lo2s – Multi-Core System and Application Performance Analysis for Linux

Thomas Ilsche, Robert Schöne, Mario Bielert, Andreas Gocht, Daniel Hackenberg
Center for Information Services and High Performance Computing (ZIH)
Technische Universität Dresden, 01062 Dresden, Germany
{firstname.lastname}@tu-dresden.de

***Abstract*—In this paper we present lo2s – a lightweight performance monitoring tool to sample applications as well as the executing system. It enables the user to analyze the performance of a parallel application without requiring the time-consuming and error-prone process of application instrumentation. The collected performance data is complemented with various metric data, i.e., perf counters, kernel tracepoints, model specific registers, and custom metric data provided by plugins. Comprehensive visualization is enabled by compatibility with established tools.**

I. Introduction

Performance analysis is the foundation for any optimization of a sufficiently complex code. Since performance depends on both the application and the executing system, a performance analysis tool should ideally cover both aspects. Further, the usability of a performance analysis tool depends heavily on the effort required to apply it to an application. lo2s uses a nonintrusive monitoring approach – there is no need to modify or recompile the user application. The monitored application is simply executed with a prefix-command. lo2s is independent of the application's parallel paradigm or programming language. Furthermore, lo2s allows users to analyze performance anomalies in relation to the hardware or operating system. Traces are stored in the Open Trace Format 2 (OTF2) that can be used in an offline analysis by existing performance analysis tools. In particular, we use the established scalable trace visualizer Vampir [8] in conjunction with lo2s.

In Section II, we give a overview of how lo2s fits the landscape of performance monitoring tools. The current implementation state regarding the process and system monitoring features of lo2s is described in Section III and Section IV respectively. Section V gives an overview about additional metrics that can be included in the analysis. Section VI describes the capabilities of the tool on the basis of two use cases. We conclude with a summary and description of future developments in Section VII.

II. Related Work

Performance monitoring tools can be classified based on how they implement the three stages of performance analysis: data acquisition, data recording, and data presentation [6].

HPCToolkit [4] is a performance analysis tool suite that focuses on sampling of parallel applications, supporting MPI and OpenMP. It uses summarization to provide performance profiles. Moreover, a timeline view based on logging is available. HPCToolkit includes information from call stack samples which are processed in userspace with sophisticated unwinding techniques and PAPI performance counters.

Linux itself includes perf [3] – a set of tools that utilize the perf_event_open [1] infrastructure. These powerful tools cover a wide range of performance related information that is available to the operating system. Available features include performance counter based sampling, certain instrumentation points, logging to record the collected information, and visualization as textual profiles. However, perf uses a monolithic file-format and lacks a scalable and user-friendly way to analyze and visualize the resulting timelines. Previously, we have used perf for performance monitoring and converted the resulting trace into a different file format that allowed the usage of timeline visualization tools [11]. OProfile [2] is an non-intrusive statistical profiler built on the perf_event_open infrastructure. It supports low-overhead system-wide and single process monitoring, but is limited to aggregated profiles only.

With lo2s, we use the underlying perf_event_open infrastructure and write traces directly as OTF2. This avoids the additional conversion step and allows us to leverage additional information sources that are not accessible to the perf tools. While perf record uses one monitoring thread and writes a monolithic trace, lo2s uses separate monitoring threads for each application thread or logical CPU that is being observed. OTF2 being a parallel trace format consists of separate files that are written independently by each monitoring thread without any synchronization. Therefore our approach is scalable within one node, as no additional synchronization overhead is introduced.

III. Process Monitoring

Usually, lo2s monitors a specific process group. It can either start a specific program acting as a prefix-command, or attach to an already running process. All forked processes or child threads inherit the monitoring, which is achieved by using ptrace on all monitored tasks. This allows lo2s to be independent of the specific parallelization paradigm and runtime. As soon as a new thread (*tracee*) is forked, lo2s registers a monitoring thread (*tracer*) that collects metrics and call stack samples for the tracee.

lo2s supports instruction based sampling via the Linux perf_event_open interface [1]. The sampling is triggered via a configurable per-thread instruction counter overflow. The default interval of 11010113 instructions is chosen as a trade-off between overhead and granularity. We chose a prime number to avoid aliasing effects on repetitive instruction execution in tight loops. Each sample includes the current instruction pointer and optionally a call stack. The call stack is only available if frame pointers are not omitted. The sampling itself is setup independently for each tracee with individual memory buffers. By collecting the call stack samples in the kernel, a context switch into userspace monitoring code is avoided. This leads to a reduced per-sample overhead, a distinct advantage of using the `perf_event_open` infrastructure.

