Bridging the Gap between Application Performance Analysis and System Monitoring

Thomas Ilsche, Mario Bielert, Christian von Elm
Center for Information Services and High Performance Computing (ZIH)
Technische Universität Dresden, 01062 Dresden, Germany
thomas.ilsche@tu-dresden, mario.bielert@tu-dresden.de, christian.von_elm@tu-dresden.de

***Abstract*—Performance analysis has a long history in the high-performance computing community. On the one hand, the traditional application analysis focuses on scalable yet detailed instrumentation of parallel execution. On the other hand, per-node or cluster-wide monitoring solutions are used in data center operation. However, performance anomalies resulting from the interaction between applications, background processes and the operating system, are difficult to analyze with tools that reveal only part of the issue. In this paper, we present a novel approach that covers all aspects of individual nodes. We extend a monitoring tool to combine call stack sampling, process monitoring, and syscall recording into a symbiotic view of the application execution, background activity, and the operating system.**

I. Introduction

Performance analysis is an important pillar for the efficient operation of high performance computing clusters. A deep understanding of practical execution is the foundation for performance optimization of applications and an energy-efficient configuration. Classical application analysis focuses on the activity within the threads and processes of one parallel application. This approach works well under the assumption that one application has exclusive access to the underlying hardware resources, at least on each individual node. In practice, however, the influence of factors outside of the application, such as operating system services, can lead to substantial variability as well as sustained performance degradation (see [1]).

Understanding and resolving such complex performance issues requires performance monitoring tools that observe applications in the context of the system and its hardware. Existing tools struggle to expose details of application execution at the same time as monitoring system events and the interaction between the application and the operating system.

In this paper, we describe improvements to the existing node-level performance monitoring tool `lo2s` to allow sampling of functions of multiple applications and their mapping to the system. Moreover, we enrich the existing information with improved monitoring of system calls (syscalls).

The remainder of this paper is structured as follows. The following section summarizes the contemporary approaches and tools for performance analysis. In Section III, we present the combination of application and per-node monitoring in `lo2s`. Further, we describe how `lo2s` can monitor the interactions between the application and the operating system through syscalls. We discuss the overhead in Section V. The last section sums up our progress and sketches future directions.

II. Related work and Background

A broad range of performance monitoring tools covers general as well as HPC-specific use-cases. For this discussion, we focus on tools that support collecting event traces and present timelines rather than exclusively summarizing information into profiles. One of the most versatile and widely available general-purpose Linux tool suites is `perf` [2, Chapter 13]. With `perf record`, it is possible to collect information about a specific application, all running processes, as well as various system information. This includes syscalls in the form of tracepoints at the beginning and end of syscall invocations. However, it is limited by a serial and very general trace file format. With the `perf` tool suite, the result can be shown as an aggregated text profile (`perf report`) or a full-text listing of events (`perf script`). Hotspot [3] offers a graphical presentation of a limited subset of traces recorded with `perf` with some support for timelines. The growing support for eBPF, a technology that enables sandboxed programs within the Linux kernel, allows profiling tools to collect specific monitoring data with reduced runtime overhead. For example, `bpftrace` [2, Chapter 15] can collect specific data from various probe types. However, it is tailored to answer concrete questions with simple outputs rather than providing a comprehensive overview of the activity.

These tools are capable of collecting data from both application execution and system events, including syscalls. However, they lack scalable recording and comprehensive visual analysis of large amounts of trace data that are inevitably generated from observing parallel applications.

In contrast, tools for HPC performance analysis specialize in scalable data recording and presentation. For example, Score-P [4] is used to collect information from large-scale OpenMP and MPI applications. Nevertheless, it is limited to the application perspective and always requires instrumentation during the build step. Score-P writes information as OTF2 traces [5], which can be visualized with Vampir [6]. HPC-Toolkit [7] focuses on sampling rather than instrumentation and can work on unmodified parallel applications. Both Score-P and HPCToolkit can use the `perf_event` interface[1] as a sampling interrupt source and to collect hardware performance counters. The Tuning and Analysis Utilities (TAU) [8] are primarily used to instrument parallel applications and generate

[1]The interface defined in `linux/perf_event.h` around the `perf_event_open` syscall is also the foundation for the `perf` suite.

profiles or traces. Morris et al. [9] demonstrate an experimental combination of instrumentation and sampling based on TAU that leverages sampling techniques from HPCToolkit. These three tools perform their sampling event handling in userspace code and are limited to monitoring a single parallel application.

