores per processor and HyperThreading active, 32 logical CPUs are available. We filtered out the idle functions that use up to 99.95 % of the total CPU time that is spent in the kernel. The largest remaining contributors are the irqbalancer, the RCU scheduler [21, 22], the rsyslog daemon, some kernel worker tasks, and the NTP daemon. We also see that there are two different kinds of per CPU threads that issue work periodically: watchdog threads and kernel worker threads that are used by the ondemand governor. Watchdog threads start their work every 4 s (displayed as vertical lines). The ondemand frequency governor is activated every 16 s on most CPUs (transversal lines). kworker-4 is the kernel worker thread of CPU 0. It uses significantly more time compared to other kernel workers since it is periodically activated by ksoftirq, which is running on CPU 0 and is handling IPMI messages at a regular interval of 1 s. CPU 22 also executes work every second triggered by the NTP daemon.

The RCU scheduler is mainly triggered by irqbalance and the rsyslog daemon. Zooming into the trace, we see that these tasks use the `__call_rcu` function. Shortly afterwards, the RCU scheduler starts and handles the grace periods of the RCU data. In this example, the RCU scheduler task runs on different processor cores but always on the same NUMA node as the process that issued RCU calls. Figure 7b depicts this behavior for the rsyslogd activity. After the RCU scheduler is migrated to another CPU, a kernel worker thread is scheduled. The kernel worker thread handles the ondemand frequency governor timer (od_dbs_timer, not depicted).

## 5 Conclusion and Future Work

This paper presents a new combined workflow for recording, managing, and visualizing performance data on Linux systems. We rely on established performance monitoring infrastructures and tools, making our approach applicable in a wide range of scenarios. Call-path sampling works on standard production systems and does not require root access or special permissions. Having additional permissions to record special tracepoint events can further increase the level of detail. By using already available Linux tools that require no recompilation or re-linking, the entry barrier for performance analysis has been lowered significantly. With sampling, the overhead can be controlled by selecting an appropriate event frequency. For a visual analysis, we leverage the Vampir visualization tool that originates from the HPC community. This enables a scalable and flexible visualization of trace data that contains information from a large number of processes, running over a long time, and including a high level of detail.

We have demonstrated the versatility of our approach with several use cases, including an analysis of scientific applications running on large production systems,



All Processes, Accumulated Exclusive Ti...

| 10.0% | 0.0% |
|---|---|
| 21.13% | irqbala-1216 |
| 5.28% | rcu_sch-42 |
| 5.26% | rsyslog-857 |
| 5.02% | kworker-4 |
| 4.59% | kwor...3190 |
| 4.44% | ntpd-1515 |
| 3.92% | kwor...2069 |
| 3.2% | kworker-385 |
| 2.2% | rcuos/5-48 |
| 1.57% | kworker-274 |
| 1.46% | kworker-275 |
| 1.4% | kworker-247 |
| 1.35% | sleep-14271 |
| 1.33% | kworker-391 |
| 1.26% | kworker-273 |
| 1.25% | kworker-390 |
| 1.1% | kworker-272 |

(a) Overview of kernel activity. Regular patterns: regular vertical lines every 4 seconds are watchdog threads; transversal lines every 16 seconds represent ondemand frequency governor activity.

All Processes, Accumulated Exclusive Ti...

| 5.0% | 0.0% |
|---|---|
| 6.53% | dput |
| 5.1% | find_busiest_group |
| 5.05% | do_last |
| 3.18% | kmem_cache_free |
| 3.15% | mntput_no_expire |
| 3.14% | unlazy_walk |
| 3.08% | kmem_cache_alloc |
| 2.84% | lookup_fast |
| 2.36% | path_openat |
| 1.97% | __call_rcu |
| 1.89% | in_group_p |
| 1.88% | security_inode_permission |
| 1.83% | do_sys_open |
| 1.8% | __inode_permission |
| 1.65% | mnt_want_write |
| 1.58% | find_busiest_group |
| 1.56% | kzfree |

(b) Zoomed into rsyslogd activity, which triggers RCU scheduler activity. rsyslogd calls to RCU objects are colored red. rsyslogd runs on CPU 6. The CPU location of the RCU scheduler changes over time across unoccupied cores of one NUMA package. The same NUMA package is used by rsyslogd and rcuos/6. The light blue activity (not depicted in timeline, but in function statistics) represents the rcuos/6 task that offloads RCU callbacks for CPU 6.