In regular intervals, the buffer used by the kernel is converted to OTF2 events. Additionally, `lo2s` records per-thread metrics during the buffer flushes. By default it collects a number of metrics reflecting the activity on different levels in the cache/memory hierarchy. During monitoring, there is no explicit synchronization among the monitoring threads, but the call-stack processing is synchronized using a timer. The simultaneous perturbation of the tracee threads reduces the impact of measurement noise on tightly coupled parallel applications.

During the execution, only the current instruction or call stack is recorded as a OTF2 `CallingContextSample`. Resolving the identifier of this `CallingContextSample` based on the call stack of instruction pointers is done by traversing a local tree of instruction pointers. At the end of the instrumented run, the local trees are merged and the local identifiers are mapped to global ids. Further, for each instruction pointer, the corresponding symbol is resolved and the corresponding instruction is disassembled. To do so, the corresponding binary file and offset from each instruction pointer needs to be determined. This is done by recording `mmap` events with `perf_event_open` and combining that information with `/proc/$pid/maps`. Unfortunately, either one alone is not sufficient. Within each binary object file, `lo2s` uses `libbfd` to resolve the symbols and `libradare` to disassemble instructions.

Generally `lo2s` is agnostic to the programming language. However, for applications that run within a virtual machine like Python or Java, the mapping of instructions to symbols will be made to the virtual machine rather than the user program.

Process monitoring can be used without special permissions. In order to have kernel sampling events in the trace, `perf_event_paranoid` should be at most 1[1]. To attach to a running process, `ptrace_scope` should be disabled[2].

IV. System Monitoring

The second mode of operation of `lo2s` is a node-level system monitoring. Using this mode, `lo2s` records when which task was scheduled on a per-core basis. This information is retrieved from the `sched/sched_switch` tracepoint event. This event reveals whenever the kernel scheduler switches between two tasks, from idle or to idle. Similarly to the process monitoring, each event is written into a buffer. There is a dedicated thread for each logical CPU, which is also pinned to this CPU.

The event buffer is only read and converted to OTF2, when its occupation reaches 80 % or the monitoring is completed. Otherwise each CPU monitoring thread idles indefinitely within a call to `poll`, allowing an unperturbed measurement. An additional tracepoint, `sched/sched_process_exit`, is used to assign a command string to each process id[3].

System monitoring with `lo2s` requires administrative permissions or a `perf_event_paranoid` of -1 as well as user read access to `/sys/kernel/debug/tracing/`.

V. Additional Metrics

In both monitoring modes, `lo2s` can enhance the trace with additional metrics, i.e. tracepoint events, `x86_adapt` knobs, and Score-P plugin metrics. The process monitoring mode further supports per-thread perf metrics (cf. Section III).

a) Tracepoints: Besides the tracepoints used to gather task scheduling information, `lo2s` can take advantage of the large amount of other tracepoints that are already instrumented in the Linux kernel. For any given tracepoint event, `lo2s` will record all fields with a numerical value as a metric in the trace. As an example, `power/cpu_idle` will include information about the selected idle state by the Linux idle governor. Tracepoints provide valuable insight into the interactions between applications and the operating system.

b) x86_adapt: Generic hardware specific information can be integrated through `x86_adapt` [10]. The x86_adapt library and kernel module provides access to model specific registers (MSRs) that give in-depth insight into hardware specific performance information. This is particularly useful for new systems that run kernel versions that do not (yet) implement a specific interface to newly introduced hardware performance information.

c) Plugins: Finally, `lo2s` provides a plugin interface that is compatible with metric plugins [12] written for the Score-P [9] measurement infrastructure. This leverages a pool of existing plugins such as asynchronous PAPI recording or CPU energy counters[4]. Currently, the support is limited to plugins, which can be asynchronously recorded per host.

VI. Use Cases

In this section, we describe typical use cases for `lo2s` on a dual socket Intel Xeon E5-2690 v3 system running Ubuntu 16.04 Server.