Ilsche et. al [10] presented the node-level performance analysis tool `lo2s`. This versatile tool is also built on top of `perf_event` and writes traces as OTF2, enabling scalable performance visualization with Vampir. `Lo2s` can monitor applications in the process monitoring mode or all scheduled processes in the system monitoring mode. While the two monitoring modes already cover a large range of use-cases, the sampling of application call stacks was previously only possible in the *process monitoring mode*. In this work, we extended the capabilities of `lo2s` with the system-wide sampling of application call stacks, combining the advantages of both modes. Moreover, recording syscalls was only possible as metric events, making it difficult to highlight the individual syscall phases within the timeline. We improve this by adding native syscall support as individually identifiable execution regions.

III. Sampling in System-Monitoring

Sampling, as a technique for obtaining performance information, uses a periodic collection of the state of an application or the system (see also [11, Section 2.1.2]). In the *process monitoring mode* of `lo2s`, this periodic interruption is triggered by an overflow of an event counter, by default a certain number of *instructions*. On such an interrupt, `lo2s` will collect either the current instruction pointer or the call stack, and optionally additional performance metrics [12, Section III]. By leveraging the `perf_event` infrastructure, the Linux kernel performs the interrupt-based read-out and writes the data into a ring buffer. This approach minimizes the performance overhead compared to custom user-space interrupt handlers. The collected instruction pointers or call stacks yield a trace of the application execution. Compared to function instrumentation, sampling allows better control over perturbation, but the resulting traces are noisy. When observing threads with `perf_event`, the Linux kernel tracks the hardware performance counters even when the thread is suspended or migrated.

In the *system monitoring mode* of `lo2s`, `perf_event` is used to track all scheduling events of the operating system individually. This means, that the resulting trace contains accurate information on which processes were running during what time interval on which hardware thread. Besides application threads, the tracked processes include operating system services and `lo2s`' internal monitoring threads. Additional node-global events and metrics can be included: Kernel tracepoints cover a variety of system events (see [13, Section 2.9]) and are mapped to metric information in the trace. Moreover, metrics from hardware performance counters as well as various external sources can be included. As an example, Ilsche et. al [12] use tracepoints to monitor C-state transitions. They then correlate the recorded tracepoints with external high-resolution power measurements into a thorough trace file.

While both modes are individually powerful, we present an enhanced system monitoring mode that combines their advantages: Through our modification to `lo2s`, it is now possible to record instruction pointers and call stacks in the system monitoring mode.

Implementing Sampling in System Monitoring Mode

The implementation of the enhanced system monitoring mode leverages the `context_switch` option of `perf_event_open`. This newer option (since Linux 4.3) requires fewer privileges[2] to monitor scheduling events than using the `sched_switch` tracepoint. Together with replacing the `sched_process_exit` tracepoint with the `PERF_RECORD_COMM` event, our changes enable the *system monitoring mode* without special permissions or configuration. Moreover, scheduling events can be collected in the same buffer as sampling events, which is crucial for the presented enhancement. However, due to race conditions during scheduling in the kernel, it is still possible that a sample of a thread is recorded at a time when it appears to not be scheduled. In such cases, `lo2s` drops the sample in order to avoid an inconsistent stack state in the trace. Since the samples represent only statistical information, this is not a significant issue.

