System Monitoring with `lo2s`: Power and Runtime Impact of C-State Transitions

Thomas Ilsche, Robert Schöne, Philipp Joram, Mario Bielert, Andreas Gocht
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 improvements to the low-overhead performance monitoring tool `lo2s`. We combine a detailed recording of system events with information from a high-resolution power measurement, to record the scheduling of applications and C-state transitions. These sub-millisecond transitions are difficult to observe with traditional approaches. Our methodology provides us with insights into the behavior of the system before, during, and after C-state transitions of processors cores.**

I. Introduction

Power and energy optimizations in the software stack are a widely discussed topic in the HPC community. Most of the improvements are driven by processor features like dynamic voltage and frequency scaling, which influence the power characteristic of an HPC node. Some processor features are accessible through kernel interfaces, but on other mechanisms, like C-states, the user has no direct influence. Still, they interact with the power characteristic of processors.

With the increasing complexity of energy efficiency tuning, it becomes challenging to isolate the impact of a specific optimization and to understand their interactions. However, this insight is required to achieve optimal energy efficiency and to implement new power-saving features. Moreover, these optimizations affect the entire software and hardware stack – including application, operating system, architecture specification, and micro architecture implementation.

The low-overhead performance monitoring tool `lo2s` exposes various sources of monitoring data from all different layers. This reveals the operation of opaque mechanisms like C-states and their impact on software performance and hardware power consumption. To achieve this, `lo2s` uses kernel tracepoints to record decisions made by the operating system. Regular sampling of the instruction pointer and hardware event counters give a dynamic picture of the application execution. Moreover, `lo2s` supports a plugin interface, which enables users to record the power consumption or performance counters.

The goal is to give a holistic view on a system and expose the complex interactions between application, operating system, and hardware. By using features of modern Linux kernels to perform the majority of the monitoring tasks, the perturbation on the observed system is reduced to a minimum. Additional fine-grained power measurements are recorded on a separate system, and only integrated post-mortem with no impact on the system under test.

This paper is structured as follows: Section II provides an overview of related monitoring tools as well as C-state analyses. We describe recent changes to `lo2s` in Section III. In Section IV, we present an overhead analysis for `lo2s` and show how C-state behavior can be analyzed with `lo2s` and Vampir. Section V concludes this paper with a summary and an outlook.

II. Background and Related Work

Performance monitoring tools for High Performance Computing often focus on applications. For example, Score-P [10] uses compiler instrumentation, library interposition, and other techniques to collect events from parallel applications. HPC-Toolkit [4] primarily uses sampling, which provides a less intrusive way to record application behavior. Both tools support various parallelization paradigms for monitoring applications that run on multiple compute nodes. On a single node, `perf` [2] also provides versatile ways to monitor applications and, in addition, the system as a whole. However, its scalability is limited, particularly regarding the visualization and analysis of extensive trace data.

ACPI C-states [3, Section 8.1] describe a hardware mechanism that is used by operating systems to reduce the energy consumption of processors. Based on hints from the operating system, the processor can use voltage and frequency scaling, clock gating, and power gating to lower the power consumption of the processor. However, switching between different states introduces a latency that can have performance implications. Using C-states effectively is essential for the energy-efficient operation of modern systems [9].

In [12], we presented a plugin for the VampirTrace performance monitoring tool, in which kernel tracepoints were used to record the C-state behavior of CPUs during the execution of an application. We also included power consumption information in the collected time lines. This enabled us a to verify the usage of C-states and to describe the influence of workload, C-state configuration, and frequency configuration to the system power consumption. However, a fine-grained analysis of the transitions themselves was not feasible. One of the reasons is that the source of a C-state transition was not recorded. Furthermore, the temporal resolution of the power measurement infrastructure was too coarse-grained.