A. Instruction based energy modeling

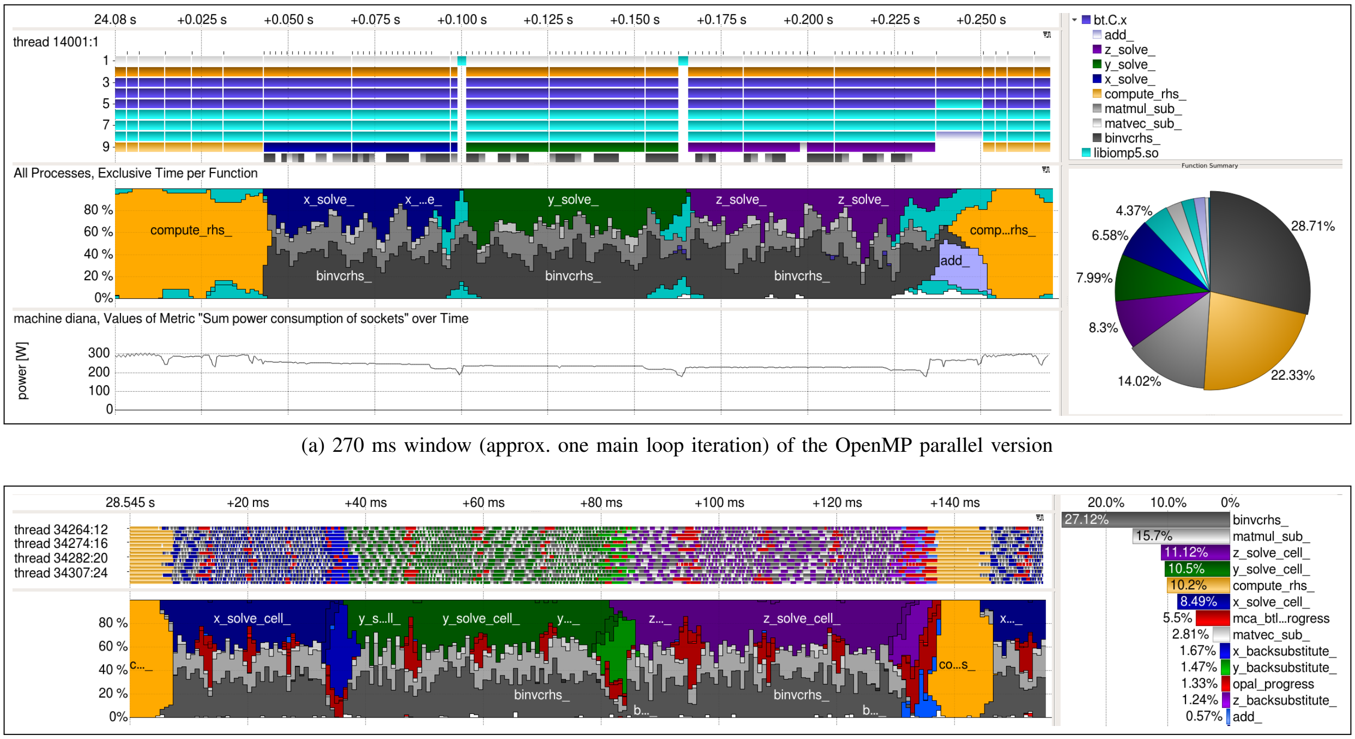
Initially, `lo2s` had been designed to support instruction based energy models. This is done by providing `perf` counters for accesses to the different memory levels, statistics about processor usage, and the assembly of the executed instruction at each sample. The energy model assigns costs to each memory

[1] `sudo sysctl kernel.perf_event_paranoid=1`

[2] `sudo sysctl kernel.yama.ptrace_scope=0`

[3] The command string associated with a process can change over time through calls to `exec`. The last one upon exit is usually the most meaningful.

[4] https://github.com/score-p



(a) 270 ms window (approx. one main loop iteration) of the OpenMP parallel version

(b) 160 ms window (approx. one main loop iteration) of the MPI parallel version

Figure 1: Vampir performance analysis of a BT benchmark on a dual-socket Intel Xeon E5-2690 v3 system.

access and instruction type as well as keeping the processor active. Based on the recorded trace, the model can estimate the energy cost for phases of application execution.