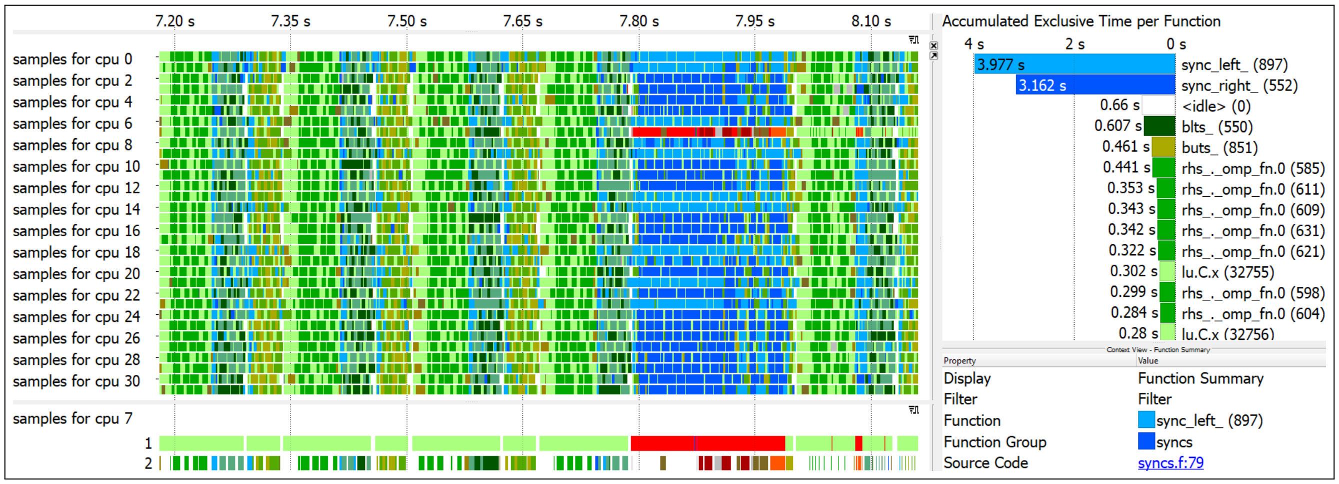
`Lo2s` needs to know the location of each process' binary and its dynamic libraries to extract the debug information for the trace. Furthermore, information about the actual process names (as opposed to the name of the binary) should be collected. In the process monitoring mode, this information is extracted from the `PERF_RECORD_MMAP` and `PERF_RECORD_COMM` events, which are generated on process start-up. However, those events are not available for processes already running before monitoring was started and are thus not sufficient for the system monitoring mode. To alleviate this limitation, `lo2s` parses `/proc/[pid]/maps` and `/proc/[pid]/comm` for every running process, given that it has adequate permissions.

An Example of Comprehensive Sampling

The following showcases the monitoring enhancement with instruction points (flat function) sampling through an exemplary performance anomaly. We run the well-known *LU* benchmark from the NAS parallel benchmark suite [14] in class C on one node comprising two Intel Xeon E5-2690 processors. Our configuration uses OpenMP on all of the 32 available hardware threads, i.e., CPU 0 to CPU 31.

Figure 1 shows a Vampir screenshot of the described situation: As expected, the OpenMP threads are scheduled on each hardware thread. The timeline shows both, functions but also the scheduled threads without function information when no instruction sample is available. For instance, the light green sections denote the threads of the `lu.C.x` process. These processes are overlayed with sections representing samples. Darker green and brown colored sections represent computation phases and synchronization phases use blue colors. During benchmark execution, the computation phases are interleaved with synchronization phases. Idle times of a CPU are visible

[2]Setting the sysctl `perf_event_paranoid` to 0 or -1 suffices



3.977 s	sync_left_ (897)
3.162 s	sync_right_ (552)
0.66 s	<idle> (0)
0.607 s	blts_ (550)
0.461 s	buts_ (851)
0.441 s	rhs_._omp_fn.0 (585)
0.353 s	rhs_._omp_fn.0 (611)
0.343 s	rhs_._omp_fn.0 (609)
0.342 s	rhs_._omp_fn.0 (631)
0.322 s	rhs_._omp_fn.0 (621)
0.302 s	lu.C.x (32755)
0.299 s	rhs_._omp_fn.0 (598)
0.284 s	rhs_._omp_fn.0 (604)
0.28 s	lu.C.x (32756)