In [5], Barrachina et al. implemented a comparable approach for Extrae/Paraver. However, they *sampled* C-state usage instead of using kernel tracepoints. While this can reduce overhead, the direct information when a C-state has changed is now hidden and only statistical information is available. Such an analysis is also possible with Score-P [14]. In [13], we analyzed C-states by instrumenting the Linux kernel. The instrumentation enabled users to measure how long it takes a CPU to wake up from different C-states.

In this paper, we address the shortcomings of previous work by introducing a measurement environment that collects the related events using a standard Linux kernel. Further, we enhance the monitoring with a high resolution power measurement.

III. Performance and Power Measurement with `lo2s`

`lo2s`[1] is a lightweight monitoring tool for Linux [8]. It uses the Linux `perf_event_open` infrastructure which provides versatile ways to get system and process performance data. The collected data is recorded with timestamps and stored in the Open Trace Format 2 [6]. This leverages existing performance analysis tools, in particular visualization with Vampir [11]. `lo2s` can operate in two modes focusing either on a particular process and it's children or on the system as a whole.

A. Reducing Measurement Overhead for Metric Recording

When measuring with `lo2s`, one key goal is to keep its impact on the measured system or process to a minimum. We improved `lo2s` to require less user-space monitoring code execution and group metrics. Using an in-kernel based approach, we were able to reduce our overhead when recording multiple metrics at once.

The `perf_event_open` system call provides a mechanism to collect a set of related events into a group that will be scheduled onto the CPU as a unit [1]. This allows a combined read of all values and prevents the events to be multiplexed. By default, the kernel will put each ungrouped event into its own group and multiplex the groups if not all hardware events can be recorded simultaneously. While multiplexing allows recording more metrics than supported by hardware, it will reduce the accuracy of the measurement data, particularly for short measurement intervals, because the gaps have to be interpolated. Multiplexing also uses a regular timer to switch the scheduled events.

In addition to a collective metric readout, this enables the recording of metrics entirely in the kernel. Typically, the group leader is an event with an independent constant rate, e.g. `ref_cycles` that is set to overflow at a given rate. Whenever this group leader event generates an overflow notification, the counters for all events in its group are recorded in a buffer by the kernel. The userspace monitoring code only needs to run whenever the buffer reaches a watermark, which is indicated by a poll event on a file descriptor. Ideally, the buffer is configured such that it provides sufficient space for all measurement samples that are recorded during the application execution. That way, the overhead of reading the buffer and writing the events to the final trace format, only occurs at the end of the monitoring. In Section IV-B, we evaluate the implications of this optimization on the measurement perturbation.

[1]https://github.com/tud-zih-energy/lo2s

B. Integrating Energy Measurements

For the established Score-P [10] performance measurement infrastructure, we have developed metric plugins [14] for three different energy measurements presented in [7], i.e., processor integrated Intel RAPL, ZES Zimmer LMG 450 AC measurement, and a custom DC measurement based on shunts read by National Instrument DAQ cards. For a seamless integration in our infrastructure, we added support for a subset of Score-P metric plugins in `lo2s`. Due to its focus on low system impact, `lo2s` can only support asynchronous system-wide metric plugins, which do not necessarily require regular user-space metric measurement code execution. `lo2s` can natively utilize such Score-P plugins without any changes. The metric measurement starts during the initialization, and each used plugin provides the recorded metric data during the finalization phase of `lo2s`. All collected metric data is written to the resulting trace and can be analyzed afterwards.

IV. Measurements and Evaluation

Based on the previous description, in this section we evaluate the effectiveness of the optimizations and use `lo2s` to analyze C-state transitions.

A. System Description

For our experiments, we use a dual-socket workstation with two 12-core Intel Xeon E5-2690 v3 (Haswell EP) processors and 256 GiB memory. During all experiments, we fixed the frequency of all CPUs to their nominal frequency of 2.6 GHz. The system runs on Ubuntu 16.04 and Linux kernel version 4.13.0 with activated page table isolation.

We rely on a high temporal resolution of the power measurements to accurately evaluate C-state transitions that last only in the order of microseconds. Therefore, we used our custom DC device, which provides a resolution of 2 µs. A separate system records the energy measurement and a metric plugin receives the data from that system during the finalization.

