

Efficient and Accurate Tracing of Events in Linux Clusters

Michel Dagenais^a
Richard Moore^b
Robert Wisniewski^c
Karim Yaghmour^d
Thomas Zanussi^e

^aEcole Polytechnique de Montreal

^bIBM Linux Technical Center

^cIBM Research

^dOpsys Inc.

^eIBM Linux Technical Center

The Linux Trace Toolkit is a low overhead, accurate, extensible tracing infrastructure for Linux systems. It is used extensively for embedded systems and is included in all major Linux distributions targeting embedded systems. It was recently optimized and expanded to offer per CPU lockless tracing and near clock cycle level accuracy, making it suitable for high performance SMP systems. Further work is underway to manipulate several traces side by side, typically collected on parallel systems within a cluster, and to allow custom trace analysis plugin modules. This makes the Linux Trace Toolkit particularly suited for performance analysis and debugging of high performance SMP clusters. Indeed, it can collect unobtrusively and most accurately any needed information directly from the operating system or the applications and offers a convenient offline graphical display and analysis tool.

1 INTRODUCTION

Several tools may be used for performance analysis. Coverage analysis tools [1] obtain a precise count of instructions executed in a program, a strong indication of the processing time required. Profilers [2] use sampling at regular time intervals to statistically determine the time spent in each function. Some profilers use hardware performance counters [3] to obtain program counter samples correlated to metrics other than CPU time, for instance cache misses, mispredicted branches or pipeline stalls. For CPU intensive applications, these tools produce useful and reasonably accurate measurements.

However, this may not be sufficient for applications, or portions thereof, where interactions between one or more processes and the operating system impact significantly the performance. In many cases, resources managed by the operating system may indeed strongly affect the performance. This includes disk heads scheduling, I/O buffer cache, virtual memory management, signal delivery, interprocess communication, network I/O and locking.

Tracing all system calls for one or more processes may be achieved through the Linux *ptrace* system call, for instance using *strace* or *GDB*. This is often helpful to understand the interactions between the processes and the operating system. However, in many cases, the timing is changed significantly, producing grossly inaccurate performance data and even changing the

behavior of asynchronous processes.

2 THE LINUX TRACE TOOLKIT

The Linux Trace Toolkit (LTT) [4] offers a low overhead detailed trace of all significant operating system events. Every time a system call, interrupt, scheduling change or other important operating system event occurs, a few bytes are written to a trace buffer. In contrast, the *ptrace* interface requires two or more context switches for each event, and has only access to system call and signal events. When the trace buffer is full, tracing continues in another buffer while the first buffer is copied asynchronously to disk. The trace daemon, responsible for writing the trace buffer to disk, has shared memory access to the in kernel trace buffer, thus avoiding any buffer copying in memory.

The resulting performance allows full tracing of a system with heavy interactive load (which is very demanding for the tracing system) with an overhead of less than 5% [4]. The Linux Trace Toolkit has been used successfully to study the performance of the Linux virtual memory subsystem and of many real time applications in embedded Linux systems.

The Linux Trace Toolkit is currently being enhanced in several ways, and is becoming even better suited for high performance Symmetric Multi-Processor (SMP) systems and clusters. The enhancements recently completed or under way are described in the following sections.

2.1 Functionality Layering

The Linux Trace Toolkit is currently distributed as a separate patch to apply to the Linux kernel, thus requiring downloading the Linux kernel source and LTT, and recompiling the patched kernel. It has been submitted for inclusion in the Linux Kernel since it is very useful in its own right, and it could serve as an efficient underlying tracing mechanism for other existing applications such as `printk` and `Evlog` [5].

While all these applications need to efficiently accumulate trace information and communicate it to a user level daemon, their different scope may motivate different choices in terms of event creation API, trace format, time granularity and notification mechanisms. For this reason, a new *data relay* file system, *relays*, was created to handle in a generic way the creation, buffering and delivery of data between the kernel and a user level daemon.

When a *relays* object is created in the kernel, buffers are allocated and a new filename becomes visible in the *relays* filesystem. The *relays* object is then used to write data as needed. Whenever a buffer start or buffer end is encountered, the *relays* object activates callback functions. These functions are specified at creation time and may be used to write any application specific (e.g., LTT, Evlog) buffer start and buffer end data.

The file associated with a *relays* object may be opened by a user level daemon, subject to usual file access permissions. The opened file may then be used in one of two modes. Read events from the file one by one, blocking or non-blocking. Memory map the file and thus obtain shared memory access to the trace buffers as well as notification each time a buffer is full.

Relays offers all the functionality required by LTT, yet is equally applicable to other logging or tracing systems such as `printk` and `Evlog`. Indeed, the exact trace format, time granularity, or number of buffers per CPU is unspecified at this level.

2.2 Minimizing Contention in SMP Systems

The latest stable release of LTT used a single set of tracing buffers, used by all processors in a SMP system, and protected by locking. This was replaced in the development version by lockless, per CPU, trace buffers. When an event arises on a processor, space is reserved in the associated trace buffer using an atomic operation, and the event data is written to the reserved space. If the process is interrupted before writing the data, the space in the buffer remains properly reserved, and any other event created gets space allocated sequentially in the trace buffer. Thus, there is no blocking lock involved, and tracing is allowed everywhere, even in interrupt routines.

Events created on different processors within a SMP system are written to separate, per CPU, trace buffers.