B. Application performance analysis

As a traditional HPC use-case we monitor the NAS parallel benchmark BT, parallelized with OpenMP [5]. Figure 1a shows a Vampir visualization of the resulting trace. The upper left represents the stack of the main thread over the time of one iteration, which consists of five parallel regions. The three `solve` functions exhibit a higher IPC rate, which leads to more frequent samples, as shown by the ticks on the top of the stack view. Supportive functions with a short runtime are colored gray. Under the stack, we show a time-series profile, which visualizes the estimated runtime share of each function for all threads based on the sampling hits. At the bottom left, we show the power consumption of the two sockets, which is included through a metric plugin. The increasing runtime share of the OpenMP library at the end of each phase, indicates an workload imbalance among the threads. This imbalance leads to a reduced power consumption whenever the threads wait for synchronization. The bottom-right chart show an exclusive function profile for the displayed time span.

Figure 1b shows the same application using MPI parallelization. The top timeline is colored by the sampled function for each of the 16 processes above the time-series profile of one iteration. The structure of the iteration is similar to the OpenMP version. However, the communication is explicit using OpenMPI function that are colored red. The timeline reveals three MPI synchronization phases within each of the `solve` functions and another MPI synchronization after each of these phases.

C. Combined system and process monitoring

Figure 2 shows the parallel build process of a C++ application as observed with both `lo2s` operation modes. The system monitoring is used to track the scheduling of processes on the CPUs of the machine. This includes the processes spawned by `make`, e.g., `cc1plus`, `ar`, and `ld`, but also `lo2s` itself. The process monitoring traces `make` and its child processes, which provides information about the lifetime of tasks involved in the build process. In the first ten seconds, the timeline shows a good saturation of processing resources with compilation tasks. After that the occupation of the CPUs goes down, as the dependencies between compilation units prevent a further parallel build.

D. Idle sleep state optimization

In [7], we describe a weakness in the Linux kernel idle governor, which leads to an insufficient use of idle sleep states and thus increased power consumption, so called *Powernightmare*s. For the analysis of this effect, we used the system monitoring of `lo2s` with different kernel tracepoints, some of which we have added to the used build of the Linux kernel. The recorded tracepoints provided information about the idle governor, in particular about the internal state, intermediate heuristical decisions, and the chosen C-State. We traced the system during idle and with a synthetic workload, which reproducibly trigger Powernightmares. With this setup, we were able to find the cause of the problem, propose a solution, and demonstrate its effectiveness.

VII. Conclusion and Future Work

We presented a novel lightweight performance analysis tool that gives detailed insight into both application and system performance. We demonstrated the different features with a traditional HPC benchmark and a parallel compilation workflow. The monitoring tool is available as open source and we encourage comments and contributions at https://github.com/tud-zih-energy/lo2s.

The versatility of Linux monitoring presents many opportunities for enhancements. As a next step, we will leverage the full range of available `perf` metrics as configurable alternative to the current predefined set of counters. Further, we want to improve the possibilities of simultaneously collecting system and process monitoring information. Performing sampling on CPUs rather than threads would also improve the system monitoring, but contradicting information from instruction sampling and scheduling events due to race conditions and timer inaccuracies have to be handled gracefully. We hope to establish `lo2s` as a useful tool for comprehensive node-level performance analysis and with the future possibility of merging traces from multiple nodes even beyond that.

Acknowledgments

This work is supported in part by the German Research Foundation (DFG) within the CRC 912 - HAEC and by the European Union's Horizon 2020 program in the READEX project (grant agreement number 671657).

References

[1] *Linux Programmer's Manual*, 2016 (accessed July 7, 2017). http://man7.org/linux/man-pages/man2/perf_event_open.2.html.

[2] *OProfile*, (accessed August 2, 2017). http://oprofile.sourceforge.net/about/.

[3] *perf: Linux profiling with performance counters*, (accessed July 7, 2017). https://perf.wiki.kernel.org/.

[4] Adhianto, L., et al. HPCTOOLKIT: tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010. DOI: 10.1002/cpe.1553.

[5] Bailey, D. H., et al. The NAS parallel benchmarks. Technical report, RNR, 1994.

[6] Ilsche, T., et al. Combining instrumentation and sampling for trace-based application performance analysis. In *Tools for High Performance Computing 2014*. 2015. DOI: 10.1007/978-3-319-16012-2_6.

[7] Ilsche, T., et al. Powernightmares: The challenge of efficiently using sleep states on multi-core systems. In *5th Workshop on Runtime and Operating Systems for the Many-core Era*. 2017, accepted for publication.

[8] Knüpfer, A., et al. The Vampir performance analysis tool-set. *Tools for High Performance Computing*, pages 139–155, 2008.