Figure 7: Kernel activity of an idle dual socket Intel Sandy Bridge node. Idle functions have been filtered out.

the activity on a highly utilized web and database server, as well as investigating operating system noise. Different performance aspects can be covered: Hardware performance counters, call-path samples, process and task management, library calls, and system calls provide a holistic view for post-mortem performance analysis, focusing either on the entire system or on a specific application. Given the pervasiveness of Linux, even more use cases are possible, for instance optimizing energy usage on mobile systems.

Our future work will focus on some remaining scalability issues. The recording process of perf record – where only one file is written – should be reconsidered as the number of CPUs will continue to increase. The conversion process to OTF should be re-implemented as it is currently single threaded. We provide kernel patches that add missing functionality to the existing tools rather than using the `perf_event_open` system call directly, as the latter would result in the re-implementation of several perf userspace tool features. Additionally, we submitted bug-fixes, one of which was accepted into the main-line kernel. Furthermore, we plan to integrate measurements on multiple nodes to generate a single sampling-based trace from a distributed application. This will allow us to study interactions between processes on different systems as in client-server scenarios and massively parallel applications. Moreover, we plan to switch to OTF2 as the successor of the OTF data format. OTF2 will include support for a sampling data type that will reduce the trace size and speed up the conversion process.

## 6 Acknowledgment

This work has been funded by the Bundesministerium für Bildung und Forschung via the research project CoolSilicon (BMBF 16N10186) and the Deutsche Forschungsgemeinschaft (DFG) via the Collaborative Research Center 912 “Highly Adaptive Energy-Efficient Computing” (HAEC, SFB 921/1 2011). The authors would like to thank Danny Rotscher, Thomas William, and all people who are involved in patching and developing perf and ftrace. You are great, keep up the good work!

## References

[1] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. HPCTOOLKIT: tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6), 2010.

[2] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks – summary and preliminary results. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, Supercomputing ’91, pages 158–165, New York, NY, USA, 1991. ACM.

[3] Jonathan Corbet. Fun with tracepoints. *LWN - Linux Weekly News - online*, August 2009.

[4] Mathieu Desnoyers. Common Trace Format (CTF) Specification (v1.8.2). *Common Trace Format GIT repository*, 2012.

[5] Advanced Micro Devices. *BIOS and Kernel Developer’s Guide (BKDG) for AMD Family 15h Models 00h-0Fh Processors, Rev 3.08*, March 12, 2012.

[6] J. Dongarra, K. London, S. Moore, P. Mucci, and D. Terpstra. Using PAPI for Hardware Performance Monitoring on Linux Systems. In *Conference on Linux Clusters: The HPC Revolution*, Urbana, Illinois, June 2001.

[7] Jake Edge. A look at ftrace. *LWN - Linux Weekly News - online*, March 2009.

[8] Jake Edge. perfcounters being included into the mainline during the recently completed 2.6.31 merge window. *LWN - Linux Weekly News - online*, July 2009.

[9] Dominic Eschweiler, Michael Wagner, Markus Geimer, Andreas Knüpfer, Wolfgang E. Nagel, and Felix Wolf. Open Trace Format 2: The Next Generation of Scalable Trace Formats and Support Libraries. In Koen De Bosschere, Erik H. D’Hollander, Gerhard R. Joubert, David A. Padua, Frans J. Peters, and Mark Sawyer, editors, *PARCO*, volume 22 of *Advances in Parallel Computing*, pages 481–490. IOS Press, 2011.

