GI-Fachgruppentreffen 2014 - Proposed Talk:

Efficient Full System Memory Tracing with Simutrace

Marc Rittinghaus Thorsten Groeninger Frank Bellosa

System Architecture Group
Karlsruhe Institute of Technology (KIT)
firstName.lastName@kit.edu

Traces are frequently used in the development and evaluation of software components and operating systems. During execution, events and state information of interest are captured in traces for later offline analysis. Traces can provide valuable insight into the dynamic behavior of a software and deliver empirical support to focus optimization and debugging efforts.

Depending on the intended use of a trace, the type and number of collected events and properties vary. *Memory traces*, that is recordings of a processor's memory accesses, leap out. While they proved to be very effective for driving memory hierarchy simulations [7, 8, 10, 11, 12, 13, 16, 17] or gathering statistics about an application's memory access patterns [14, 18], memory traces pose an extraordinary demand on the tracing components. This is due to the fact that the rate at which new events are generated—i.e., the rate at which the processor accesses main memory—is inherently higher than the rate of function calls, MPI messages or other system events that are typically recorded. Memory traces thus quickly become very large in size, consuming gigabytes of storage. A Linux virtual machine (VM) running a minimal Linux kernel build for instance produces approximately 145 billion trace entries. The same system completing the Postmark benchmark from Phoronix Test Suite [2] even generates around 175 billion write events alone. Memory traces therefore heavily depend on an efficient encoding, a scalable trace format and a tracing mechanism that is capable of dealing with a high rate of incoming events.

Over the years, various memory tracing frameworks have been developed [4, 7, 13, 15, 20]. A major limitation of these frameworks is however their restriction to track only selected processes and their inability to monitor privileged system components. Profiles generated with these tools therefore do not encompass memory references performed by the operating system (OS), system daemons or (kernel-mode) drivers. That raises questions concerning the accuracy of results obtained through such narrow traces [6] as the interaction of tracked processes and the system is completely left out. Further distortions can originate from the incurred slowdown through instrumentation and tracing, which influences the relative timing between processes and the system. Tracing tools able to capture events in the OS kernel [3, 9, 19] on the other hand do not offer memory tracing capabilities. These limitations make current tracing frameworks not applicable to memory centric *operating system research*.

Contribution. We present **Simutrace**, a novel tracing toolkit, which has been conceived with full length, no-loss memory tracing in mind. Simutrace captures memory accesses at the hardware level using *functional full system simulation*, thus including all user-space programs, the operating system, drivers and direct memory access (DMA) operations. Through the use of full system simulation, Simutrace fully supports tracing of just-in-time (JIT) as well as self-modifying code and requires no special modification or preparation of the target system. Tracing is non-intrusive, as the simulation preserves the timing between running workloads and the underlying OS.

Simutrace employs an aggressive but fast compression of recorded traces by using a modified version of VPC4 [5]—one of the best memory trace compressors reported in current literature—and combining it with LZMA [1]. At the same time, Simutrace implements a scalable and flexible storage format, which easily handles traces of hundreds of gigabytes in size while maintaining fast access through partial decompression. Moreover, in-memory indices based on the timing information in the trace allow to quickly locate specific time spans.

Simutrace uses a flexible streaming model to separate trace entries of different types, thus allowing traces to contain arbitrary data—including variable-sized data. We take advantage of this feature by supplementing memory access traces with introspection information such as process creation/destroy or context switch events. That allows us to correlate memory accesses in the simulation with the originating process.

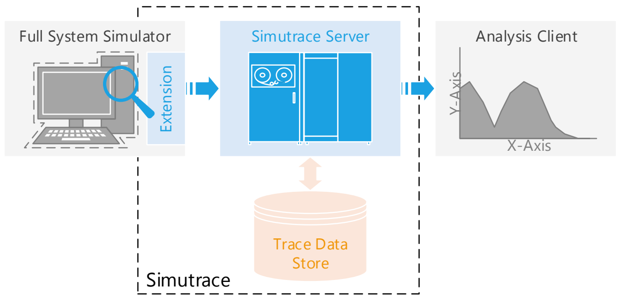

Figure 1: An extension in a full system simulator collects events. Simutrace receives the data, stores it and provides access for later analysis and inspection.