Property	Value
Display	Function Summary
Filter	Filter
Function	sync_left_ (897)
Function Group	syncs
Source Code	syncs.f:79

Figure 1: Vampir screenshot of an OpenMP benchmark. The top shows the timeline of the function samples, if available, otherwise the scheduled process on each CPU. The bottom shows the scheduled process (1) and samples (2) for CPU 7.

in white. After ~7.8 s, we artificially schedule a disrupting process on CPU 7. The rogue process, highlighted in red, can be identified with interactive exploration in Vampir. Until that point, the synchronization phases are barely noticeable. But, the interrupting process displaces one thread of `lu.C.x` causing a delay for all threads in the following synchronization phase of the benchmark. Additionally, `lo2s` reveals that in this scenario, the threads are busy waiting rather than going into an idle state. The provided source code locations for samples of `sync_left_` and `sync_right_` show, that these functions use loops on atomics for inter-thread communication.

With classical analysis tools, users would need to choose between an application view and a process-focused view. The application view only highlights the effect on the application but lacks the cause of the degradation. While a node-level view can hint at the culprit, it lacks detailed information about the application behavior. We closed that gap with the comprehensive event sampling introduced to `lo2s`.

IV. Monitoring of System Calls

System calls, or short syscalls, are the interface between the applications and the operating system. They allow switching the control flow from the user space to the privileged kernel space. This switching allows guarding critical tasks against and sharing hardware resources between untrusted applications. This important separation, however, introduces costly context switches from user space to kernel space and back. Depending on the nature of the syscall, the control flow might even be blocked in kernel space until the correlated task gets finished. In performance analysis, syscall activity can be correlated with a lot of problems, e.g., file I/O, thread synchronization, and network communication. Hence, analyzing the syscalls can give an orthogonal view of performance problems in an application.

Implementing Syscall Recording

We extend `lo2s` to record syscall events using the `raw_syscalls:sys_enter/exit` tracepoints. Of course, a typical application uses different syscalls and not all are of interest, thus, we use Ftrace [2, Chapter 14] to enable filtering.

In order to record both enter and exit of syscalls, `lo2s` needs to set up two independent perf recordings for the two tracepoints. For a more efficient implementation, the events should be collected into the same buffer. Unfortunately, the flag `PERF_FLAG_FD_OUTPUT` is "broken since Linux 2.6.35" [15]. Instead, we set `PERF_EVENT_IOC_SET_OUTPUT`. This method, however, has the disadvantage, that it only works for events that occur on the same CPU. Consequently, our current implementation only supports syscall recording in the system monitoring mode.

This limitation has two major implications. Firstly, while the recording of syscalls per CPU has its value for some use cases, analyzing the behavior of a specific application is challenging, as syscalls are hard to correlate to their originating thread. Especially syscalls that lead to a de-scheduling of the process, e.g. *futex* and *poll*, interfere with this correlation. Secondly, when a thread calling a blocking syscall is preempted, a second process can be scheduled and execute a syscall. As OTF2 does not allow overlapping region calls, `lo2s` cannot map the true sequence of events to region enter/leave events in the trace. To mitigate this problem, `lo2s` writes an exit event for any syscall that is still in-flight right before another syscall is entered.

Another note-worthy implementation detail is the translation of syscall names. `perf_event` only provides the id of syscalls and there is no readily available look-up table. Therefore, during build, a script parses the `unistd.h` header file containing `__NR_syscall_[name]` macros with the id as value.

Given these limitations, we want to expand the implementation in the future to allow syscall recording per process. This would alleviate both problems, as one thread can only call one syscall at a time. However, to achieve this, we need to go back to two separate event buffers and merge during buffer flushes in `lo2s`. For this implementation, it is important to ensure that the introduced perturbation is minimal.