B. Overhead Evaluation

In order to evaluate, whether the grouped in-kernel metric measurements decreased perturbation, we compare `lo2s` kernel-space measurements to the previous implementation in user-space and `perf stat`. Like `lo2s`, the Linux tool `perf stat` also provides a way to record metric data in intervals of at least 10 ms. For `perf stat`, we use two different configurations, i.e., all events in the same group, and each event within its own group.

All four configurations record metrics at 10 ms intervals. To that end, the new implementation uses the *ref-cycles* event as a group leader with an overflow count of 26 000 000, which equals 10 ms at a nominal frequency of 2.6 GHz.

As a workload, we use a simple micro-benchmark that increments a counter for 100 s in a single thread. Based on

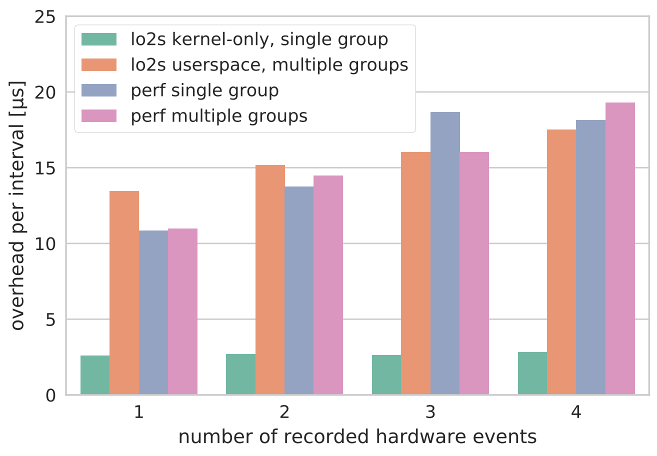


Figure 2: Overhead of metric readouts for different `lo2s` implementations and `perf` configurations with increasing number of observed hardware events

```
static pthread_cond_t cv; pthread_mutex_t lock;
/*                  callee                  */
void *callee(void *v) {
// ... initialize variables, set affinity
 for (int i = 0 ; i < number_measurements ; ++i) {
  // wait for wake up signal loop
  pthread_cond_wait(&cv, &lock);
  // ... busy wait for 200 microseconds
 }
}
/*              main/caller              */
int main(int argc, char ** argv) {
 // ... initialize variables, set affinity
 pthread_create(t, NULL, callee, NULL);
 for (int i = 0 ; i < number_measurements ; ++i) {
  // ... busy wait for 1 second
  // send wakeup signal
  pthread_cond_signal(&cv);
 }
}
```

Listing 1: Setup for triggering a C-state wakeup

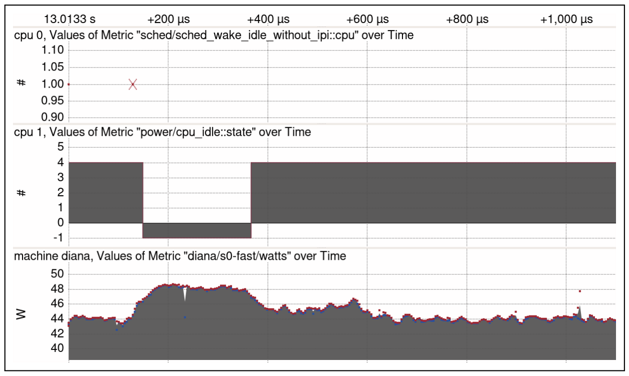
a reference execution with no measurements, we determine the effective computation time loss introduced by the measurements.

Figure 2 shows that the new implementation has a significantly smaller overhead per interval, i.e., ∼2.5 µs compared to >10 µs for the previous `lo2s` implementation and `perf stat`. Furthermore, the overhead does not increase notably with additional recorded metrics. These numbers depend on a variety of factors, such as workload, frequency, amount of threads involved, and sampling rate. To account for this, we kept the environment consistent for the different tools and implementations.

