Using Tracing to Diagnose or Monitor Systems

Dominique Toupin

With the increasing complexity of systems and the pervasive use of multicore technology, increasing numbers of problems can only be diagnosed via tracing tools. Author Dominique Toupin introduces the open source Linux Trace Toolkit Next Generation (LTTng) and describes how to use it. I look forward to hearing from both readers and prospective authors about this column and the technologies you want to know more about. — Christof Ebert

SOFTWARE TRACING IS an efficient way to record information about a program’s execution. Programmers, experienced system administrators, and technical support personnel use it to diagnose problems, tune for performance, and improve comprehension of system behavior. Tracing is ideally suited for applications in which several processes interact with the operating system (OS) and with each other, or where time behavior is important. Examples of such systems include

- online servers with heavy interactions between several cooperating processes, the OS, the network, and the disks;
- embedded real-time systems for machine and vehicle control, telecommunication systems, cellular phones, and multimedia pocket computers; and
- desktop computers running several cooperating processes for the GUI, the window manager, the sound server, and the wireless network manager.

Tracing can also assist in monitoring system health and enhancing security via intrusion detection. Compared to logging, tracing typically records events that occur much more frequently. Tracers must therefore be optimized to handle a lot of data while having the smallest possible impact on the system and allowing access to almost any information while the system runs.

An area in which tracing has proven to be particularly useful is multicore systems. Realizing the full benefit of multicore architectures normally requires a software redesign (to introduce more parallelism). Consequently, developers face problems whose resolution requires understanding interactions between all layers, including...
third-party products such as hypervisors, OSs, virtual machines, system libraries, and applications. These layers might also be written in different languages, such as C/C++, Java, and Erlang, or run on different hardware, such as general-purpose CPUs, DSPs, and so on. Furthermore, many problems are only reproducible when hardware and software interact under real workloads, during which some tools can drastically change behavior and therefore prevent problem reproduction. In such situations, a tracer is the only tool that can properly diagnose those problems.

This article illustrates the most desirable characteristics of a tracing tool, using the Linux Trace Toolkit Next Generation (LTTng) and an Eclipse-based analysis tool as examples.

**LTTng**

LTTng is a highly efficient, full-system solution that allows tracing of bare metal, kernel, and user space as well as trace viewing, analysis, and data streaming. LTTng helps developers, testers, system administrators, and field engineers understand what’s happening in a system when other tools make the program fail or when a large amount of data (gigabytes) needs to be recorded. LTTng has a broad range of end users and open source contributors. The characteristics described here are needed to trace a wide range of systems and are not Linux specific.

**Static and Dynamic Tracepoints**

LTTng uses static tracepoints, which are hooking mechanisms that can be activated at runtime to record information about a program execution. Static tracepoints represent the wisdom of developers who are most familiar with the code; the rest of the world (testers, system administrators, field engineers, developers) can use them to extract a great deal of useful information without having to know the code. Because they’re already available in the Linux kernel and in many Linux applications, there’s no need to recompile, stop a program, or reboot; static tracepoints are activated at runtime. When program execution hits an active tracepoint, the connected probe is called and the execution continues when the probe returns. A naive implementation of tracepoints could result in code somewhat equivalent to the following:

```
if (tracepoint_1_activated)
  (*tracepoint_1_probe)(arg1, arg2);
```

In practice, the implementation can achieve several different optimizations and enhancements—for example, by...
using a static jump to check the activation status, giving a hint to the compiler, or placing the probe call setup instructions at the function end to diminish instruction cache pollution. (For more information, see the “Static Jump” sidebar). It’s also possible to connect several probes to the same tracepoint, and various operations such as probe connection or tracepoint activation are thread safe. With these optimizations, the overhead of adding an inactive static tracepoint is practically zero. For active static tracepoints, the overhead is an order of magnitude lower than dynamically inserting new tracepoints. Static tracepoints are therefore used to extract a large amount of data with the lowest possible overhead.

Because it’s impossible to predict all tracing needs, it’s sometimes necessary to complement static tracepoints by dynamically adding new tracepoints—that is, without recompile. Dynamic tracepoints are inserted via a system call, trap, or dynamic jump, and added in the Linux kernel via kprobe or in the user space via the GNU debugger (GDB) tracepoints. The number of dynamic tracepoints should be kept to a minimum to achieve low overhead.

**Trace Data Storage**

Trace data is initially stored in shared memory buffers; a tracing daemon then handles the chosen trace store, which can be a circular “flight recorder,” the local disk, a remote disk, or a remote stream. The trace format is a self-describing highly optimized binary format that can store variable-sized events, each recording an arbitrary number of arguments. The tracing daemon uses specialized algorithms such as splice to avoid making any copy of the data from event generation to disk write. A common trace format is now available for not just Linux-based systems but also for bare metal and other OSs.