No global locking is therefore necessary and cache conflicts are avoided, unlike with shared trace buffers.

2.3 Time granularity

The latest stable release of LTT uses the real time clock (`gettimeofday`) as timestamp for each event. The cost of calling `gettimeofday` is non negligible and its time granularity is limited. The development version now directly reads the processor cycle counter, if available. Cycle counters are fast to read but may reflect time inaccurately. Indeed, the exact clock frequency varies with time as the processor temperature changes, influenced by the external temperature and its workload. Moreover, in SMP systems, the clock of individual processors may vary independently.

LTT corrects the clock inaccuracy by reading the real time clock value and the 64 bits cycle counter periodically, at the beginning of each block, and at each 10ms. This way, it is sufficient to read only the lower 32 bits of the cycle counter at each event. The associated real time value may then be obtained by linear interpolation between the nearest full cycle counter and real time values. Therefore, for the average cost of reading and storing the lower 32 bits of the cycle counter at each event, the real time with full resolution is obtained at analysis time.

2.4 Tracing at all Times

LTT currently starts tracing when enabled by the tracing daemon, and the trace is obtained when filled event buffers are written to disk. There are many difficult to diagnose problems which occur at other times and cannot easily be traced using the current version of LTT.

Boot time tracing will be obtained by allocating a static buffer to receive events until the tracing facility itself is properly initialized. The events will then be copied from the static buffer to the regular LTT buffers.

Similarly, when a system crashes, the tracing daemon does not have time to save on disk the events in the yet unfilled tracing buffer. Yet, those are often the most interesting events for the system engineer. By integrating LTT to the Linux Kernel Crash Dump facility [6], it will be possible to recover from memory the unfilled buffer upon the soft reboot following the crash.

Transient problems are another difficult category to study. Enabling tracing for long periods of time incurs a reasonable system overhead, but the disk space required for the trace soon becomes prohibitive, as it may reach 1MB/s. The solution is to periodically overwrite the trace data, only keeping the last few minutes or the last hour. Whenever the transient problem is encountered, the tracing may be stopped before overwriting the events associated with the problem just

noticed.

2.5 New Events

Event types in LTT are currently defined as packed C structures in the kernel, and accessed using corresponding structure definitions in the LTT visualiser. Custom event types may be defined at run time using a textual description, but creating custom events is slightly less efficient than builtin event types. Adding new builtin event types requires modifying and recompiling both the kernel and the LTT visualiser.

In development is a simple event type description format, used at kernel compilation time to generate the needed event type declarations. The same description is used at runtime by the LTT visualiser to properly interpret event types in traces. Checksums are used to verify that the event type descriptions provided to the trace visualiser are those used in the kernel which generated the trace to visualise.

This new scheme even allows dynamically loaded modules to define new event types, or new versions of event types.

2.6 Adding Analysis Tools

As it became more widely disseminated, LTT has been used for many new applications. In each case, new event types and new specialised computations were required. Examples include the detailed analysis of process elapsed time (e.g., 3.56s waiting for process X, 1.24s waiting for page faults on file Y...) and a study of average block device queue length and processing time. Under development is a tool to correlate traces from several communicating nodes within a cluster.

Each specialised analysis requires the insertion of new code in the LTT visualiser main processing loop, a software development and maintenance nightmare. To solve this problem, the visualiser is being rearchitected. The new modular design allows dynamic loading of modules and provides hooks at numerous places to let the modules perform conveniently the desired computations. Furthermore, a hierarchical *property* table is associated with each important object in the visualiser (trace, process, CPU, device...). This way, modules can store and accumulate computed statistics as needed during the trace processing phase, and print the results at the end, all using generic hooks. This keeps the modules completely decoupled from the visualiser.

3 DISCUSSION

The Linux Trace Toolkit is widely used in embedded Linux development. It can already provide very useful information for performance measuring and understanding in high performance SMP distributed systems. As the new discussed features are completed,

it will become even more accurate, less intrusive and easier to use in high performance SMP distributed systems.

Tracing systems have existed in the past. The distinguishing feature of LTT is the freely available source code of both the operating system and the tracing tool. Not only can LTT benefit from contributions of the community, it enables its interoperability with other useful tools like Dynamic Probes [7] and the Linux Kernel Crash Dump facility [6]. Moreover, it is possible to add tracepoints anywhere in the Linux kernel and thus obtain precise traces for any desired event in Linux.

REFERENCES

1. Gcov a test coverage program, 14 March 2003. <http://gcc.gnu.org/onlinedocs/gcc-3.2.2/gcc/Gcov.html>.
2. Gnu gprof, 14 March 2003. <http://sources.redhat.com/binutils/docs-2.12/gprof.info/index.html>.
3. Oprofile, 14 March 2003. <http://oprofile.sourceforge.net>.
4. Karim Yaghmour and Michel R. Dagenais. Measuring and characterizing system behavior using kernel-level event logging. In *Proceedings of the USENIX Annual 2000 Technical Conference*, pages 13–26, San Diego, California, USA, June 2000.
5. Linux event logging for enterprise-class systems, 14 March 2003. <http://evlog.sourceforge.net>.
6. Linux kernel crash dumps, 14 March 2003. <http://lkcd.sourceforge.net>.
7. Dynamic probes, 14 March 2003. <http://www-124.ibm.com/developerworks/oss/linux/projects/dprobes>.