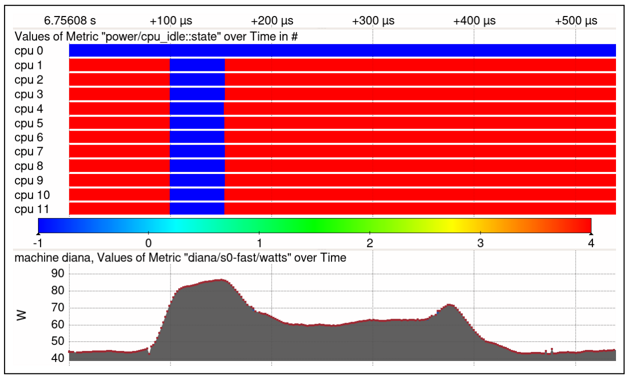
C. Analysis of C-State Transitions on Haswell-EP

The combination of low-overhead with well-integrated measurements of kernel tracepoints, user-space application sampling, and high-resolution energy measurements, allows us to use `lo2s` for investigating hardware and operating system behavior. We show this by transferring and extending previous work on C-state latencies [13]. In contrast to our previous work, we now use an unmodified kernel and an external power measurement infrastructure to record the hardware behavior during the transitions. To trigger a C-state change, we implemented a simple C program, shown in Listing 1. This program sets up two threads, `caller` and `callee`, and pins each of them to a given CPU. The remaining CPUs are kept in an idle state. In an interval of 1 s, the caller thread sends a signal using `pthread_cond_signal` to the callee. This invokes a call to the Linux kernel, which triggers the callee's CPU to wake up and writes a *sched_wake_idle_without_ipi* event almost immediately afterwards. The callee's CPU returns from its idle state, writes a *power_idle* event, and continues executing the callee thread. To verify that the system uses the assumed idle state, we use this *power_idle* event. However, after the callee thread returns from its waiting state, it busy-waits for 200 µs, before going back to idle using `pthread_cond_wait`.

Figure 3a visualizes a single wake-up event of the described workload with Vampir. The upper metric display shows that CPU 0 triggers waking up CPU 1. After 17 µs, CPU 1 switches to an active mode (-1) as shown in the middle metric display. It stays active for 220 µs, which includes the 200 µs busy-wait and the overhead for locking the mutex and switches from and to kernel-space. The active period can also be seen in the power consumption of the socket, which increases from 44 W to 48 W, as visualized in the bottom metric display. However, even though CPU 1 is idling again after 220 µs, the power consumption is still increased to 45–46 W for another 230–240 µs. The same behavior can also be seen when multiple CPUs are activated and deactivated simultaneously, as shown in Figure 3b. The reason for this behavior is not documented. However, within this time period, the core could possibly flush its state. This option is unreasonable since only a small amount of data is accessed within each wakeup of the callee, and even less data is modified. Alternatively, the processor core could apply power gating only after a certain grace period. At the end of this period, the state is flushed, which could cover for the power consumption increase shortly before the idle state is applied. There are two supportive observations for this thesis. First, the power consumption during the grace period, after all cores were active, is similar to the power consumption when all cores are in C1. Second, if the caller triggers another wakeup within this grace period, the latency is significantly lower (∼1 µs). Furthermore, in such a scenario, the time the callee resides in an active state is decreased to ∼205 µs and therefore significantly closer to the expected 200 µs busy wait time. This can be attributed to caches, which are still warm since the core has not been power gated. Another interesting finding is that the second wake-up in such a scenario has almost no initializing peek power. This can also be attributed to the fact that used data still resides in the caches and does not have to be transferred.



(a) CPU 0 triggering a wakeup on CPU 1. In the upper metric display, the cross depicts the event that triggers CPU 1 to wake up. The second metric display shows for how long CPU 1 remained active (≈220 µs). The power consumption is increased for ≈460 µs.