Talk. The proposed talk will cover the concepts behind Simutrace and provide a more detailed insight into the challenges of memory tracing such as a scalable storage format, effective compression and flexibility in the type of recordable data. To round off our talk, we will give an overview of practical research, where we used Simutrace to analyze the characteristics of redundant memory pages.

1. REFERENCES

[1] Lzma sdk. `http://www.7-zip.org/sdk.html`.

[2] Phoronix test suite. `http://www.phoronix-test-suite.com/`.

[3] Solaris dtrace. `http://wikis.oracle.com/display/DTrace/DTrace`.

[4] S. Budanur, F. Mueller, and T. Gamblin. Memory trace compression and replay for spmd systems using extended prsds. *The Computer Journal*, 55(2):206–217, 2012.

[5] M. Burtscher, I. Ganusov, S. J. Jackson, J. Ke, P. Ratanaworabhan, and N. B. Sam. The vpc trace-compression algorithms. *Computers, IEEE Transactions on*, 54(11):1329–1344, 2005.

[6] H. W. Cain, K. M. Lepak, B. A. Schwartz, and M. H. Lipasti. Precise and accurate processor simulation. In *Workshop on Computer Architecture Evaluation using Commercial Workloads, HPCA*, volume 8, 2002.

[7] L. DeRose, K. Ekanadham, J. K. Hollingsworth, and S. Sbaraglia. Sigma: A simulator infrastructure to guide memory analysis. In *Supercomputing, ACM/IEEE 2002 Conference*, pages 1–1. IEEE, 2002.

[8] J. Edler and M. D. Hill. Dinero iv trace-driven uniprocessor cache simulator, 1998.

[9] P.-M. Fournier, M. Desnoyers, and M. R. Dagenais. Combined tracing of the kernel and applications with lttng. In *Proceedings of the 2009 linux symposium*, 2009.

[10] M. Holliday. Techniques for cache and memory simulation using address reference traces. *International journal in computer simulation*, 1(1):129–151, 1991.

[11] H. Kang and J. L. Wong. vcsimx86: a cache simulation framework for x86 virtualization hosts. 2013.

[12] J. Marathe, F. Mueller, T. Mohan, B. R. de Supinski, S. A. McKee, and A. Yoo. Metric: Tracking down inefficiencies in the memory hierarchy via binary rewriting. In *Code Generation and Optimization, 2003. CGO 2003. International Symposium on*, pages 289–300. IEEE, 2003.

[13] J. Marathe, F. Mueller, T. Mohan, S. A. Mckee, B. R. De Supinski, and A. Yoo. Metric: Memory tracing via dynamic binary rewriting to identify cache inefficiencies. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(2):12, 2007.

[14] R. C. Murphy and P. M. Kogge. On the memory access patterns of supercomputer applications: Benchmark selection and its implications. *Computers, IEEE Transactions on*, 56(7):937–945, 2007.

[15] A. Snavely, N. Wolter, and L. Carrington. Modeling application performance by convolving machine signatures with application profiles. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 149–156. IEEE, 2001.

[16] R. A. Uhlig and T. N. Mudge. Trace-driven memory simulation: A survey. *ACM Computing Surveys (CSUR)*, 29(2):128–170, 1997.

[17] J. Weidendorfer, M. Kowarschik, and C. Trinitis. A tool suite for simulation based analysis of memory access behavior. In *Computational Science-ICCS 2004*, pages 440–447. Springer, 2004.

[18] J. Weinberg, M. O. McCracken, E. Strohmaier, and A. Snavely. Quantifying locality in the memory access patterns of hpc applications. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 50. IEEE Computer Society, 2005.

[19] R. W. Wisniewski and B. Rosenburg. Efficient, unified, and scalable performance monitoring for multiprocessor operating systems. In *Supercomputing, 2003 ACM/IEEE Conference*, pages 3–3. IEEE, 2003.

[20] Q. Zhao, R. Rabbah, S. Amarasinghe, L. Rudolph, and W.-F. Wong. Ubiquitous memory introspection. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 299–311. IEEE Computer Society, 2007.