[10] Pierre-Marc Fournier, Mathieu Desnoyers, and Michel R. Dagenais. Combined Tracing of the Kernel and Applications with LTTng. In *Proceedings of the 2009 Linux Symposium*, July 2009.

[11] Francis Giraldeau, Julien Desfossez, David Goulet, Mathieu Desnoyers, and Michel R. Dagenais. Recovering system metrics from kernel trace. In *Linux Symposium 2011*, June 2011.

[12] Sudhanshu Goswami. An introduction to KProbes. *LWN - Linux Weekly News - online*, April 2005.

[13] Thomas Ilsche, Joseph Schuchart, Jason Cope, Dries Kimpe, Terry Jones, Andreas Knüpfer, Kamil Iskra, Robert Ross, Wolfgang E Nagel, and Stephen Poole. Optimizing I/O forwarding techniques for extreme-scale event tracing. *Cluster Computing*, pages 1–18, 2013.

[14] Intel. *Intel 64 and IA-32 Architectures Software Developer’s Manual Volume 3A, 3B, and 3C: System Programming Guide*, February 2014.

[15] James A. Keniston. Ptrace, Utrace, Uprobes: Lightweight, Dynamic Tracing of User Apps. In *Proceedings of the 2007 Linux Symposium*, June 2007.

[16] Namhyung Kim. perf tools: Introduce new ’ftrace’ command, patch. *LWN - Linux Weekly News - online*, April 2013.

[17] Andreas Knüpfer, Ronny Brendel, Holger Brunst, Hartmut Mix, and Wolfgang E. Nagel. Introducing the Open Trace Format (OTF). In Vassil N. Alxandrov, Geert D. Albada, Peter M. A.

Sloot, and Jack J. Dongarra, editors, *6th International Conference on Computational Science (ICCS)*, volume 2, pages 526–533, Reading, UK, 2006. Springer.

[18] Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen D. Malony, Wolfgang E. Nagel, Yury Oleynik, Peter Philippen, Pavel Saviankou, Dirk Schmidl, Sameer S. Shende, Ronny Tschüter, Michael Wagner, Bert Wesarg, and Felix Wolf. Score-P – A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir. In *Proc. of 5th Parallel Tools Workshop, 2011, Dresden, Germany*. Springer, September 2012.

[19] John Levon. OProfile Internals. *OProfile online documentation*, 2003.

[20] John Levon. OProfile manual. *OProfile online documentation*, 2004.

[21] Paul E. McKenney. The new visibility of RCU processing. *LWN - Linux Weekly News - online*, October 2012.

[22] Paul E. McKenney and Jonathan Walpole. What is RCU, Fundamentally?. *LWN - Linux Weekly News - online*, December 2007.

[23] Philip J. Mucci, Shirley Browne, Christine Deane, and George Ho. PAPI: A Portable Interface to Hardware Performance Counters. In *In Proceedings of the Department of Defense HPCMP Users Group Conference*, pages 7–10, 1999.

[24] Matthias S. Müller, Andreas Knüpfer, Matthias Jurenz, Matthias Lieber, Holger Brunst, Hartmut Mix, and Wolfgang E. Nagel. Developing Scalable Applications with Vampir, VampirServer and VampirTrace. In *Parallel Computing: Architectures, Algorithms and Applications*, volume 15. IOS Press, 2008.

[25] Steven Rostedt. trace-cmd: A front-end for Ftrace. *LWN - Linux Weekly News - online*, October 2010.

[26] Steven Rostedt. Using KernelShark to analyze the real-time scheduler. *LWN - Linux Weekly News - online*, February 2011.

[27] Sameer S. Shende and Allen D. Malony. The TAU Parallel Performance System. *Int. J. High Perform. Comput. Appl.*, 20(2), 2006.

[28] Mark Wielaard. A SystemTap update. *LWN - Linux Weekly News - online*, January 2009.