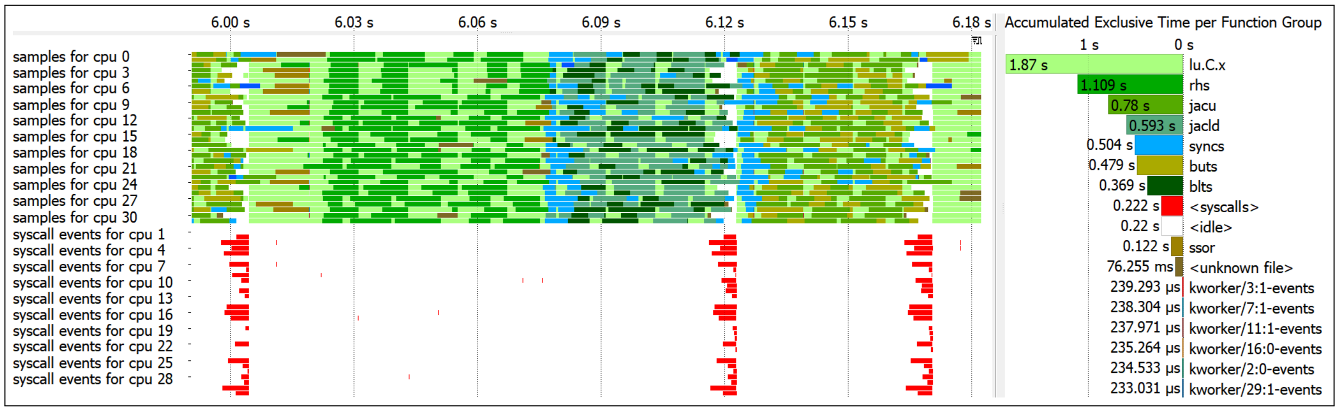


Figure 2: Vampir screenshot of the *LU* benchmark running on a 16C/32T machine. Compute phases are colored in shades of green and brown, synchronization in blue. The recorded syscall `futex` is red.

A Syscall Timeline Example

Likewise to the node-wide sampling, we use the *LU* benchmark as a synthetic example for syscall recording. In this case, we enable the recording of syscalls and zoom into one complete iteration of the benchmark. Figure 2 depicts the Vampir screenshot of the resulting situation. The upper half of the timeline is again the samples of the hardware threads. It shows the three computational phases, i.e., `rhs`, `jacld/blts`, and `jacu/buts`. The lower half shows the syscall events; in this case, only `futex` syscalls are present. In particular, the aligned `futex` calls are present between different phases, which hints at a global synchronization. And indeed, the source code of the LU benchmark uses OpenMP barriers. Also, during the `futex` syscalls, the `lu.C.x` process gets de-scheduled as opposed to the busy waiting functions discussed in Section III.

This shows how enriching the traces with syscall information can yield additional insight into the application and its interaction with the operating system. Even more so, together with the comprehensive sampling, the syscall events allow inferring implementation details of the application and its libraries and runtimes. Hence, not only the presence of syscall events can provide more context, but also the lack thereof.

V. Overhead Discussion

A critical aspect of a performance analysis tool is its impact on the observed workload, i.e., the perturbation. Given that `lo2s` utilizes the data collection provided by the kernel rather than a user-space interrupt handler, we expect a low perturbation. As a metric, we use the mean core runtime reported by the benchmark, which does not include the cost of setup and post-processing. We compare *LU* running unmonitored with `lo2s` in the system monitoring mode and with `lo2s` also recording all syscalls, each repeated for 200 times. Due to the increased variances, no rogue process was injected for this measurement. With the default of one sample every 11 010 113 instructions, the perturbation is just barely statistically detectable at 0.3 % without and with syscall recording. Hence, we reduce the sampling interval to ∼500 µs but increase the kernel buffer size to allow it to hold all samples of one run[3]. In this configuration, the perturbation is 3.3 % without and with syscall recording.

For a rough comparison, Morris et al. [9, Table I] report 7.7 % and 7.2 % overhead for TAU and HPCToolkit, respectively. While these numbers refer to the same sampling rate of 500 µs, they are limited to the application threads but use a full call stack and a different application.