**Scalability**

The LTTng tracer scales to high core numbers by using the wait-free read-copy-update (RCU) mechanism, per-CPU buffers, and nonblocking atomic operations. Wait-free RCU provides read-side access, which allows multiple copies of a given data structure to live simultaneously and monitors data structure accesses to detect grace periods after which memory reclamation is possible. Performance on multicore systems is at least five times better than dynamic tracing, a margin that increases with the number of cores. Figure 1 summarizes some of the important characteristics.

**Eclipse Trace Analysis**

Analysis tools have views that can help troubleshooters (developers, testers, system administrators, or field engineers) to understand traces. Eclipse is an open platform composed of extensible frameworks, tools, and runtimes. It provides many benefits for a trace analysis tools such as integration with the integrated development environment (IDE). LTTng traces can be analyzed with an Eclipse-based tool that provides a trace analysis framework with a few already built-in options to help understand latency, system state through time, scheduling, process state, hardware state, and other logical resources. Troubleshooters can perform analyses on a trace such as computing the causality links between events, thus finding the time-critical path between two events, or searching for specific patterns such as excessive swapping, spurious timeouts, or overloaded disk subsystems. Additional capabilities include filtering, synchronization of data, correlation, live streaming, and handling of traces.
larger than 10 Gbytes. Accurate event ordering is key to enabling trace synchronization or correlation of traces taken from different CPUs, cores, nodes, virtual machines, and so on. The LTTng timestamp precision is in the 1- to 100-ns range, which enables precise correlation of tracing data. Figures 2 and 3 give examples of LTTng views.

Other views include a detailed event list and an event statistics view showing statistics per event type, per resource, or per CPU. In addition to LTTng traces, the Eclipse tracing framework...
provides several features for text format traces: custom parsers to handle text or XML, a parser generator for custom text format, data visualization, filtering, searching, correlation, and comparison. The framework also provides a module to integrate custom analysis.

With increasing system complexity comes an increasing number of problems that can only be diagnosed via tracing tools. Fortunately, the Linux kernel’s built-in static tracepoint is facilitating the resolution of such problems. For programs running in the user space, the same highly efficient tracing techniques have been ported from the kernel through the LTTng User Space Tracer (UST). With UST, the common trace format, and the addition of static tracepoint in many user space programs, the software industry has all the ingredients to create a new de facto standard to ease the resolution of some truly tough problems.

DOMINIQUE TOUPIN manages developer tools at Ericsson. His interests include debug, trace, edit/compile/build, and static analysis. Toupin previously developed systems to manage cellular networks. He holds an MS in software engineering from Ecole Polytechnique Montreal. Contact him at dominique.toupin@ericsson.com.

FIGURE 3. Event rate histogram. This view displays event distribution for a whole trace and controls which event are shown in other views.

C A L L  F O R  P A P E R S

Special Issue on Engineering Fun

Publication: September/October 2011
Submission Deadline: 1 February 2011

As large and complex software projects, modern computer games pose many software engineering challenges, with complex and performance-intensive design and implementation choices. As entertainment products, games also rely heavily on less well-defined, abstract properties such as playability and fun. The influence of soft, thematic, and aesthetic requirements on precise and practical designs introduces a variety of interesting development constraints and goals. This requires novel solutions to design, workflow, and specific technical problems, but it also allows approaches that optimize performance by exploiting the greater latitude afforded to entertainment products.

In this special issue, we invite researchers and practitioners to submit work that addresses the practical development of modern computer games. We especially invite papers that consider the multimedia nature, requirements, and abstract playability goals of game development, their integration and development within a complex, performance-oriented software context, as well as potential transfer to other software domains. Topics include but are not limited to

- Game design, software architectures
- Workflow: multimedia integration, prototyping systems
- Development processes, modular component design and integration
- Console, hand-held, multiplatform requirements
- Quality assurance, monitoring and preserving game properties
- Techniques for ensuring immersive game-play
- Game AI: perception-based, simplified designs
- Multiplayer and networked systems: designs, fault-tolerant and scalable techniques

Questions?
For more information about the focus, contact the guest editors:
- Clark Verbrugge, McGill University, clump@cs.mcgill.ca
- Paul Kruszewski, GRIP Entertainment, paul.kruszewski@gripent.com

For the full call for papers: www.computer.org/software/cfp5
For general author guidelines: www.computer.org/software/author.htm
For submission details: https://mc.manuscriptcentral.com/sw-cs