[9] Knüpfer, A., et al. Score-p: A joint performance measurement run-time infrastructure for Periscope, Scalasca, TAU, and Vampir. In *Tools for High Performance Computing*. 2012. DOI: 10.1007/978-3-642-31476-6_7.

[10] Schöne, R. et al. Integrating performance analysis and energy efficiency optimizations in a unified environment. *Computer Science - Research and Development*, pages 1–9, 2013. ISSN 1865-2034. DOI: 10.1007/s00450-013-0243-7.

[11] Schöne, R., et al. Scalable tools for non-intrusive performance debugging of parallel linux workloads. In *Proceedings of the Ottawa Linux Symposium*. 2014.

[12] Schöne, R., et al. Extending the functionality of score-p through plugins: Interfaces and use cases. In *Tools for High Performance Computing 201*. 2017. DOI: 10.1007/978-3-319-56702-0_4.

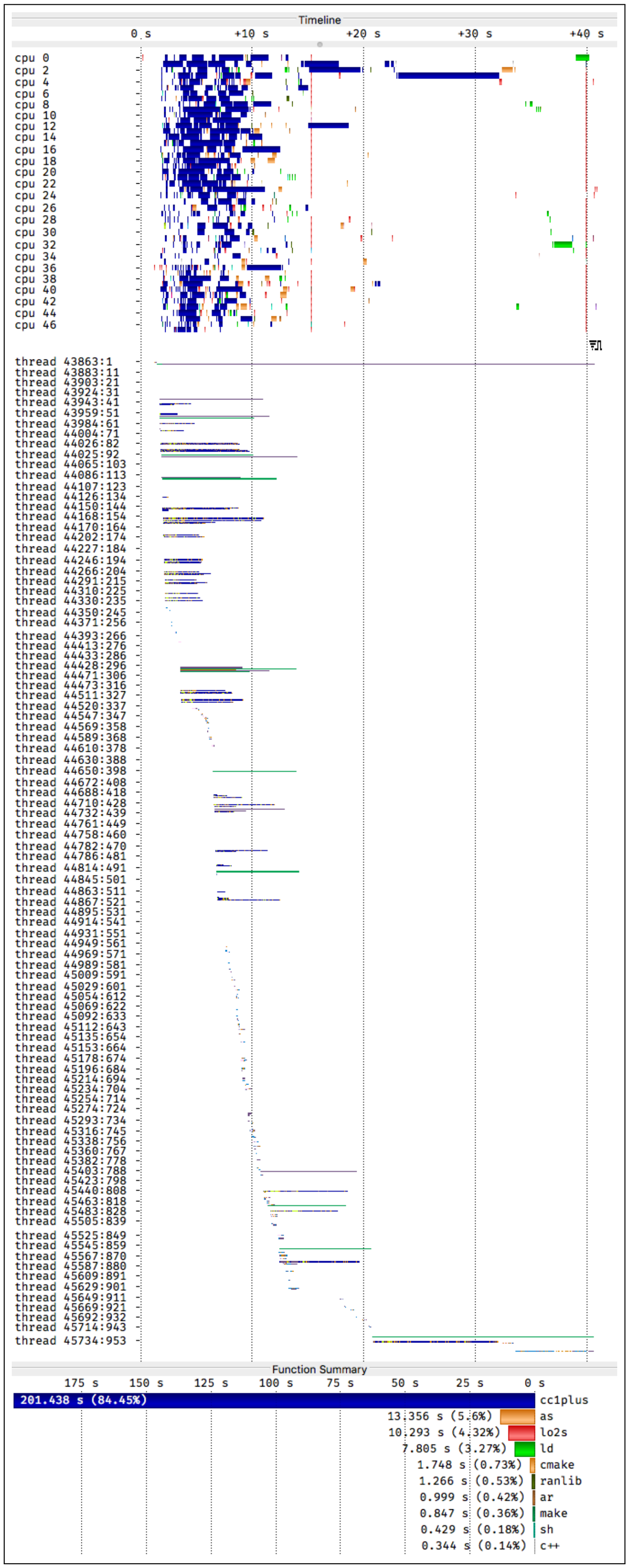


Figure 2: Combined process and system monitoring of a parallel build using `make -j`. The top section shows the scheduled processes. The lifetime of processes and threads is shown in the second part. The bottom part denotes the cpu time of the involved processes.