VI. Conclusion and Future Work

In this paper, we presented two major extensions to the node-level performance analysis tool `lo2s`. First, the novel combination of application sampling and per-CPU monitoring allows pinpointing the culprit and impact of performance degradations in complex setups. We presented an exemplary scenario where the introduced method reveals the cause and effect of a rogue process disturbing an application run. Second, we added an improved recording of syscalls. The resulting timeline shows syscalls similarly to functions with a type, begin, and end, but at separate locations for each CPU. The presented changes are part of the v1.6.0 release[4] of `lo2s` and are publicly available under the open-source license GPLv3.

As a next step, we will make syscall recording available for application monitoring since it can be challenging to track blocking system calls across wait phases and process migrations. Moreover, we want to leverage syscall recording to monitor file-descriptor-based I/O operations in applications.

With these enhancements, `lo2s` is becoming a comprehensive node-level performance analysis tool. The unique combination of application, operating system, and hardware perspective facilitates complex use-cases while retaining the lightweight approach with low perturbation and no separate instrumentation phase.

Acknowledgments

This work is supported in part by the German Research Foundation (DFG) within the CRC 912 - HAEC and the german National High Performance Computing (NHR@TUD).

[3]`lo2s -m 2048 -e cpu/cpu-cycles -c 1450000 -A`
[4]https://github.com/tud-zih-energy/lo2s/releases/tag/v1.6.0

References

[1] S. Chunduri, K. Harms, S. Parker, V. Morozov, S. Oshin, N. Cherukuri, and K. Kumaran, "Run-to-run variability on xeon phi based cray XC systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, nov 2017.

[2] B. Gregg, *Systems Performance*. Pearson Academic, 2020.

[3] M. Wolff, "hotspot – a gui for the linux perf profiler." [Online]. Available: https://www.kdab.com/hotspot-gui-linux-perf-profiler/

[4] A. Knüpfer, C. Rössel, D. a. Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony, W. E. Nagel, Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, S. Shende, R. Tschüter, M. Wagner, B. Wesarg, and F. Wolf, "Score-P: A joint performance measurement run-time infrastructure for Periscope, Scalasca, TAU, and Vampir," in *Tools for High Performance Computing 2011*, H. Brunst, M. S. Müller, W. E. Nagel, and M. M. Resch, Eds. Springer Berlin Heidelberg, 2012, pp. 79–91.

[5] D. Eschweiler, M. Wagner, M. Geimer, A. Knüpfer, W. E. Nagel, and F. Wolf, "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*, ser. Advances in Parallel Computing, vol. 22, 2012, pp. 481 – 490.

[6] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach, "VAMPIR: Visualization and analysis of MPI resources," *Supercomputer 63*, vol. XII, no. 1, pp. 69–80, 1996.

[7] L. Adhianto, S. Banerjee, M. W. Fagan, M. W. 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*, 2009.

[8] S. S. Shende and A. D. Malony, "The tau parallel performance system," *The International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, 5 2006.

[9] A. Morris, A. D. Malony, S. Shende, and K. Huck, "Design and implementation of a hybrid parallel performance measurement system," in *2010 39th International Conference on Parallel Processing*. IEEE, 9 2010.

[10] T. Ilsche, R. Schöne, M. Bielert, A. Gocht, and D. Hackenberg, "lo2s—multi-core system and application performance analysis for Linux," in *Cluster Computing (CLUSTER), 2017 IEEE International Conference on*. IEEE, 2017, pp. 801–804.

[11] T. Ilsche, J. Schuchart, R. Schöne, and D. Hackenberg, "Combining instrumentation and sampling for trace-based application performance analysis," in *Tools for High Performance Computing 2014*. Springer International Publishing, 2015, pp. 123–136.

[12] T. Ilsche, R. Schöne, P. Joram, M. Bielert, and A. Gocht, "System monitoring with lo2s: Power and runtime impact of c-state transitions," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2018, pp. 712–715.

[13] B. Gregg, *BPF Performance Tools*. Addison-Wesley Professional, 2019.

[14] 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 *et al.*, "The NAS parallel benchmarks summary and preliminary results," in *Supercomputing'91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*. IEEE, 1991, pp. 158–165.

[15] *perf_event_open(2) – Linux Programmer's Manual*, 4th ed., September 2017.