(b) Aligned wake-up of all CPUs. In the upper metric display, the C-state of all CPUs of the first socket is visualized (blue: active, red: C6). After all CPUs switch to idle, the power consumption remains high for ≈230 µs

Figure 3: C-state observation illustrated with Vampir. The bottom display shows the power consumption of the first socket.

V. Conclusion and Future Work

In this paper, we presented new functionality and advanced features, which we recently introduced in the monitoring tool `lo2s`. Now, `lo2s` can record metrics with significantly lower overhead, while still being NUMA-aware and scalable, distinguishing it from alternative tools. With the new functionality, it is possible to investigate hardware and system behavior at a fine-grained level. Furthermore, the support for asynchronous Score-P metric plugins enables users to perform advanced analyses that include, for example, power measurements. In our paper, we used these features to implement a new methodology for measuring performance and power characteristics of C-state changes. The analysis results are consistent with the previous observations that used a specially patched kernel. A new finding is that the power consumption is increased for more than 200 µs after entering a deep C-state. We explained a possible reason for this behavior, but the underlying mechanism is unknown and needs further investigation. By combining the low-overhead system monitoring with high resolution energy measurements, we expose the dynamic power characteristics of C-state transitions on a sub-millisecond level. The newly gained insight provides the basis for further optimizations, particularly regarding the delicate selection of C-states that are have to balance the trade-off between energy consumption and performance.

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] *perf_event_open(2) – Linux Programmer's Manual*, 4.14 edition, September 2017.

[2] *perf: Linux profiling with performance counters*, (accessed Februrary 2, 2018). https://perf.wiki.kernel.org/.

[3] Advanced configuration and power interface (acpi) specification, revision 6.1, January 2016. online at uefi.org (accessed 2017-01-30).

[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] Barrachina, S., et al. An integrated framework for power-performance analysis of parallel scientific workloads. *Int. Conf. on Smart Grids, Green Communications and IT Energy-aware Technologies*, pages 114–119, 2013.

[6] Eschweiler, D., et al. Open trace format 2: The next generation of scalable trace formats and support libraries. In *Applications, Tools and Techniques on the Road to Exascale Computing*, volume 22 of *Advances in Parallel Computing*, pages 481 – 490. 2012. ISBN 978-1-61499-040-6. doi:10.3233/978-1-61499-041-3-481.

[7] Ilsche, T., et al. Power measurements for compute nodes: Improving sampling rates, granularity and accuracy. In *2015 Sixth International Green Computing Conference and Sustainable Computing Conference (IGSC)*. December 2015. doi:10.1109/IGCC.2015.7393710.

[8] Ilsche, T., et al. lo2s—multi-core system and application performance analysis for Linux. In *Cluster Computing (CLUSTER), 2017 IEEE International Conference on*, pages 801–804. IEEE, 2017. doi:10.1109/CLUSTER.2017.116.

[9] Ilsche, T., et al. Powernightmares: The challenge of efficiently using sleep states on multi-core systems. In *Euro-Par 2017: Parallel Processing Workshops*. 2017.

[10] 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.

[11] Nagel, W. E., et al. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer 63*, XII(1):69–80, 1996.

[12] Schöne, R., et al. The VampirTrace Plugin Counter Interface: Introduction and Examples. In *Euro-Par 2010 Parallel Processing Workshops*, volume 6586 of *Lecture Notes in Computer Science (LNCS)*. Springer-Verlag, 2011. ISBN 978-3-642-24877-4. ISSN 0302-9743. doi: 10.1007/978-3-642-21878-1_62. DOI: 10.1007/978-3-642-21878-1_62.

[13] Schöne, R., et al. Wake-up Latencies for Processor Idle States on Current x86 Processors. *Computer Science - Research and Development*, 2014. ISSN 1865-2034. doi:10.1007/s00450-014-0270-z. DOI: 10.1007/s00450-014-0270-z.

[14] Schöne, R., et al. Extending the functionality of score-p through plugins: Interfaces and use cases. In *Tools for High Performance Computing 2016*, pages 59–82. Springer, 2017.